Shortest Path Queries in Large Time-Dependent Road Networks

Ole Borup

Master Thesis
University of Copenhagen
Department of Computer Science

February 15 2008

Abstract

Solution algorithms for the static shortest-paths problem in road networks have undergone rapid development in recent years, leading to methods that are up to one million times faster than the classical algorithm by Dijkstra, and able to answer shortest-paths queries for whole continents within microseconds. In reality, however, road networks tend to have dynamic characteristics which require more sophisticated approaches.

In this thesis we study the time-dependent shortest-paths problem where edge weights are a predictable function of time. A shortest-path is found with regard to the weight changes that happens as one travels through the network. The problem is generally $\mathcal{NP}$-hard, but if the network admits a FIFO property, natural to assume for road networks, the problem is polynomially solvable. We try to adapt existing speed-up techniques for the static shortest-paths problem to find efficient solution algorithms for large time-dependent road networks. Two goal directed methods are adapted: A* search using landmarks and the triangle inequality, and precomputed cluster distances. Furthermore the two methods are combined to give an additional speed-up.

Time-dependent road networks with real reliable travel time forecasts are not yet available, but it is believed that progress is being made. To test the developed algorithms a method to find simulated time-dependent road networks with some touch of realism is given. An experimental evaluation of the developed speed-up techniques shows that all methods are highly sensible to the level of delay imposed by travel time functions, compared to static travel times.

Contents

Chapter 1

Introduction

Consider planning a car trip in a road network from an origin to a destination, using a route planner like a GPS or an online service. The route planner will find the fastest route, using some estimated static travel times for each road segment, and give some estimated total travel time for the trip. The road network is represented as a set of intersections and road segments connecting the intersections. Now consider actually driving from the origin at a specific departure time to the destination, using the proposed route. It may be discovered that the estimated travel time does not hold, because of traffic jams or other sources of congestion. The proposed route may be the fastest if driven at times when no congestion occur, but for certain departure times the route will result in delay, for example during rush hours.

Imagine now, that an oracle exists, that for each road segment can give an estimated travel time, given the departure time. That is, the oracle is a function that given road segment and departure time, returns an estimated travel time. Maybe this oracle could be used to find an alternative route that was fastest for the given departure time. Figure 1.1 shows two alternative routes in the Greater Copenhagen area, with the blue one being the fastest when no congestion occurs, and the red one being an alternative that may be faster for certain departure times.

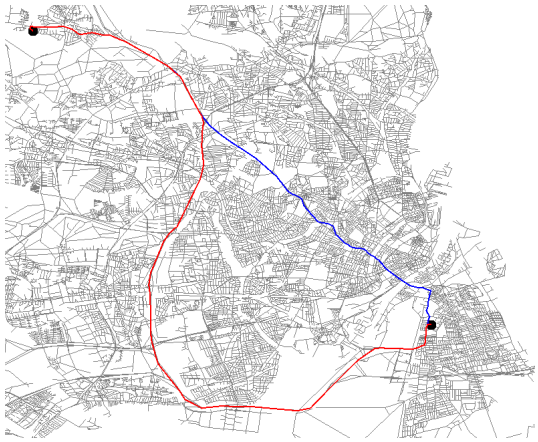


Figure 1.1: Two alternative routes in the Greater Copenhagen area.

In practice such an oracle does not exist, but recent advent in GPS and mobile technology may in the future allow collection of enough data to make realistic forecasts on travel time for road segments. For example by equipping vehicles with GPS and transmitters that sends back observed information on the travel time at specific times to a central database.

The underlying problem of finding fastest paths in a road network with static travel time on road segments is one of the most studied problems in algorithmic graph theory. For road networks,

recent research has led to algorithms that can determine fastest paths within microseconds for whole continents. When travel time on road segments become a function of time, we have a *time-dependent* problem, where a fastest path is with regard to the travel time changes that happens as one travels through the network.

Time-dependent shortest-paths problems have existed mainly in other areas like public transportation networks, but we believe that data is going to be available to allow real reliable travel time forecasts for road networks. In this thesis we treat various aspects of finding shortest-paths in time-dependent road networks. By shortest we will always mean fastest. We will explore how to represent travel time functions, and how to solve the time-dependent problem efficiently enough to allow interactive use. That is, the solution time should be within few seconds. To allow an experimental evaluation, time-dependent road networks are found by applying simulated travel time functions to road segments.

In this chapter further motivation for the problem and an overview of the thesis is given.

1.1 Motivation

Vehicle traffic has increased overall the world and congestion is becoming an increasing problem for both commercial and private transportation. It is expected that vehicle transport will continue to grow at a pace faster than road infrastructure can be extended [34], so route planning needs to be more sophisticated than just finding statically shortest paths.

Congestion is not only a problem for commuters experiencing traffic jams during rush hours, but also a problem for commercial transportation. Consider a trucking company planning a trip from Denmark to the Netherlands. A truck will have to pass several bigger cities on the way, with the risk of getting caught in traffic jams if passing the cities at certain times. Being able to find shortest paths that takes these time-dependent phenomena into account by avoiding roads with congestion, would save fuel expenses and allow more reliable delivery times.

When departure time is fixed we have an *earliest arrival time* problem. Another problem where arrival time is fixed is called the *earliest departure time* problem. This problem is also interesting, for example for a trucking company having to deliver at a certain time. We explore ways to solve this problem exact, but also give a simple heuristic using earliest arrival time to solve the problem. There are also problems where travel time is to be reduced, and departure, or arrival time, can freely be chosen within a period of time. We also propose a heuristic to solve this problem.

1.2 Thesis Overview

The thesis is organized in three parts and nine chapters as presented below.

Part I - Theory

Chapter 2 A general introduction to shortest-paths problem variants, together with a description of current best solution algorithms to the static shortest-paths problem for road networks. This should lay a foundation for a later discussion of if and how algorithms can be adapted to shortest-paths problems in time-dependent road networks.

Chapter 3 A thorough study of the time-dependent shortest-paths problem is given. Notation is defined, and complexity and properties analyzed. The problem is generally $\mathcal{NP}$-hard, but a FIFO property is defined that allows polynomial solving. A first simple algorithm is given.

Part II - Speed-Up Algorithms

Chapter 4 It is discussed if and how speed-up techniques for the static shortest-paths problem can be adapted to the time-dependent shortest-paths problem. It is described how A^* *search* and *landmarks* can be used to speed-up the time-dependent problem.

Chapter 5 A detailed description of adapting the speed-up technique *precomputed cluster distances* to the time-dependent problem is given.

Part III - Practical

Chapter 6 To do an experimental evaluation of the developed algorithms, time-dependent road networks are needed. It is discussed how real forecasts could be obtained, and a method to find simulated forecasts with some touch of realism is given.

Chapter 7 A description of the implementation of data structures, algorithms and applications.

Chapter 8 Experiments and results are presented.

Chapter 9 The conclusions of the thesis.

Part I

Theory

Chapter 2

Shortest-Paths

Finding shortest paths in graphs is a classical problem in algorithmics, and a showpiece of real-world application. In this chapter we give a general introduction to shortest-paths problems in the context of road networks. A thorough description of the current best solution methods for the point-to-point problem is given. This should lay the foundation for a later discussion of which algorithms may be applied directly, or modified, to the problem of finding shortest paths in time-dependent networks.

2.1 General Problem Description

In a *shortest-paths problem* (SPP) we are given a weighted directed graph $G = (V, E)$, with function $w : E \to \mathbb{R}$ mapping edges to some real-valued weight. Given a path $p = v_0 v_1 \dots v_k$ in G the corresponding *path weight* $w(p)$ is the sum of the weights of its constituent edges.

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) \tag{2.1}$$

The *shortest-path weight* from $u \in V$ to $v \in V$ is defined by

$$\delta(u, v) = \begin{cases} \min\{w(p) \mid p \in P(u, v)\} & P(u, v) \neq \emptyset \\ \infty & P(u, v) = \emptyset \end{cases} \tag{2.2}$$

where $P(u, v) = \{p \mid u \overset{p}{\rightsquigarrow} v\}$ is the set of paths between u and v. A *shortest path* from node u to node v is defined as any path $p \in P(u, v)$ with minimum weight $w(p) = \delta(u, v)$. For shortest paths the following optimality condition (triangle inequality) clearly holds.

$$\delta(s, v) \leq \delta(s, u) + w(u, v) \quad \forall (u, v) \in E \tag{2.3}$$

Figure 2.1 shows an example of a shortest path in a small graph.

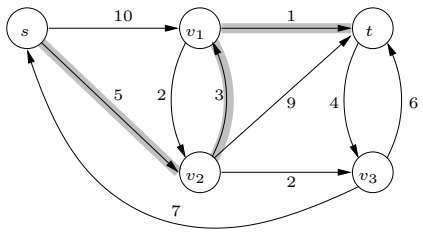


Figure 2.1: Shortest path in a small graph.

What differentiates shortest-paths problems is the objective and the definition of the weight function. Below different problem variants are listed.

- *Single-pair shortest-paths problem* or *point-to-point shortest-paths problem* – Find a shortest path between a given pair of nodes $s \in V$ and $t \in V$.
- *Single-source shortest-paths problem* – Find a shortest path from a given source $s \in V$ to each node $v \in V$.
- *Single-destination shortest-paths problem* – Find a shortest path to a given destination node $t \in V$ from each node $v \in V$. This problem can be reduced to a single-source problem by reversing the direction of each edge.
- *All-pairs shortest-paths problem* – Find a shortest path from u to v for every pair of nodes $u \in V$ and $v \in V$.

How to solve a shortest-paths problem is dependent on the nature of the weight function. If the weight function never changes we have a *static* shortest-paths problem (SSPP), where edge weights are static for multiple queries. If the weight function changes in an instantaneous and unpredictable way we have a *dynamic* shortest-paths problem (DSPP). Edge weights are static for a single query, but can change between queries. If weights are a predictable function of time we have a *time-dependent* shortest-paths problem (TDSPP). That is, the weight function becomes $w : E \times T \to \mathbb{R}$ where T is some time domain, continuous or discrete. A shortest path is found with regard to the weight changes that happens as one travels through the network. If the time-dependent weight function changes often in a unpredictable way we have a *dynamic time-dependent* shortest-paths problem (DTDSPP). For each query the weight function is fixed, but can change between queries. Notice that the time-dependent problem is a generalization of the static problem, as edge weights in this problem are constant functions.

The complexity of solving SSPP and DSPP depends on the weight function. If a graph contains a negative weight cycle, a shortest path is clearly not well defined. Disallowing the use of a negative cycle more than a fixed number of times makes the problem $\mathcal{NP}$-hard. Most solution algorithms further require edge weights to be non-negative. The complexity of TDSPP and DTDSPP will be described in Chapter 3.

There is a difference between finding the shortest-path weight and finding an actual corresponding shortest path. Most algorithms focus on finding shortest-path weights. In the algorithms of this thesis it will be described how to find the actual shortest path only when this is not obvious.

In the next sections and the following chapter, point-to-point shortest-paths problems, with different weight function variants will be described more thoroughly, together with an overview of the most efficient solution algorithms.

2.2 Canonical Shortest Paths

When working with shortest paths it is often convenient to ensure uniqueness even though shortest paths are not necessarily unique. A way to ensure this is to employ a deterministic rule that decides which of a set of paths to choose. We use a *canonical ordering* of V which is an injective mapping $o : V \to \mathbb{N}$. Let p be a shortest path from s to t, and let $q = v \dots t$ be the maximal subpath ending at t that p and any alternative shortest path $\tilde{p}$ have in common. Finally let u be the predecessor of v and p and $\tilde{u}$ the predecessor of v on $\tilde{p}$, then shortest path p is a *canonical shortest path* if $o(u) < o(\tilde{u})$. Figure 2.2 shows the situation.

Canonical shortest paths have the following properties: the canonical shortest path is unique; for a connected graph and pair of nodes s,t existence is guaranteed; subpaths of canonical paths are also canonical paths. The proofs are straightforward. Shortest-path algorithms can be modified so they find canonical shortest paths. We will describe the change when this is not obvious.

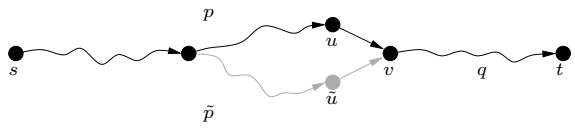


Figure 2.2: Canonical shortest path p because $o(u) < o(\tilde{u})$ for any alternative shortest path $\tilde{p}$.

2.3 Shortest-Paths in Road Networks

In this thesis we are working with road networks and use a graph representation where nodes are intersections and edges are road segments between adjacent intersections. The edges are directed so a one-way street is represented by a single edge in the driving direction, two-way streets are represented by two opposite directed edges. We will use the term road network meaning the graph representation. Figure 2.3 shows a graph representation of the road network on the Danish island Bornholm.

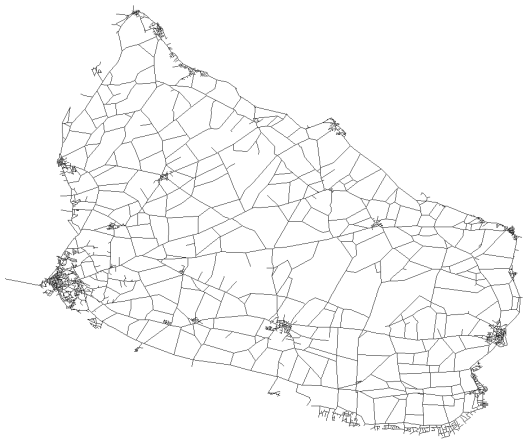


Figure 2.3: Graph representation of the road network on the Danish island Bornholm.

When working with road networks we know the following about the number of edges in relation to nodes $m = \Theta(n)$ where $m = |E|$ and $n = |V|$. The graph is sparse as the number of roads per intersection is a small constant on average and only for large roundabouts is it more than 4. This is an important fact when evaluating the complexity of solution algorithms. We will assume that road networks are *strongly connected*. That is $P(u, v) \neq \emptyset$ for all pairs of nodes $u \in V$ and $v \in V$. This is fair to assume as ferries are also included in the road network connecting islands to the rest of the network.

The subject of this thesis is finding shortest paths in the context of vehicular transportation, so we will solely be focusing on solving the point-to-point shortest-paths problem. When planning a trip, a shortest path can be in terms of distance, time or some other cost measure. In this thesis we will solely be focusing on fastest paths. That is, the weight function gives the non-negative estimated time it takes to traverse a road segment. In some cases it is relevant to find the shortest distance path. For example for a trucking company wanting to prioritize equipment wear over delivery time. Solving the shortest distance path problem in road networks is simpler than solving the fastest path problem, because the problem has fewer dynamic characteristics. The length of a road segment very rarely changes, but the time it takes to traverse it does.

2.4 Static Shortest-Paths

Solution algorithms for the static shortest-paths problem have undergone a rapid development in recent years, leading to methods that are up to one million times faster than the classical algorithm by Dijkstra [14]. Most of these algorithms are tuned specifically towards shortest paths in road networks. In this section we will give an overview of the most notable speed-up techniques.

2.4.1 Dijkstra's Algorithm

The classical algorithm for solving shortest-paths problems is the label-setting algorithm by Dijkstra given as Algorithm 2.1.

Algorithm 2.1 (STATIC-DJK) Dijkstra's Algorithm

```
DIJKSTRA(G, s, t)
 1  d(v) ← ∞   ∀v ∈ V
 2  l(v) ← UNDISCOVERED   ∀v ∈ V
 3  p(v) ← v   ∀v ∈ V
 4  d(s) ← 0
 5  l(s) ← DISCOVERED
 6  Q ← V
 7  while Q ≠ ∅
 8        do u ← argmin{d(u) | u ∈ Q}
 9           l(u) ← SETTLED
10           if u = t
11              then return
12           Q ← Q\{u}
13           for each v ∈ adjacent(u)
14                 do l(v) ← DISCOVERED
15                    if d(u) + w(u, v) < d(v)
16                       then p(v) ← u
17                            d(v) ← d(u) + w(u, v)
```

The algorithm maintains tentative distances $d(u) \geq \delta(s,u)$ for each node $u \in V$. The algorithm maintains the invariant that when a node is *settled* in line 8, the tentative distance equals the shortest-path distance $d(u) = \delta(s,u)$. When a node u is settled, its outgoing edges are *relaxed* in line 17. The algorithm terminates when the target node is settled if used in a point-to-point context. Otherwise the algorithm grows a shortest-path tree rooted in s and finds shortest-path distances to all other nodes. Using a binary heap, the running time of the algorithm is $O(n \log n)$ when used on a road network. Notice that because $m = \Theta(n)$ there is no asymptotic improvement by using a fibonacci heap, as is the case for denser graphs, because $O(n \log n + m) = O(n \log n)$. The canonical shortest path can be found by changing the relaxation condition in line 15 to

$$(d(u) + w(u,v) < d(v)) \ \vee \ (d(u) + w(u,v) = d(v) \ \wedge \ o(u) < o(p(v)))$$

Given a source node s we define the *Dijkstra rank* $\text{rank}_s(u)$ of each node u as the order in which it is settled. Notice that we do not have to maintain all nodes in Q from the start, but only have to insert a node v when the node is discovered for the first time in line 13. In Chapter 7 we give a detailed description of an actual effective implementation of Dijkstra's algorithm. The actual shortest path can be reconstructed using the predecessor $p(v)$ of v in the shortest-path tree.

The size of the search space is $O(n)$. Search time and space can be reduced using a *bidirectional* search where Dijkstra's algorithm is used in both the forward direction from source s and in the backward direction from target t. When the search frontiers meet, the shortest path can be constructed from the two search spaces.

The efficiency of bidirectional search depends on how well the search directions are balanced. Different strategies can be used, the simplest being alternating between the search directions, another is to always choose the direction with the minimum tentative distance settled so far. In a road network, where search spaces will take roughly circular shape, we can expect a speed-up around two if search spaces are well balanced. One disk with radius $\delta(s,t)$ have twice the area of two disks with radius roughly $\delta(s,t)/2$. Bidirectional search is a speed-up ingredient used in many shortest-paths algorithms. Figure 2.4 shows the principle in bidirectional search.

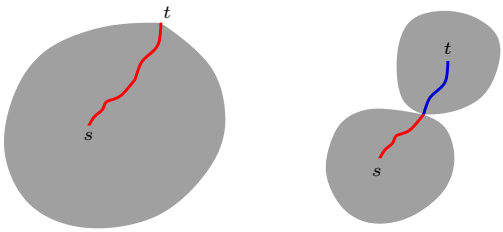


Figure 2.4: Unidirectional search from source s and bidirectional search from source s and target t.

2.4.2 Preprocessing

For large road networks, like the whole of Germany or the whole of Europe, Dijkstra's algorithm is simply too slow. The static nature of edge weights allow a preprocessing stage, so subsequent queries will be faster. The preprocessing stage either generates a search data structure or some kind of information that allows a speed-up of Dijkstra's algorithm by pruning parts of the search space. Preprocessing is the key to the dramatic speed-up in shortest-path queries in recent years. The union of a preprocessing algorithm and the subsequent algorithm for answering the queries is often called an *oracle*.

Oracles are a balance between time and space used in the preprocessing stage, and the subsequent query times. As an extreme example we could find all-pairs shortest-paths with the algorithm by Floyd-Warshall [10, chap. 25] or the algorithm by Johnson [10, chap. 25] in time $O(n^3)$ and $O(n^2 \log n)$ respectively, and using space $O(n^2)$. The query time would be $O(1)$ but the time and space of the preprocessing stage only allows the procedure for very small networks. In the context of large road networks, with millions of nodes and edges, the time and space of the preprocessing stage is a major concern. Generally we say that when we work with large graphs we cannot afford more than $O(n)$ memory consumption for precomputed data.

2.4.3 Highway Hierarchies

Currently most commercial applications compute paths heuristically, that is, they do not always produce the shortest. The heuristics build on the intuition that shortest paths often use small roads only locally at the beginning and the end of a path. That is, the heuristic performs some local search from source and target, and then switch to some highway network that is much smaller than the complete network. This requires labeling of edges to categorize them, and a trade-off between computation speed and suboptimality of computed paths.

The *highway hierarchies* method by Sanders and Schultes [35] introduces an idea to automatically compute a hierarchy that yields optimal routes very quickly, without any pre-labeling of edges. The basic idea is to define a neighborhood $\mathcal{N}_H(u)$ for each node u that consist of the H closest neighbors of u. An edge (u,v) is a highway edge if there is some shortest path $s \dots uv \dots t$ such that neither u is in the neighborhood of t nor v in the neighborhood of s. Figure 2.5 shows the principle.

The above defines the first level of a highway hierarchy. A highway hierarchy of L levels of a graph $G = (V,E)$ is a sequence of graphs $G = G^0, \dots, G^L$ with node sets $V = V^0 \supseteq V^1 \supseteq \cdots \supseteq V^L$

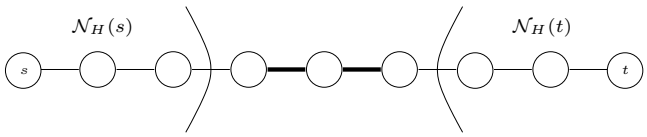


Figure 2.5: Shortest path from s to t with indication of highway edges.

and edge sets $E = E^0 \supseteq E^1 \supseteq \cdots \supseteq E^L$. Each edge has maximum hierarchy level, the maximum i such that it belongs to E^i, such that for all pairs of nodes there exists between them a shortest path $e_1 \ldots e_k$, where e_i are the consecutive path edges, whose search level first increase and then decreases, and each edge's search level is not greater than its maximum hierarchy level. Figure 2.6 shows a small example network, and the level $L = 1$ highway network for $H = 3$.

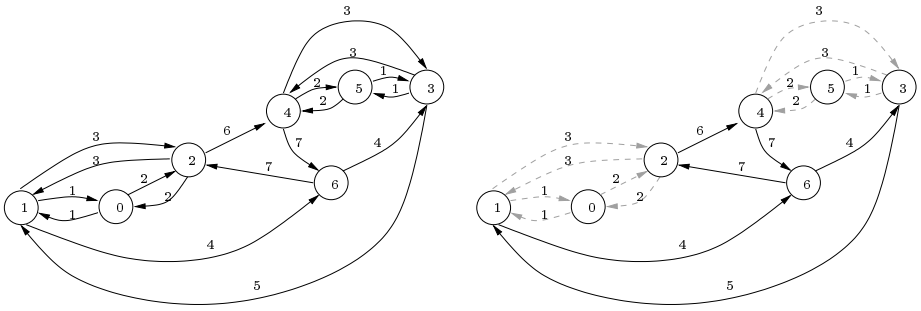


Figure 2.6: A graph (left) and the corresponding highway network (right) for $H = 3$, $L = 1$. Solid lines indicate highway edges.

The levels of the highway hierarchy are found by an iterative process where the highway network is found, followed by a contraction phase. The contraction phase is the following pruning operations: isolated nodes can be removed; trees attached to a biconnected component can only be traversed at the beginning and end of a path; paths consisting of nodes with degree two can be replaced by a single edge. The contracted network, only containing nodes of degree at least three, is significantly smaller than the original network. The iterative process creates highway networks of highway networks, each containing less and less nodes.

The space complexity of the highway hierarchy is linear, which is very important, as this enables the use on very large graphs. For the road network of Northern America, the largest available at the moment, consisting of about 24 million nodes and 58 million edges, the reported construction time is 4 hours and 15 minutes in the original article. Recently Sanders and Schultes [36] gave an extensive revision of highway hierarchies that reduces preprocessing time to only 15 minutes for the North American network.

Quering for shortest paths in the highway hierarchy is done by a multi-level bidirectional Dijkstra algorithm. A reported speed-up of more than 2000 compared to the normal Dijkstra algorithm is reported. This is due to the fact that the search scope occur mostly on a high hierarchy level, with fewer nodes and edges than in the original graph. Quering in the largest networks is done in around 10 milliseconds, which is more than enough to allow interactive use. In the revised version the quering is just around 1 millisecond.

Notice that special handling is needed to get an actual shortest path because edge pruning is performed, replacing multiple edges with a single edge. This can be achieved by saving enough information to enable reconstruction of original edges. It is unclear how much effort this will require.

2.4.4 Goal Direction

Goal directed search is a technique where the ball of nodes settled by Dijkstra's algorithm is stretched in the direction of the target. An improvement in running time is achieved because the number of nodes visited is reduced. Figure 2.7 shows examples of a normal unidirectional Dijkstra search and a goal directed search.

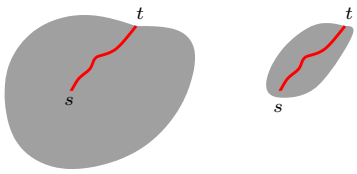


Figure 2.7: A normal Dijkstra search (left) and a goal directed search (right).

A* search

A* search is a technique to direct the Dijkstra search by using *node potentials* $\pi : V \to \mathbb{R}$. The search is a normal Dijkstra search performed on the original graph but with a different weight function $w_\pi(u, v)$ defined as the reduced weight $w_\pi(u, v) = w(u, v) - \pi(u) + \pi(v)$. Notice that the underlying graph does not have to be changed, reduced weight can be found on the flight.

Consider any path $p = v_1 \dots v_k$ between any pair of nodes v_1 and v_k. Because the potentials telescope the reduced weight of a path will be $w_\pi(p) = w(p) - \pi(v_1) + \pi(v_k)$. This ensures that a shortest path using the reduced weights is also a shortest path using the original weights.

Dijkstra's algorithm only works on graphs free of negative cycles. To ensure this we demand that all edges have non-negative weight. A potential function obeying this is called *feasible*. More formally a potential function π is feasible if $w_\pi(u, v) \geq 0$ for all $(u, v) \in E$.

In the context of finding a shortest path from s to t there is a potential function $\pi_t(v)$ giving an estimate on the distance from v to t. If π_t is a feasible potential function and $\pi_t(t) = 0$, we talk of *lower bounding algorithms* where $\pi_t(v)$ is a lower bound on the distance to t. Tighter lower bounds will push the search more into the direction of the target t, because the edges that leads towards the target have been shortened. Notice in this case using reduced weights is the same as selecting the node with the smallest value of $k(u) = d(u) + \pi_t(u)$ to be settled in each iteration of Dijkstra's algorithm.

If we have a set of feasible potential functions $\pi_1, \dots, \pi_k$, a tighter feasible potential function π can be found as

$$\pi(v) = \max\{\pi_1(v), \dots, \pi_k(v)\} \tag{2.4}$$

The question now is how to obtain good lower bounds.

Geometric Lower Bounds

When edge weights are lengths and there are geographic coordinates on nodes available, it is possible to use Euclidean distance $\|u - v\|$ as lower bound on distance $\delta(u, v)$. More advanced approaches using geometric containers exist. We will not describe geometric lower bounds further as they are not very suitable when working with time rather than length weights. If they where to be used the maximum speed of any edge could be used to find the minimum time it could ever take to travel an edge. This is a very conservative estimation, the speed-up for finding fastest paths is rather small. Goldberg et al. [18] even reports a slow-down of more than a factor of two, because the search space is not significantly reduced but a considerable overhead is added.

Landmarks

The landmark technique is a part of the ALT algorithm introduced by Goldberg et al. [18]. ALT is a combination of **A*** search, **L**andmarks and the **T**riangle inequality. In the landmark technique a very small number of nodes are selected as landmarks $L = \{l_1, \ldots, l_k\} \subset V$. For each landmark $l \in L$ the shortest paths distances $\delta(v, l)$ and $\delta(l, v)$ are found for each node $v \in V$. Notice that the last distances can be found by growing a shortest path tree rooted at each $l \in L$. The first distances can be found as a single-destination shortest-paths problem, which, as noted earlier, can be reduced to a single-source shortest-paths problem by reversing the direction of each edge. Hence we need to grow an additional shortest-path tree rooted in each $l \in L$ using reversed edges.

Shortest-paths distances to landmarks can be used in triangle inequalities as indicated in Figure 2.8. The same is the case for shortest-paths distances from landmarks. The triangle inequalities yields the following lower bounds for distances to landmarks and from landmarks, respectively.

$$\delta(v, l) - \delta(t, l) \leq \delta(v, t) \qquad \delta(l, t) - \delta(l, v) \leq \delta(v, t) \tag{2.5}$$

The tightest lower bound can be used to define node potentials $\pi_t(v)$ for each target t and node $v \in V$.

$$\pi_t(v) = \max \left\{ \max_{l \in L} \{\delta(v, l) - \delta(t, l)\}, \max_{l \in L} \{\delta(l, t) - \delta(l, v)\} \right\} \tag{2.6}$$

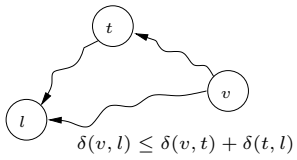


Figure 2.8: Lower bound using landmarks and the triangle inequality.

In [18] Goldberg et al. use the following optimization. For a given s and t a fixed subset $L' \subseteq L$ of landmarks, that give the highest lower bounds on the s-t distance are selected. During the s-t shortest path computation only landmarks from L' are used when computing lower bounds. Goldberg et al. gives in [19] an alternative optimization where the set of active landmarks L' is updated dynamically.

Finding good landmarks is critical for the overall performance of lower-bounding algorithms. Having to pick k landmarks a couple of different strategies are given below.

- The simplest way is to select k landmarks randomly.
- If we have two dimensional layout of nodes, we can use an approach where a center node c is selected and the rest of the graph divided using a pie with center c and k sectors having approximately the same number of nodes. For each sector the node farthest away from c is selected as landmark. The method derives from the observation that having landmarks geometrically lying behind the destination tends to give good potential functions.
- A greedy method known as *farthest landmark* works as follows. A start node is selected arbitrarily and a node v_1 farthest away from it is found. Add v_1 to the set of landmarks. Proceed in iterations, at each iteration find a node that is farthest away from the current set of landmarks and add it to the set. This algorithm can be seen as a quick approximation to the k-center problem, where the distance between nodes is the shortest path distance. The method can be improved by appending a local search after k landmarks have been found, where a landmark is replaced with the node farthest from the remaining set of landmarks. In [19] Goldberg et al. notice, that slightly better results are obtained with the farthest landmark method, if farthest is defined in terms of number of hops instead of distance. This also allows using a simple breadth-first search to find farthest landmarks.

- Goldberg et al. [19] introduces a new method called *avoid*, the method works as follows. Assume there is a set L of landmarks already selected and that we want to add an additional landmark. A shortest-path tree T_r is found rooted at some node r. For every node v a *weight* is found as the difference between $\delta(r, v)$ and the lower bound $\pi_v(r)$, based on the current landmarks L. This is a measure of how bad the current distance estimates are. For every node v now compute the *size* $s(v)$ which depends on T_v the subtree of T_r rooted at v. If T_v contains a landmark then $s(v) = 0$; otherwise $s(v)$ is the sum of the weights of all nodes in T_v. Let u be the node of maximum size. Traverse T_u, starting from u and always follow the child with the largest size until a leaf is reached. Make this leaf the new landmark. The method tries to identify regions of the graph that are not well-covered. No path from u to a node in its subtree has a landmark "behind it". By adding a leaf of this tree to the set of landmarks, *avoid* tries to improve the coverage.
- In [19] Goldberg et al. give a method called *max-cover* to improve on solutions found by avoid. They employ a local search, where in each iteration an instance of maximum cover has to be solved, for which they use a heuristic.

In [19] Goldberg et al. give an experimental evaluation of the methods to find landmarks. They find that the max-cover method gives the best results, followed by avoid which was a factor 0.9 worse. The farthest landmark method was 0.84 and 0.8 worse, using number of hops, and shortest-path distance, respectively.

In the ALT implementation given in [18] up to 16 landmarks are used. For each node 16×2 distances needs to be saved giving an acceptable memory consumption. The quering achieves speed-ups around 28 compared to unidirectional Dijkstra for different North American road networks. Notice that bidirectional search is used which also contributes to the speed-up.

2.4.5 Precomputed Cluster Distances

Maue et al. [30] give a different way to use precomputed distances for goal directed search. In their method (PCD) the graph is partitioned into k disjoint clusters $V = V_1 \cup \cdots \cup V_k$. For each pair of clusters (V_i, V_j) the shortest-path distance is precomputed.

$$\delta(V_i, V_j) = \min \{\delta(s, t) \mid s \in V_i,\ t \in V_j\} \tag{2.7}$$

The distance, and starting and end node, of the the shortest path is saved for each pair of clusters in a distance table where lookups can be made in constant time. These distances and pairs of nodes can quite easily be found because growing a single shortest-path tree for each cluster suffices. For each cluster S a new node s' is added and connected to all border nodes with zero weight edges. After a shortest-path tree has been grown from s', the shortest path to the other clusters can be found as

$$\delta(S, V_i) = \min \{\delta(s', v) \mid v \in V_i\} \tag{2.8}$$

Figure 2.9 shows the principle. The running time of this preprocessing stage is $(kn \log n)$ because growing the shortest-path trees dominates. Finding border nodes takes $O(n)$ using aggregate analysis, and finding shortest distance nodes can be done by scanning through all nodes in each iteration, contributing $O(kn)$ to the running time.

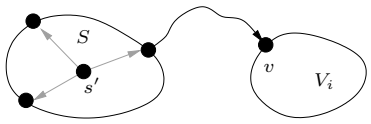


Figure 2.9: Finding shortest path from cluster S to cluster V_i.

The precomputed cluster distances were originally intended for use as bounds in an A* search, but it turned out that they where not usable as they did not always result in a feasible potential

function. Instead they were used on their own to provide upper and lower bounds for $\delta(s,t)$ at query time. This can be used to prune the search. The search can either be uni- or bidirectional. Here we describe the bidirectional version. The search is in two phases. Let S and T be the clusters of s and t respectively.

1. An ordinary bidirectional search is performed from s and t until the search frontiers meet, or until $\delta(s,s')$ and $\delta(s,t')$ are known, where s' is the first border node of S settled the forward search, and t' the first border node of T settled in the backward search.

2. The forward search grows a shortest-path tree, additionally maintaining an upper bound $\overline{\delta}(s,t)$ for $\delta(s,t)$, and computing lower bounds $\underline{\delta}(s,w,t)$ for the distance of any path from s via w to t. The edges out of w are pruned if $\underline{\delta}(s,w,t) > \overline{\delta}(s,t)$ as the upper and lower bound guarantees that a shortest path from s to t does not go through w. Phase 2 ends when the search frontiers meet. The backward search is completely analogously.

The upper bound is updated when a node $u \in U$ is settled and u is the starting node of the shortest path from U to T. Let t_{UT} be the end node for the shortest path from U to T, then the following bound holds.

$$\delta(s,t) \leq \delta(s,u) + \delta(u,t_{UT}) + \delta(t_{UT},t) \tag{2.9}$$

Notice that $\delta(u,t_{UT}) = \delta(U,T)$ which is precomputed and $\delta(s,u)$ is known as u has just been settled. The last part $\delta(t_{UT},t)$ is known if t_{UT} has been settled by the backward search, otherwise an upper bound on the diameter of T can be used. At any time $\overline{\delta}(s,t)$ is the smallest of the bounds from equation (2.9) encountered in the forward or backward search. For any node $w \in W \neq T$ the lower bound is

$$\underline{\delta}(s,w,t) = \delta(s,w) + \delta(W,T) + \min_{t' \in \text{border}(T)} \delta(t',t) \tag{2.10}$$

where $\text{border}(T) = \{t' \in T \,:\, \exists \overline{t} \notin T \,:\, (\overline{t},t') \in E\}$. The lower bound can be found, as w is settled by the forward search, $\delta(W,T)$ is precomputed and the shortest-path distance to any border node in T has been determined in phase 1. Figure 2.10 illustrates the constituents of upper and lower bound.

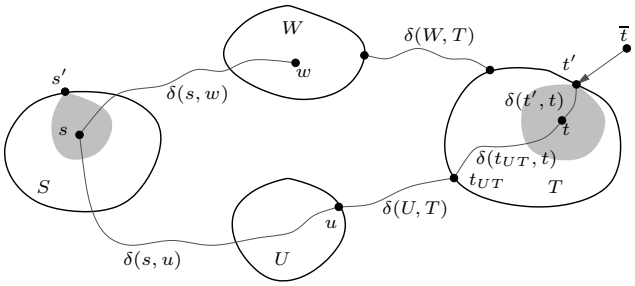


Figure 2.10: Constituents for finding upper and lower bounds for $\delta(s,t)$.

The algorithm can be implemented using space $O(k^2+n)$ where the n part is due to storing cluster identifiers for nodes. This can be reduced to $O(k^2+B)$ where B is the number of border nodes, because it is only necessary to store cluster identifiers for border nodes. They can be saved in a hash table.

Maue et al. [30] propose different clustering methods. Their best results are achieved with a k-center clustering heuristic. An advantage of using the k-center heuristic is that it finds a bound on the diameter to be used for the upper bound in equation (2.9).

The speed-ups achieved with PCD are reported to scale with $\sqrt{k}$. The additional space used is very low, and it is presumably the first sublinear space technique that gives speed-ups $\gg 2$. This offers a very flexible trade-off between space consumption and query time. The preprocessing stage is acceptable as long as $k \ll n$.

Other speed-up techniques based on partitioning has been proposed. Möhring et al. [31] give an experimental study. The concept of edge-labels is a technique where edges are labeled if they are part of a shortest path to a specific part of the network. If for example the network is partitioned into k disjoint clusters, then each edge would have a bit vector of k bits. If an edge is part of a shortest path to cluster i, then bit i will be set for the edge. In a shortest-path search, edges can be pruned if the destination cluster bit is not set.

However the preprocessing time required by these other methods also depends on the number of border nodes B, and they all need more space. Furthermore some of them rely on the geometrical layout of nodes and are most efficient when edge weight are lengths.

2.4.6 Reach-Based Routing

Gutman [21] introduces the notion of *node reach*. Given a path p from s to t and node v on p, the reach $r_p(v)$ of v with respect to p is

$$r_p(v) = \min\{w(p_{sv}), w(p_{vt})\} \tag{2.11}$$

That is, the minimum of the prefix of p and suffix of p defined by v. The reach $r(v)$ of v, is the maximum, over all shortest paths P through v, of the reach of v with respect to $p \in P$.

$$r(v) = \max_{p \in P}\{r_p(v)\} \tag{2.12}$$

That is, the reach of a node is high, if it lies in the middle of a long shortest path. The reach of an edge is defined similarly. Given a path p from s to t and edge (v, w) on p, the reach of (v, w) with respect to p is found by using path p_{sw} as prefix and p_{vt} as suffix. Figure 2.11 shows node and edge reach with respect to a path p.

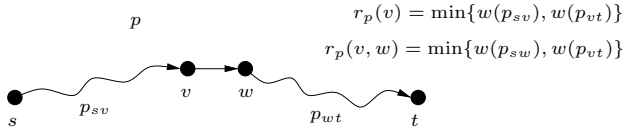


Figure 2.11: Node and edge reach with respect to path p.

The intuition behind using reach for pruning, is to exclude nodes or edges from consideration, if they do not contribute to any path long enough to be of use for the current query. Instead of using exact reach, which is too demanding to determine, upper bounds $\overline{r}(v)$ and $\overline{r}(v, w)$ on reach are used. There is a choice between using node or edge based reaches. Edge reaches are more powerful than node reaches, but as there are more edges than nodes in a road network, they also require more space. From know on we will limit the description of reach-based routing to node reaches. Using edge reaches instead is straightforward.

Pruning in quering is done by lower and upper bounding where reach delivers the upper bounds. Gutman [21] used geometric lower bounds, while Goldberg et al. [17] introduce implicit lower bounds available in a bidirectional search. Assume we have lover bounds $\underline{\delta}(s, v)$ and $\underline{\delta}(v, t)$, then we can prune v if the following holds.

$$\overline{r}(v) < \underline{\delta}(s, v) \text{ and } \overline{r}(v) < \underline{\delta}(v, t) \tag{2.13}$$

In the backward search a similar pruning can be done. In bidirectional search the opposite search can deliver the lower bounds. A self-bounding version also exists, where the two search directions are more independent of each other. In a forward search a node v is pruned if the following holds.

$$\overline{r}(v) < d(v) \tag{2.14}$$

Here $d(v)$ is the tentative distance from s to v in the forward search. The pruning is legal because the node will be handled from the opposite direction. A special stopping criteria is needed for the two searches.

It remains to be discussed how upper bounds on reach can be determined. Exact reach could be found using the following method. Initialize $r(v) = 0$ for all $v \in V$. For each node $x \in V$ grow a shortest-path tree T_x rooted at x. For every vertex $v \in V$ determine its reach $r_x(v)$ within the tree, given by the minimum of its distance to the root, and its distance to the farthest descendant. If $r_x(v) > r(v)$, update $r(v)$. The method requires growing n shortest-path trees, resuling in $O(n^2 \log n)$ running time, which is unacceptable for large graphs. If we only grew trees for a subset of nodes we would end up with lower bounds on reach. Finding upper bounds are much more complex. Bauer [4] gives a simplified version of a method given by Goldberg et al. [17]. The method finds reach bounds by iteratively growing bigger and bigger partial trees from all nodes in a graph that becomes smaller and smaller. The method is quite complex and requires too much space to be described here.

The preprocessing stage where upper bounds on reach are found requires a substantial amount of time. Goldberg et al. [17] report a preprocessing time of 27 hours for the North American road network using their method to find upper bounds. Quering is very effective with a speed-up around 1500 compared to normal Dijkstra search.

Interestingly Goldberg et al. [17] suggest an interpretation of highway hierarchies in terms of reach. Reach is aimed at pruning the shortest-path search, while highway hierarchies takes advantage of inherent road hierarchy to restrict the search to a small subgraph. It turns out that nodes pruned by reach-based routing have low reach values and as a result belong to a low level of a highway hierarchy.

2.4.7 Transit Node Routing

Bast et al. [25] give a new approach based on the concept of *transit nodes*. Their method has astonishing quering times, down to only 10 microseconds for the largest available networks. This is an improvement of two orders of magnitude over other methods.

Like highway hierarchies, the method exploits an inherent hierarchy in road networks, where more and more important roads are used the longer one is from the source and target. The basic intuition behind the method is that a small number of nodes are always used in *non-local* shortest paths, where non-local means that the shortest path covers a certain, not too small, euclidean distance. It turns out that for road networks only ten such transit nodes are needed per node, to reduce routing in large networks to 100 table lookups.

For a given network the method compute a small set of transit nodes, with the property that every shortest path that covers a certain euclidean distance, passes through at least one of these nodes. For every node in the graph the set of *closest transit nodes* are found defined by the property that every shortest path out of the node will pass the transit node. For each node the distance to all closest transit nodes are computed, and the distance between all pairs of transit nodes are computed.

A non-local s, t query can now be answered by fetching the sets of closest transit nodes T_s and T_t respectively. For each pair of transit nodes $u \in T_s$ and $v \in T_t$ the length of shortest path passing through these nodes, which is $\delta(s, u) + \delta(u, v) + \delta(v, t)$, is computed. Notice that all three distances have been precomputed. The minimum is the length of the shortest path.

For local queries, which do not pass through transit nodes, the method on its own cannot offer any speed-up. For these queries other methods needs to be used. Either a simple Dijkstra search, as it is quite fast for a close pair of nodes, or another speed-up technique like highway hierarchies.

The method is extremely fast for answering non-local distance-only queries, where the actual shortest path is not needed. If it is needed, it has to be reconstructed by traversing the shortest path, and in each node make a shortest-path query to determine which adjacent node to choose next.

The memory usage of the methods is reported to around 21 bytes per node, which is acceptable. Preprocessing time is reported to be 15 hours for the North American road network.

2.4.8 Separators

A property of road networks is that they are almost planar. Roads mainly cross at intersections, the only exception is bridges and tunnels. Techniques developed for planar graphs will often also work for road networks. The idea is to partition the graph into small components by removing a hopefully small set of separator nodes. These separator nodes together with edges representing precomputed paths between them constitute the next level of the graph.

Holzer et al. [26] describe the practical approach *separator-based multi-level method*. For a node set $V' \subseteq V$, a *shortest-path overlay graph* $G' = (V', E')$ has the property that E' is a minimal set of edges such that each shortest-path distance $\delta(u, v)$ in G' is equal to the shortest-path distance from u to v in G. In the separator based approach, V' is chosen in such a way that the subgraph induced by $V \backslash V'$ consists of small components of similar size. By recursing on G' a multi-level graph is obtained. Quering is done by searching the multi-level graph. The speed-up achieved is limited to around 10 for road networks.

Shultes and Sanders [39] make a generalization of the separator-based multi-level method to *highway-node routing*, where overlay graphs are defined using arbitrary node sets $V' \subseteq V$ rather than separators.

Thorup [41] suggests an approximation oracle accurate within $(1 + \epsilon)$ in near constant time using $O((n \log n)/\epsilon)$ space and preprocessing time. Muller and Zachariasen [32] do an experimental evaluation on undirected planar graphs. They use randomly generated graphs and graphs derived from real world road networks of up to 1 million nodes. Average query time for the largest graphs is less than 20 microseconds. The preprocessing time varies between 67 minutes and 150 minutes for ϵ between 0.01 and 0.10. The average deviation from optimality is less than 1%. The problem with the oracle is the superlinear space consumption which inhibits practical use on very large road networks.

2.4.9 Combining Methods

Given different speed-up techniques it seems natural to try to combine these methods to achieve an additional speed-up. Here we will mention some of the most successful combinations.

Bidirectional search is an important ingredient in almost all speed-up techniques. In some it is even a necessary ingredient because in one search direction information from the opposite search direction is used. This is the case for highway hierarchies and the reach-based method of Goldberg et al. [17], where implicit lower bounds are deduced from the opposite search direction.

The reach based method of Goldberg et al. [17] combines A* search using landmarks, with reach based routing. They also introduces shortcut edges that dramatically improves performance because they result in much tighter reach bounds. The quering times are comparable to those of highway hierarchies.

Delling et al. [12] try to speed-up highway hierarchies by combining it with A* search using landmarks (ALT) to give it a sense of direction. No significant speed-up is achieved. After the discovery of transit nodes, work was done by Sanders and Schultes [37] to combine highway hierarchies with transit nodes. They make different variants where a trade-off between preprocessing and quering time is possible. With a small decline in query time over original transit node routing, they achieve preprocessing times that are like the ones for highway hierarchies. The work was followed by a joint work in [3] with Bast et al. Trade-off between query time, preprocessing time and storage overhead is made possible and the following figures were reported: Query times are between 5 and 63 microseconds, preprocessing time 59 to 1200 minutes, and storage overhead 21 to 244 bytes per node.

2.5 Dynamic Shortest-Paths

Given the low query times, preprocessing time, and storage complexity of speed-up techniques, the static shortest-paths problem can largely be regarded solved. Query times are actually faster than traversing a corresponding shortest path.

The static problem is the most studied, but in reality many networks, and certainly road networks, tend to have dynamic characteristics. In the dynamic shortest-paths problem, unpredictable changes can happen to the edge set and weight function. For example due to road work or congestion. Using precomputed data to answer distance queries as in the static speed-up techniques poses a challenge. In the dynamic case $update(u, v, w)$, $insert(u, v, w)$ and $delete(u, v)$ operations are needed on the preprocessed data, which are more effective than just reproducing all data. The dynamic shortest-paths problem can be seen as a *reoptimization* problem where a series of closely-related static shortest-paths problems are to be solved.

In this section we give an overview of some of the speed-up techniques used for the dynamic shortest-paths problem. Static speed-up techniques are fairly new, so little research has been done on extending them to the dynamic problem. Progress is rapidly being made, so the section reflects what we to the best of our knowledge know at the time of writing.

2.5.1 Dijkstra's Algorithm

Dijkstra's algorithm, both in the uni- and bidirectional form, rely on no precomputed data, so it is perfectly applicable for the dynamic problem. But as was the case for the static problem, Dijkstra's algorithm is simply too slow for queries in very large networks.

If distance queries always have the same source s (or target), an existing shortest-path tree T_s rooted in s can be updated faster than recomputing the whole tree. Frigioni et al. [16] give an algorithm where the running time depends on the output updates, which is the number of nodes that gets modified shortest-path distance or new parent by an update. Each output update requires $O(\sqrt{m} \log n)$ in the worst case. Inserts and deletes are shown to have the same amortized bound.

2.5.2 Goal Direction

Goal direction for the dynamic problem depends on the method to find lower bounds. If for example euclidean distances are used and the edge weights are proportional to these, then no preprocessing is needed and the lower bounds are fully dynamic. If edge weights are travel time and euclidean distance is used as lower bound, the maximum speed on any edge needs to be updated if an update affects this speed.

Delling and Wagner [13] deal with extending lower bounding by landmarks to the dynamic problem. Assume that a set of landmarks $L = \{l_1, \ldots, l_k\} \subset V$ have been found by some method. From each landmark a shortest-path tree is grown to give distance to all other nodes. If the set of landmarks are kept static, an edge update is now a matter of updating the shortest-path trees containing the edge. As was just described, this can be done rather effectively. Delling and Wagner exploit the fact that for road networks the static travel time weights are most optimistic weights when there is no congestion. Lower bounds found in this network also holds for the network with increased edge weights, and edge weights never drop below the optimistic weights. They use a lazy strategy where bounds are never updated, at the cost of larger search space if many edges have increased weights. Delling and Wagner achieve good results as the search space does not increase considerably for the lazy method under moderate edge updates. Altering low category edges nearly has no impact, while altering motorway edges has a bigger effect. If high-category edge weights are increased by a factor 2, a 15-45% decrease in performance is seen for different road networks.

Notice that keeping the set of landmarks static may decrease performance over time, because the basis at which the landmarks was found has changed. For road networks this is again not a problem, because we assume the network to resume to normal regularly.

To the best of our knowledge, no work has been done on extending precomputed cluster distances to the dynamic problem. There may be a method to effectively update bounds on edge updates. Another strategy would be to base cluster distances on a most pessimistic and a most optimistic network. This requires that there are bounds on the edge weight variation, which may not be the case in practice.

2.5.3 Reach-Based Routing

Bauer [4] gives an algorithm that updates precomputed reach bounds on edge updates, guaranteeing the same bounds that a full recomputation would give. The algorithm builds on the update of shortest-path trees, as the method used to find reach bounds is using partial shortest-path trees. No time complexity and no practical experiments are given. It is mentioned that an update in the worst case is slower than a complete recomputation. The concept of shortcut edges given in [35] and [17], which significantly speeds-up reach based pruning, has not yet been extended to the dynamic problem.

2.5.4 Hierarchy Based Methods

Efforts have been made to extend the highway hierarchy method to the dynamic problem, but without success. Nannicini et al. [33] used the highway hierarchies for a heuristic solution to the dynamic problem. They build the highway hierarchy using static travel times, but make queries using dynamic travel times. That is, static times are used to make decision of when to switch level, and dynamic times are used as tentative distances determining order in the priority queue of the multilevel Dijkstra algorithm. This leads to a heuristic as shortest paths are not guaranteed. The tests performed shows a 18% maximum deviation from optimal. Their test is performed by making random updates to edges, which is far from realistic for road networks, where congestion often happen on certain critical roads, and around certain times of day.

Instead of pursuing a dynamic version of highway hierarchies, Sanders and Schultes [39] developed *dynamic highway-node routing*, which allows efficient update operations and fast query times. Highway node routing is, as earlier mentioned, a generalization of the separator-based multi-level method. Separator nodes are *important* nodes that are used by many shortest paths. They can be compared to transit nodes in the transit node routing method. Important nodes are found by the highway hierarchy method like in [37]. The strategy of Sanders and Schultes is to move the complexity of highway hierarchies to a preprocessing stage, leaving a much simpler multi-level graph for quering. And most importantly, the multi-level graph can be updated quite efficiently on edge changes. Query times of the method is around 1 millisecond for large networks, the same as for highway hierarchies. The memory usage is 32 bytes per node, which is only 4 more than the highway hierarchies method. Finally edge update times vary between only 2 and 40 milliseconds. The dynamic highway method can be regarded as a breakthrough in the research on the dynamic shortest-paths problem.

Chapter 3

Time-Dependent Shortest-Paths

If network characteristics change with time in a predictable fashion, we have a time-dependent shortest-paths problem (TDSPP). In this problem variant the weight, or travel time, of edges is a function of time. There is a big difference between SPP/DSPP and TDSPP because in SPP/DSPP the shortest path is found based on current travel times, while in TDSPP the changes that will happen as one travels through the network are taken into account.

In this chapter a thorough study of the time-dependent shortest-paths problem is given. The basic theoretical properties are largely based on proofs given by Dean [11].

3.1 Problem Description

In a time-dependent shortest-paths problem we need a *travel time function* $w : E \times T \to \mathbb{R}_+$ which given departure time returns the time it takes to travel an edge. We will use the notation $w_{uv}(t)$ for travel time along edge $(u, v) \in E$ at departure time $t \in T$.

Given a travel time function, a time-dependent network can be formulated by an *arrival time function* $a_{uv}(t)$ which gives the the arrival time at v if leaving u at time t along edge (u, v). The travel time and arrival time are simply related in the following way

$$a_{uv}(t) = w_{uv}(t) + t \tag{3.1}$$

Some properties of time-dependent networks are more naturally formulated using the arrival time function. We know the travel time is non-negative so $a_{uv}(t) \geq t$.

Time-dependent shortest-paths problems can be divided into problems that allow waiting at nodes, and problems that do not. Waiting may be appropriate for some problem domains, but for road networks it is not, as waiting at intersections is prohibited. Also in section 3.5 it will be made clear that waiting in a road network can almost never be beneficial.

Given that waiting is not allowed we can define a *path arrival time function* $a_p(t)$ of a path $p = v_0 v_1 \dots v_k$ with departure time t as

$$a_p(t) = a_{v_{k-1}v_k}(a_{v_{k-2}v_{k-1}}(\cdots \ a_{v_0 v_1}(t))) \tag{3.2}$$

We can now define *earliest arrival time* $\delta_{uv}(t)$ from u to v when departing from u at time t as

$$\delta_{uv}(t) = \min \{a_p(t) \mid p \in P(u, v)\} \tag{3.3}$$

Notice how this formulation resembles the formulation of shortest-path weight in the static case given in equation (2.2). An *earliest arrival time path* from node u to node v at time t is defined as any path $p \in P(u, v)$ with earliest arrival time $a_p(t) = \delta_{uv}(t)$. The shortest path travel time $\gamma_{uv}(t)$ is

$$\gamma_{uv}(t) = \delta_{uv}(t) - t \tag{3.4}$$

In the next sections it will be shown that TDSPP is $\mathcal{NP}$-hard in the general case, but under a FIFO property the problem exhibits many nice structural properties that enables the development of a polynomial-time solution algorithm. First a short summary of the history of the problem is given and it is discussed if a time-dependent network is a good model for road networks.

3.2 Literature Review

Cooke and Halsey [7] proposed in 1966 a shortest-paths problem in networks where inter-nodal time requirements are included. Their suggested algorithm is a modified form of Bellman's [10, chap. 24] label-correcting shortest-path algorithm. The algorithm is pseudo-polynomial with complexity $O(n^3 T_{max})$ where T_{max} is the longest travel time of any edge at any time.

Dreyfus [8] claimed in 1969 that TDSPP could be solved by using a modified version of Dijkstra's algorithm. However Kaufman and Smith [27] in 1993 proved that this was wrong in the general case, but true if the network admits a FIFO property.

Orda and Rom [9] consider in 1990 three classes of networks: first, where vehicles may wait an unlimited duration at any node; second, vehicles may wait only at the origin node of the trip; and third, where vehicles are not permitted to wait at any node. In road networks unrestricted waiting at intersections is not allowed so only the last case is interesting in this context. Orda and Rom demonstrated that in this case there may be no finite optimal path because waiting can be achieved by using cycles.

Sung et al. [28] suggested a new model for time-dependent networks where the flow speed of each edge depends on the time interval. They gave a modified version of Dijkstra's algorithm and showed that their model obeyed the FIFO property.

Dean [11] gave in 2004 a concise study of the theoretical properties of TDSPP and its solution algorithms. He determines that the problem is $\mathcal{NP}$-hard in the non-FIFO case. The solution algorithms are all modified versions of Dijkstra's algorithm or label-correcting algorithms.

3.3 Time-Dependent Road Networks

When finding a shortest path in a static road network, an estimated total travel time is found. The estimated travel time may not hold if one makes stops or not cover road segments in the estimated travel time, but the shortest path does not change. For time-dependent road networks, this need not be the case. Travel time on edges is found from the estimated departure time, so if a vehicle does not stay on schedule, the shortest path may change. For long trips it is unlikely that a driver can stay on the exact schedule, with regard to estimated travel times, so this raises the question if a time-dependent network is a good model for road networks.

Notice that for short trips the inaccuracy is negligible, because the travel times are still an estimate, and the amount of time a vehicle drift off schedule is unlikely to change the estimated travel times dramatically. For longer trips reoptimization may be needed if the vehicle drift more away from schedule than a certain threshold. A new time-dependent shortest path is simply found from the current location to the destination with current time as departure time.

For trucks on long trips there may be rules on resting periods after a certain amount of driving. If resting places are chosen along the way, solving the corresponding time-dependent problem gets much more complicated. If resting places were chosen in advance, time-dependent shortest-paths problems between the resting places could just be solved. We will not address this problem further in the thesis.

When finding a time-dependent shortest path, it is done for a single vehicle. Estimated travel times are based on the assumption that all other vehicles behave as normal. If all vehicles used paths found as time-dependent shortest paths, a discrepancy would appear between estimated and observed travel time. If every vehicle tries to avoid congestion be using the same detour, this detour will of course congest. We will ignore this fact, as the only real alternative is solving a global *vehicle routing problem* for all vehicles in a road network.

3.4 Complexity

Dean [11] mentions that TDSPP is $\mathcal{NP}$-hard in the general case. In this section we prove this with a reduction from *the partition problem* which is a special case of *the subset-sum problem* [10, chap. 34]. The proof is based on a similar proof given by Sherali et al. [22] for the related problem: *Time-dependent shortest pair of disjoint paths*. Firstly we define the partition problem and TDSPP as decision problems.

Problem 3.1 (PARTITION) *Given set $S = \{s_1, s_2, \ldots, s_n\} \subset \mathbb{Z}$, where $\sum_{i=1}^{n} s_i = 2r$, does there exist a partition $S' \subseteq S$ of these integers such that $\sum_{s \in S'} s = r$.*

Problem 3.2 (TDSPP-DECISION) *Given a time-dependent network $G = (V, E)$, $k \in \mathbb{R}$, and source and target nodes u and v, determine if there exists a path p with arrival time $a_p(t) \leq k$.*

We now continue to show that the decision version is $\mathcal{NP}$-complete.

Theorem 3.1 *TDSPP-DECISION is $\mathcal{NP}$-complete.*

Proof. The problem is clearly a member of $\mathcal{NP}$ since there exists a finite set of paths between u and v, and given any certificate of such a path, it can be verified in polynomial time whether or not the arrival time is less than or equal to k. It remains to show that the problem is $\mathcal{NP}$-hard.

Given any instance of PARTITION we construct an equivalent instance of TDSPP-DECISION as a time-dependent graph with $n+2$ nodes and $2n+3$ edges as shown in Figure 3.1. Waiting at nodes is not allowed in this graph.

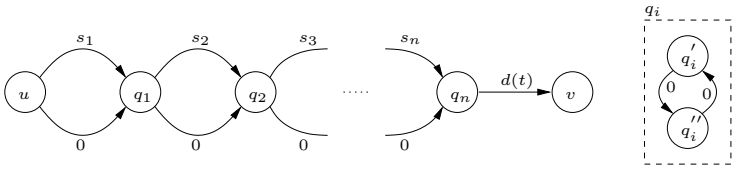


Figure 3.1: An instance of TDSPP corresponding to any instance of PARTITION. The multigraph can be converted to a normal graph by splitting the nodes.

Each integer $s_i \in S$ has a corresponding node q_i, and u and v are added as source and target node. All edges have constant travel time of 0 or s_i except the last edge connecting q_n and v which has the following travel time function

$$d(t) = \begin{cases} 0 & \text{if } r - \varepsilon \leq t \leq r + \varepsilon \\ r + 1 & \text{otherwise} \end{cases} \tag{3.5}$$

where $\varepsilon < 1$ is a small constant. The function $d(t)$ is shown in Figure 3.2 and gives zero only for integer argument r. Notice that the graph is a multigraph but can easily be converted to a normal graph by splitting nodes q_i to q_i' and q_i'' as shown in Figure 3.1. Lastly let $k = r$ and set departure time $t = 0$ to complete the construction of a TDSPP-DECISION instance from any instance of PARTITION. The construction can clearly be done in polynomial time.

The constructed TDSPP-DECISION instance is clearly only a yes-instance when there is a path p to q_n with arrival time $a_p(t) = r$. This results in arrival time $a_{q_n v}(t) = r = k$. All other arrival times at q_n will be penalized so arrival time $a_{q_n v}(t) > r$. Clearly the only way there can be a path p to q_n with arrival time $a_p(t) = r$ is if the PARTITION instance is a yes-instance. We conclude that TDSPP-DECISION is also $\mathcal{NP}$-hard. This completes the proof. □

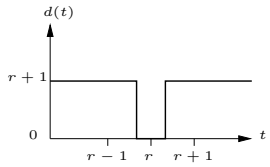


Figure 3.2: Travel time function $d(t)$ of last edge (q_n, v) in the TDSPP instance.

3.5 FIFO Property

Intuitively what makes the constructed instance in Theorem 3.1 hard, is that arriving early at q_n can actually result in a later arrival time at v, than if arriving later at q_n. In a road network this corresponds to a scenario where a car arriving later at an intersection than other cars will pass some of these cars on the road segment because it arrived later. The property of optimal substructure is lost for TDSPP in the general case. This property, that states that subpaths of shortest paths are also shortest paths, is a key property for polynomial solving SSPP. In this section we define and analyze a FIFO property of time-dependent road networks, and we argue that it is fair to assume for road networks. We start by defining a FIFO time-dependent network.

Definition 3.1 (FIFO time-dependent network) *We say that $G = (V, E)$ is a FIFO time-dependent network if the arrival time function is non-decreasing.*

$$a_{uv}(t_1) \leq a_{uv}(t_2) \quad \text{if} \quad t_1 < t_2 \qquad \forall (u, v) \in E \tag{3.6}$$

The property is called FIFO because vehicles will travel through the network in a First-In-First-Out manner. The property is also called the *non-passing* property, because vehicles cannot pass each other on road segments. Figure 3.3 shows an example of an arrival time function violating the FIFO property and one that does not. The vertical lines are two nodes u and v at different times.

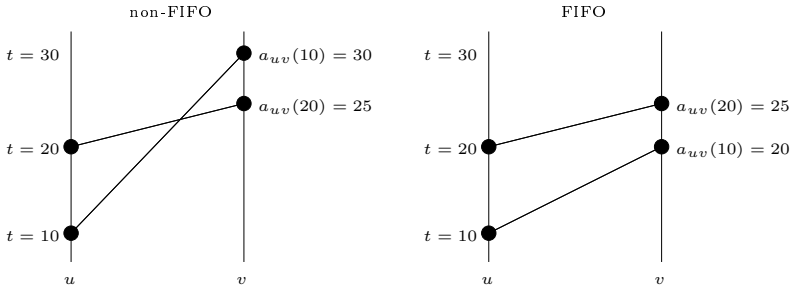


Figure 3.3: A non-FIFO and a FIFO arrival time function.

If the arrival time function is strictly increasing, that is $a_{uv}(t_1) < a_{uv}(t_2)$ if $t_1 < t_2$, we call the network *strictly FIFO*.

For static shortest-paths problems we saw the optimality condition of triangle inequality given in equation (2.3). For earliest arrival times in FIFO time-dependent networks similar optimality conditions apply. The main difference is that addition is replaced by function composition. The following lemma from Dean [11] states necessary and sufficient conditions for optimality.

Lemma 3.1 *The following conditions are necessary and sufficient for optimality of earliest arrival*

times in a FIFO time-dependent network.

$$\delta_{ss}(t) = t \tag{3.7}$$
$$\delta_{sv}(t) \leq a_{uv}(\delta_{su}(t)) \qquad \forall (u,v) \in E \tag{3.8}$$
$$\delta_{su}(t) = a_p(t) \text{ for some } p \in P(s,u) \qquad \forall u \in V \tag{3.9}$$

Condition (3.8) is equivalent to the triangle inequality. Figure 3.4 illustrates the condition.

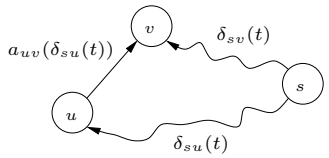


Figure 3.4: Illustration of condition (3.8).

Proof. It is straightforward to see that conditions (3.7) and (3.9) must hold for any optimal solution. Condition (3.8) is necessary because otherwise $\delta_{sv}(t)$ may be reduced to $a_{uv}(\delta_{su}(t))$, thereby contradicting the optimality of the current earliest arrival time values.

To show sufficiency we assume for the purpose of contradiction that the conditions hold for some non-optimal set of arrival times. Since they are non-optimal, there exists some destination node d, departure time t, and path $p = sv_1v_2 \dots v_kd$ for which $a_p(t) < \delta_{sd}(t)$. By repeatedly composing condition (3.8) along the edges in p, we have the following contradiction.

$$\begin{aligned}
a_{sv_1}(t) &\geq \delta_{sv_1}(t) \\
a_{v_1v_2}(a_{sv_1}(t)) &\geq a_{v_1v_2}(\delta_{sv_1}(t)) \\
a_{v_1v_2}(a_{sv_1}(t)) &\geq \delta_{sv_2}(t) \\
a_{v_2v_3}(a_{v_1v_2}(a_{sv_1}(t))) &\geq a_{v_2v_3}(\delta_{sv_2}(t)) \\
a_{v_2v_3}(a_{v_1v_2}(a_{sv_1}(t))) &\geq \delta_{sv_3}(t) \\
&\vdots \\
a_p(t) &\geq \delta_{sd}(t)
\end{aligned} \tag{3.10}$$

□

It was mentioned that we work with time-dependent road networks where waiting at nodes is not allowed. The following lemma shows that in a FIFO time-dependent network it is never beneficial.

Lemma 3.2 *In a FIFO time-dependent network, waiting at nodes will never reduce the arrival time at the destination.*

Proof. The arrival time function is non-decreasing. By equation (3.2) this means that the path arrival function $a_p(t)$ is also non-decreasing. For any path p between nodes u and v in the network, using $t' > t$ instead of t will give no earlier arrival time as $a_p(t) \leq a_p(t')$. □

For static networks with non-negative edge weight there can always be found shortest paths with no cycles. The following lemma states that the same is the case for FIFO time-dependent networks.

Lemma 3.3 *In a FIFO time-dependent network, shortest paths can always be found which are acyclic.*

Proof. A cycle p can always be replaced by waiting at a node u formerly on the cycle. Lemma 3.2 showed that waiting will not reduce arrival time. A shortest path may contain a zero-delay cycle consisting of zero-delay edges, but another shortest path can always be found be removing the cycle. □

TDSPP in the general case did not obey optimal substructure. The next lemma shows that shortest paths in a FIFO time-dependent network has optimal substructure.

Lemma 3.4 *In a FIFO time-dependent network, one may always find shortest paths containing subpaths that are also shortest paths. More precisely, for a given source node s, destination node d and departure time t, one may always find a path $q \in P(s,d)$ for which $a_q(t) = \delta_{sd}(t)$ such that every subpath $p \in P(u,v)$ satisfies $a_p(\delta_{su}(t)) = \delta_{uv}(\delta_{su}(t))$.*

Proof. Suppose for some departure time t and some subpath p of q there exists a shorter path p' for which $a_{p'}(\delta_{su}(t)) < a_p(\delta_{su}(t))$ (see Figure 3.5). By replacing p with p' we can construct a new path q' of no greater length than q.

$$\begin{aligned} a_{p'}(\delta_{su}(t)) &< a_p(\delta_{su}(t)) \\ \delta_{vd}(a_{p'}(\delta_{su}(t))) &\leq \delta_{vd}(a_p(\delta_{su}(t))) \\ a_{q'}(t) &\leq a_q(t) \end{aligned} \tag{3.11}$$

By repeatedly performing this procedure, one will eventually transform q into a shortest path whose subpaths are all shortest paths.

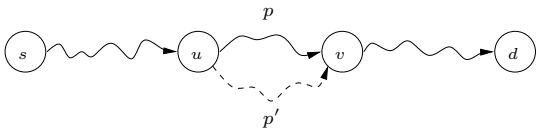


Figure 3.5: Illustration of optimal subpath p.

□

Notice that if the network is strictly FIFO, we can replace less than or equal to in equation (3.11) with strictly less than. This means that subpaths of any shortest path are always shortest paths.

Effectively solving TDSPP clearly rely on the network to be FIFO. Is it fair to assume that road networks are FIFO? This question is the same as asking if arriving later at an intersection will ever be beneficial. The situation may occur at traffic lights where cars arriving just as the light turns green can pass accelerating cars in other lanes, but at the level of detail we are working the answer to the question is no. It is fair to assume that real road networks are FIFO.

3.6 Arrival Time Function

Until now the arrival time function has just been an oracle, somehow returning some estimated arrival time, given departure time. All the theory so far has been independent of the actual arrival time function, except of course for the restriction to non-decreasing functions in the FIFO case. In this section we discuss the domain and structure of different arrival time functions in the context of road networks.

3.6.1 Travel Time Characteristics

Time can either be continuous so functions are real-valued on the domain $\mathbb{R}_+$, or discrete so functions are integer-valued on the domain $\mathbb{Z}_+$. In the continuous case the unit of time does not

matter as any fraction of time can be represented with real-valued time. In the discrete case the unit of time matters. If we use for example minutes we loose a great deal of accuracy if many road segments have travel time less than a minute or rounded to a few minutes. If a path can be constructed between two nodes u and v consisting of many short road segments with travel times rounded to 0, this path will be unfairly favored over other paths. For road networks we will use 10th of a second as time unit to enable the use of discrete time.

In practice the time functions will be defined for some finite *planning horizon* $T = [0, t_{max}]$. This could be a day, month or a whole year. For road networks we assume that all days can be categorized from a finite set of day categories, for example: week day, Saturday or Sunday. More fine tuned categories are possible like Fridays, and special days like travel season days. Given k categories and a corresponding travel time function $w^i_{uv}(t)$ for each category $1 \leq i \leq k$ defined on the time horizon $[0, t_{max}]$, we can use modulo time to define a travel time function $w_{uv}(t)$ for any time $t \geq 0$.

$$w_{uv}(t) = \begin{cases} w^1_{uv}(t \bmod t_{max}) & t \text{ is in category 1 day} \\ \vdots & \\ w^k_{uv}(t \bmod t_{max}) & t \text{ is in category } k \text{ day} \end{cases} \tag{3.12}$$

To avoid continuity problems of the function $w_{uv}(t)$ we will assume that $w^1_{uv}(0) = \cdots = w^k_{uv}(0)$. If $t = 0$ is defined as midnight and t_{max} as a day this is not an unfair assumption. Figure 3.6 shows two different category travel time functions joined at t_{max}.

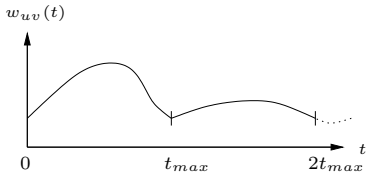


Figure 3.6: Two different category travel time functions joined at t_{max}.

Beside from having different travel time functions for different categories of days, there could also be different travel time functions for different types of vehicles. To simplify matters we will only be working with one type of vehicle and one day category. We define an arrival time function for any time $t \geq 0$ as

$$a_{uv}(t) = w_{uv}(t \bmod t_{max}) + t \tag{3.13}$$

where $w_{uv}(t)$ is a travel time function defined on the planning horizon of one day. Working with integers is generally more effective in a computer than working with floating point values, so we will use integer values for time and round the result of the arrival time functions.

3.6.2 Forecasts

In the context of road networks we cannot expect to have some easily obtained continuous travel time function for each edge. Instead we assume to have delay factor forecasts for roads (paths). We have chosen to work with delay factors as this is assumed to be the most likely representation used by road agencies and other sources of forecasts. Notice that one road r consist of several road segments, so edges $(u, v) \in r$ will share the same delay factors. The delay factors are in the form of a collection $F_r = \{(t_1, d_1), \ldots, (t_k, d_k)\}$ of points, where t_i is time and $d_i \in [1, \infty[$ is forecasted delay, ordered ascending by time. Notice that delay percentage is given by

$$(d_i - 1) \times 100 \tag{3.14}$$

We assume that for each edge (u, v) we know a static travel time $w(u, v)$ which is the best travel time possible, and the one used in static shortest-paths problems. For road networks it seems fair to assume that travel time has a most optimistic lower bound when no congestion or

other problems are present. This is an important assumption when we try to develop speed-up techniques for the time-dependent problem, as lower bounds on travel time will remain valid over time.

Given delay factor d_i a corresponding travel time forecast w_i can be found as $d_i w(u,v)$. Figure 3.7 shows an example of delay factor forecasts. How to collect real forecasts or simulate realistic forecasts is handled in Chapter 6.

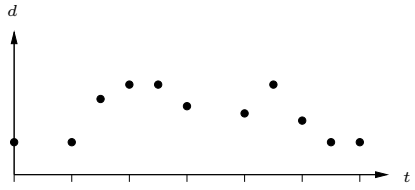


Figure 3.7: Delay factor forecast points.

The points will generally not be equally spaced in time, but more dense around periods where the delay changes. Given delay forecasts we need to convert these into delay factor functions $d_r(t)$ so we can find travel time for edges as

$$w_{uv}(t) = d_r(t)w(u,v) \quad \forall (u,v) \in r \tag{3.15}$$

One way to do this is by interpolation.

Constant interpolation The simplest interpolation method is to locate the nearest time, and assign the same delay. Assuming that the forecast points are kept sorted by time, a binary search yield a $O(\log k)$ running time, where $k = |F_r|$, to find delay. Figure 3.9 a) shows constant interpolation. Constant interpolation will always result in a non-FIFO arrival time function. Consider the simple example shown in Figure 3.8 where the travel time is 20 minutes before the time t' and 10 minutes after t'. The resulting arrival time function is clearly non-FIFO as the function is not non-decreasing.

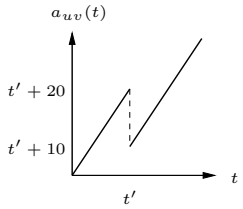


Figure 3.8: A simple non-FIFO arrival time function obtained from constant interpolation of delay factors.

Linear interpolation In linear interpolation the resulting function is a piecewise linear function obtained by connecting neighboring points by straight lines. Figure 3.9 b) shows an example. Between point i and $i+1$ the delay $d^i_{uv}(t)$ is given by

$$d^i_{uv}(t) = d_i + (t - t_i)\Delta^d_i \tag{3.16}$$

where the slope Δ^d_i is given by

$$\Delta^d_i = \frac{d_{i+1} - d_i}{t_{i+1} - t_i} \tag{3.17}$$

The running time of a piecewise linear delay function is a search in $O(\log k)$ to find the linear function to use, and then one multiplication and a couple of adds and a subtraction.

It is only a small constant factor slower than constant interpolation. Linear interpolation can be unprecise if the given data is not sufficient to capture the points at which the function changes. We will rely on the forecast data to be sufficient to allow the use of linear interpolation.

Polynomial interpolation Polynomial interpolation is a generalization of linear interpolation where the linear function is replaced by a polynomial of higher degree. As described in Heath [23, chap. 7], if we have k data points, there is exactly one polynomial of degree $k-1$ going through all the data points. Figure 3.9 c) shows an example. The polynomial is

$$p(x) = a_k x^k + a_{k-1} x^{k-1} + \cdots + a_2 x^2 + a_1 x^1 + a_0 \tag{3.18}$$

Using the *Horner scheme*, Heath [23, chap. 7], the polynomial can be written

$$p(x) = ((\cdots (a_k x + a_{k-1})\, x + \cdots + a_2)\, x + a_1)\, x + a_0 \tag{3.19}$$

This enables solving the polynomial with $O(k)$ multiplications. This is considerable more expensive than solving the piecewise linear function. Another disadvantage of using polynomial interpolation is the behaviour of the function between the data points. The function oscillates and is not smooth between the points. The error is especially big at the ends, which is described by *Runge's phenomenon*, Heath [23, chap. 7].

Spline interpolation Instead of using a single polynomial, spline interpolation uses low-degree polynomials in each of the intervals, and chooses the polynomial pieces such that they fit smoothly together. The resulting function is called a spline. Figure 3.9 d) shows a spline interpolation. Calculating the resulting function requires a lookup phase in $O(\log k)$ time to find the correct polynomial and then an evaluation of that polynomial using h multiplications where h is the degree of the polynomial. Assuming that $h \ll k$ this is considerably more effective than using polynomial interpolation.

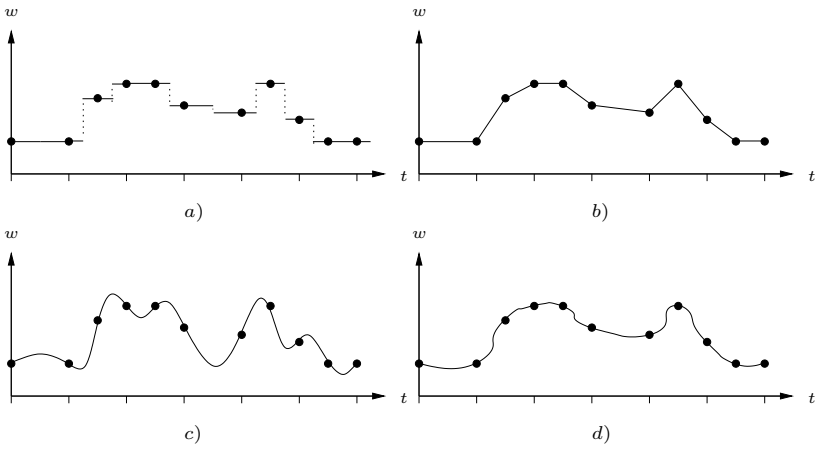


Figure 3.9: Interpolation of forecasts points: a) constant b) linear c) polynomial d) spline.

All the interpolation methods create functions that goes through all the data points. Another way to obtain a function is by *curve fitting* where a function is found that matches the data points as close as possible. One way to do this is by *linear least squares* as described in Heath [23, chap. 3]. Figure 3.10 shows an example.

A lookup operation is needed for some of the interpolation methods because the points are not equally spaced. It is possible though to convert the points into a set of equally spaced points by inserting extra points and moving existing points. This enables lookup in constant time but

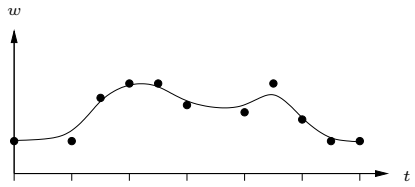


Figure 3.10: Curve fitting of forecasts points.

requires more space. For periods where the travel time does not change much, for example in the night, it is a waste of space to have a lot of points on a straight line. Given that we will work with very large instances, it is expected that the storage usage will be excessive, if using equally spaced points close enough to sufficiently capture the delay.

Spline interpolation and curve fitting give better representations of the underlying forecast data than using linear interpolation, but they have some drawbacks: the process of finding and representing the functions are more complex than using straight lines, and the time to calculate the functions are longer. Another problem is the constraint that the resulting arrival time functions need to be non-decreasing. This is the same as saying that the derivative should be non-negative.

$$a'_{uv}(t) = w'_{uv}(t) + 1 \geq 0 \quad \Rightarrow \quad d'_r(t)w(u,v) \geq -1 \tag{3.20}$$

That is, the slope may not be steeper than -1, which can be interpreted as: the travel time may not decline faster than time. This is a difficult constraint to enforce when creating interpolated or spline functions with higher degree polynomials. We choose to use linear interpolation as it is considered to give sufficient accuracy and is fast to calculate and easy to represent. Furthermore a process to guarantee that the resulting function is FIFO can more easily be found.

3.6.3 Finding Arrival Time

Given a delay forecast F_r we can apply this to an edge (u, v) to construct a travel time function $w_{uv}(t)$. The delay points are converted to travel time points (t_i, w_i) where $w_i = d_i w(u, v)$. Algorithm 3.1 shows how the arrival time can be found given departure time t for a fixed edge (u, v).

The arrival time function is only FIFO if the following holds for all $1 \leq i \leq k - 1$.

$$\Delta_i^w = \frac{w_{i+1} - w_i}{t_{i+1} - t_i} \geq -1 \tag{3.21}$$

Consider the following example where we have a forecast with $d_i = 10$, $d_{i+1} = 1$ and $t_{i+1} - t_i =$ 10 minutes. This means that the delay falls from 10 times longer than normal to normal within 10 minutes. This is a very rapid decline. Assuming that the given decline is the most rapid in the forecast, the forecast can only be used on edges with static travel time

$$\frac{d_{i+1} - d_i}{t_{i+1} - t_i} w(u,v) \geq -1 \quad \Rightarrow \quad w(u,v) \leq 66.7 \text{ seconds} \tag{3.22}$$

without becoming non-FIFO. Notice that the example decline is very rapid and the maximum allowed travel time is rather long when roads consist of many short edges. It is not expected that it will be difficult for forecasts to obey the FIFO property.

3.6.4 Enforcing The FIFO Property

If not all forecasts result in FIFO arrival time functions, we need a way to convert functions into FIFO compliant versions. Notice that if we have a non-FIFO arrival time function $\alpha_{uv}(t)$ we can

Algorithm 3.1 Finding arrival time for fixed edge (u, v) given departure time t.

```
ArrivalTime(t)
1  t' ← t mod t_max
2  i ← FindInterval(t')
3  return w_i + (t' − t_i)(w_{i+1} − w_i)/(t_{i+1} − t_i) + t

FindInterval(t)
 1  low ← 1, high ← k
 2  while low ≤ high
 3        do mid ← ⌊(low + high)/2⌋
 4           if mid = low or mid = high
 5              then return mid
 6           if t ≥ t_mid
 7              then if t < t_{mid+1}
 8                      then return mid
 9                   low ← mid + 1
10              else ▷ t < t_mid
11                   if t ≥ t_{mid−1}
12                      then return mid − 1
13                   high ← mid − 1
```

construct a FIFO arrival time function $a_{uv}(t)$ by incorporating waiting in the following way.

$$a_{uv}(t) = \min_{\tau \geq t} \alpha_{uv}(\tau) \tag{3.23}$$

For the simple arrival time function given in Figure 3.8 this would result in the following arrival time function.

$$a_{uv}(t) = \begin{cases} t + 20 & t \leq t' - 10 \\ t' + 10 & t' - 10 < t < t' \\ t + 10 & t \geq t' \end{cases} \tag{3.24}$$

Figure 3.11 shows a plot of the converted arrival time function. Notice that it is now non-decreasing.

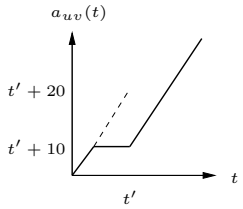


Figure 3.11: Non-FIFO arrival time function in Figure 3.8 converted to FIFO.

We need to convert the piecewise linear travel time function to a FIFO version. We will assume that the travel time functions meet at the ends, so $w_{uv}(0) = w_{uv}(t_{max})$. This is a necessary condition if we are to use the arrival time function given in equation (3.13). Furthermore we will assume that the travel time function has a minimum at $w_{uv}(0)$. If this is not the case the described algorithm should be started in the minimum point i and moved backwards in time modulo k and end back at i.

When converting to FIFO we would like to preserve the height of the delay while reducing the rate of decline. This can be done by moving point i, with $\Delta_i^w < -1$, horizontally back in time until $\Delta_i^w = -1$. Figure 3.12 shows the principle.

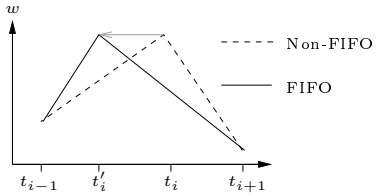


Figure 3.12: Making linear piece i FIFO by moving point horizontally back.

This approach may not be enough because we cannot move point i beyond point $i-1$. In fact we cannot let it be equal to $i-1$, because we cannot handle vertical linear functions as $t_{i+1}-t_i = 0$. Because we work with integers we can move the point to $(t_{i-1}+1, w_i)$. If the linear piece is still not FIFO we have to further lower point i until $\Delta_i^w = -1$. Figure 3.13 shows the principle of lowering point i. Algorithm 3.2 gives a method to make a travel time function FIFO.

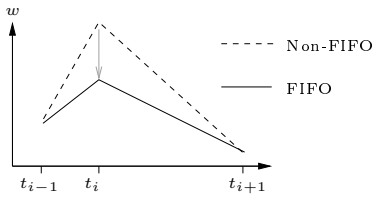


Figure 3.13: Making linear piece i FIFO by lowering point.

Algorithm 3.2 Converting a non-FIFO travel time function to a FIFO version.

```
MakeFIFO()
1  for i ← k − 1 downto 1
2      do if Δ_i^w < −1
3          then Move point i back until Δ_i^w = −1, or as much as possible
4          if Δ_i^w < −1
5          then Lower point i until Δ_i^w = −1
```

Moving point i back can be done by finding new time coordinate in the following way.

$$\frac{w_{i+1} - w_i}{t_{i+1} - t_i} = -1 \quad \Rightarrow \quad t_i = w_{i+1} - w_i + t_{i+1} \tag{3.25}$$

If $t_i \leq t_{i-1}$ we have moved the point too far and in this case we set $t_i = t_{i-1} + 1$. Notice that we assumed the function to have minimum at the start and the end so the first linear piece will always be FIFO ($\Delta_1^w > 0$).

The lowering of point i in line 5 of the algorithm is done by setting the travel time coordinate to the following.

$$\frac{w_{i+1} - w_i}{t_{i+1} - t_i} = -1 \quad \Rightarrow \quad w_i = t_{i+1} - t_i + w_{i+1} \tag{3.26}$$

The correctness of Algorithm 3.2 follows from the loop invariant, that at start of each iteration all points ahead in time of i makes FIFO linear travel time functions. Lowering point i will not make points ahead in time non-FIFO. This requires us to start in a minimum point.

3.7 Problem Variants

The time-dependent problem we work with is a point-to-point earliest arrival time version, where departure time is given. In this section we briefly mention other problem variants and other domains where time-dependent shortest-paths problems occur.

Recall that there are four variants of the static shortest-paths problem. In time-dependent shortest-paths there are 16 variants because of the following four choices:

1. Single or all sources.
2. Single or all destinations.
3. Earliest arrival or latest departure time.
4. Fixed or any time.

The by far two most common are the point-to-point earliest arrival or latest departure time variants with fixed time. Finding the fastest shortest-path at any time could be interesting in for example the trucking industry. Solving such problems where time is not fixed are much more difficult. Dean [11] proposes to solve the problem where source or destination is fixed by $|T|$ shortest-paths computations if time is discrete. This is only viable for very small time domains. Another option is to find a shortest path in the time-expanded network, described shortly, which may be more effective. Given that travel times are only estimates, the optimality could be dropped, and a local search heuristic used to search in T. The more effective we can find a shortest path for fixed time, the more effective such an heuristic will be.

3.7.1 Public Transportation Networks

The most obvious time-dependent networks are public transportation networks. Here nodes are stops where travellers can enter the network, like going on a bus, or shift between different transportation forms. Edges are connections between stops like a bus route or train connection. The network is a multigraph as different transportation forms can link stops. There are different ways to model a public transportation network, but here we will give the pure time-dependent one, later we will mention time-expanded networks.

We will define travel time in terms of earliest arrival time, shortly we will describe a latest departure time variant. The arrival time function for an edge in a public transportation network is simpler than the one we just described. Given a departure time t and edge (u, v) the function returns the earliest arrival time at v using the transportation represented by edge (u, v). Figure 3.14 shows an example of an arrival time function in a public transportation network. The horizontal parts indicate waiting for the next transportation, for example a bus. It does not matter if you wait five or ten minutes for the bus, you will arrive at the same time.

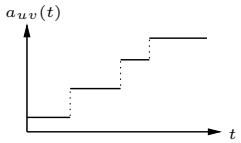


Figure 3.14: Example arrival time function in a public transportation network.

Notice that if transportation forms are not shifted, then the arrival time at node u will yield the "optimal" departure time at u, using the same transportation form. That is, there is no waiting at nodes if the same transportation form is used, for example the same bus.

The arrival time function for a single transportation form has non-negative derivative, so it obeys the FIFO property. If multiple transportation forms are included, the network becomes a multigraph. As long as each edge connecting two nodes u and v have FIFO arrival time function,

an optimal solution will still be found. That you have to wait at a stop, and maybe skip one transportation to catch the express bus, is not the same as saying that waiting will be beneficial. It just tells that the arrival time is constant for a period of time. For k edges $e_1, \ldots, e_k$ connecting u and v with arrival time functions $a_{e_i}(t)$, an equivalent network could be found by replacing all edges with a single edge e with arrival time function $a_e(t) = \min\{a_{e_i}(t) \mid 1 \leq i \leq k\}$. Clearly this arrival time function will also be FIFO.

3.7.2 Latest Departure Time

We can solve the single-destination shortest-paths problem for static networks by reversing edges and start in the destination node. Similarly we can solve the latest departure time problem for time-dependent networks, finding the latest one has to depart from the source to arrive at the destination at a fixed time. To do so, we need the inverted arrival time function called the *latest departure time function* $a_{uv}^{-1}(t)$, which gives the latest one should depart from u to arrive at v at time t.

We now reverse edges as in the static case and start in the destination using the desired arrival time. Notice that we cannot just replace arrival time for edges with $a_{uv}^{-1}(t)$ as $a_{uv}^{-1}(t) \leq t$. This would result in a problem with declining time. Instead we reflect the problem by negating all times and the departure time function. That is, we start at time $-t$ and assign arrival time function $-a_{uv}^{-1}(-t)$ to edges. We can now define a *path latest departure time function* of a path $p = v_0 \ldots v_k$ in the original graph as

$$a_p^{-1}(t) = a_{v_0 v_1}^{-1}(a_{v_{k-1} v_{k-2}}^{-1}(\cdots \ a_{v_{k-1} v_k}^{-1}(-t))) \tag{3.27}$$

FIFO arrival time functions that are not strictly increasing may not be invertible. To overcome this the latest departure time function can be defined as

$$a_{uv}^{-1}(t) = \max\{\tau \mid a_{uv}(\tau) \leq t\} \tag{3.28}$$

This guarantees that the reflected inverse will be FIFO. For the piecewise linear function we use as arrival time in this thesis, the inverted function can be found by first inverting all pieces.

$$a_i^{-1}(t) = \frac{t - w_i + t_i \Delta_i^w}{1 + \Delta_i^w} \tag{3.29}$$

Special handling, as described above, is necessary for $\Delta_i^w = -1$. The piece $(t_i, w_i), (t_{i+1}, w_{i+1})$ to use is the one for which the following holds.

$$w_i + t_i \leq t < w_{i+1} + t_{i+1} \tag{3.30}$$

Suboptimal latest departure time can also be found by using earliest arrival time in a local search heuristic. Given a desired arrival time t, find two departure times so one arrival time is before t and the other after t. Then do binary search until an arrival time close enough to t is found. It is expected that for most uses, only a dozen of iterations are needed to produce an acceptable result.

3.7.3 Time-Expanded Network

For some finite discrete time domain T the time-dependent problem can be converted into a static shortest-paths problem by constructing the time-expanded network $G^T = (V^T, E^T)$ and solving a static shortest-paths problem in this. A time-expanded network has a node set V^T where each node $u_t \in V^T$ represent the original node u at time t. The edges are defined as

$$E^T = \{(u_{t_1}, v_{t_2}) \mid (u, v) \in E \ \wedge \ a_{uv}(t_1) = t_2\} \tag{3.31}$$

That is, an edge exists from node u in time t_1 to node v in time t_2 if they are connected and arrival time is t_2 for departure time t_1.

Using the time-expanded network allows the use of existing static speed-up techniques, but obviously it is only viable for very simple time domains, for example railway timetables. Bauer et al. [5] do an experimental study and conclude that current speed-up techniques have very little effect used on time-expanded networks, and that using a pure time-dependent model is more effective. This confirms that speed-up techniques are dependent on special properties in road networks.

3.8 Dijkstra's Algorithm

The FIFO property enable us to use the label setting algorithm of Dijkstra to solve the shortest-paths problem in a time-dependent network. Algorithm 3.3 shows a compact version of the algorithm. The stopping condition to stop when target node is settled in line 5 is omitted for brevity.

Algorithm 3.3 (DJK) Dijkstra's label setting algorithm

DIJKSTRA(G, s, t)

```
1  d(v) ← ∞   ∀v ∈ V
2  d(s) ← t
3  Q ← V
4  while Q ≠ ∅
5        do u ← argmin{d(v) | v ∈ Q}
6           δ_su(t) ← d(u)
7           Q ← Q\{u}
8           for each v ∈ adjacent(u)
9                 do d(v) ← min{d(v), a_uv(d(u))}
```

The output of the algorithm respects the optimality conditions given in Lemma 3.1, this verifies the correctness. The complexity of the algorithm is the same as for the static version except for extra running time due to the arrival time function. The running time of finding arrival time is $O(\log k)$, where k is the maximum number of forecasts points in any of the arrival time functions. The running time becomes $O(n \log n + n \log k) = O(n \log n)$, assuming $k < n$. The storage complexity for a static network is $O(m + n) = O(n)$. For a time-dependent network we need, in the worst case, to store arrival time functions for each edge, this gives a storage complexity of $O(mk + n) = O(kn)$. In Chapter 6 practical instances and ways to reduce storage complexity will be handled.

Figure 3.15 shows a time-dependent shortest path in Denmark found with algorithm DJK and simple randomized travel time functions close to static travel times. Notice how many nodes (blue) are settled to find the shortest path. We now turn to the subject of improving query time for the time-dependent shortest-paths problem.

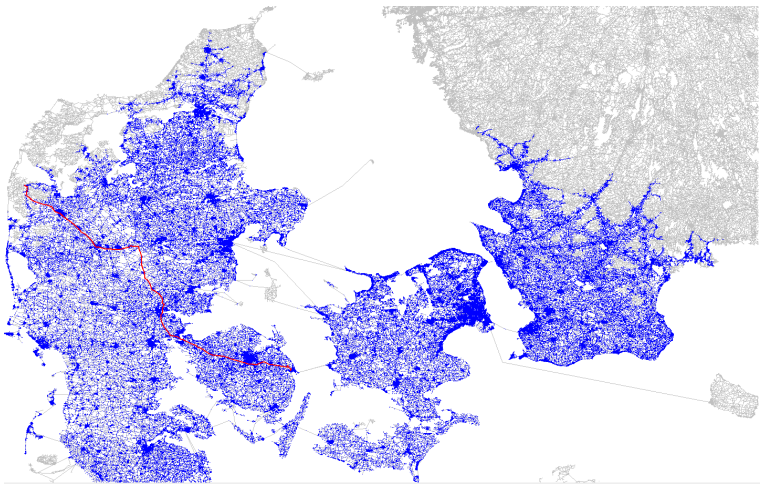


Figure 3.15: A shortest path in Denmark found by Dijkstra's Label-Setting algorithm. Blue nodes indicate settled nodes.

Part II

Speed-Up Algorithms

Chapter 4

Adapting Speed-Up Techniques

We have an algorithm to solve the time-dependent shortest-paths problem under the FIFO property. It uses no preprocessing stage, so it also solves the dynamic time-dependent variant, where the travel time function can change in an unpredictable way, but as for the static problem, Dijkstra's algorithm is simply too slow for very large networks.

To the best of our knowledge, Delling and Wagner [13] are the only ones to have successfully applied a speed-up technique to the time-dependent problem. They achieve speed-ups around five using ALT, corresponding to solution times around 2 seconds for the European road network. They work with a time domain of 24 discrete time intervals. Each edge is assigned a travel time function with a constant travel time for each of the 24 intervals. Notice that this does not lead to a FIFO arrival time function, as indicated in Figure 3.8.

Bauer et al. [5] do an experimental study on speed-up techniques for public transportation networks. They model the problem using three different approaches: condensed model, time-dependent model and time-expanded model. They use existing speed-up techniques on the two static models, but give no speed-up technique for the time-dependent model. They mention that adapting speed-up techniques to the time-dependent problem was more complicated than expected.

In this chapter we discuss if and how existing speed-up techniques can be applied to the time-dependent shortest-paths problem. First some prerequisites and basic limitations are given, followed by a discussion for each category of techniques.

4.1 Prerequisites

When trying to extend speed-up techniques to TDSPP we are faced with a basic limitation. We can search in the opposite direction from the target using inverted arrival time as shown in section 3.7.2, but naturally we do not know at which time to start, as that would be knowing the earliest arrival time we are trying to find. Hence bidirectional search is not a possible speed-up technique for the time-dependent problem. Bidirectional search is an ingredient in many speed-up techniques, so an adaption requires a change of the methods to unidirectional search. It may be difficult to adapt techniques that are inherently bidirectional.

To allow upper and lower bounding of travel time in a time-dependent network, we will define travel time bounds and two derived networks. For a given time interval T we define upper and lower travel time bounds respectively.

$$\overline{w}(u,v) = \max\{w_{uv}(t) \mid t \in T\} \qquad \underline{w}(u,v) = \min\{w_{uv}(t) \mid t \in T\} \tag{4.1}$$

Finding travel time bounds depend on the travel time function. For continuous functions, where the derivative is not easily obtained, approximations could be found by numerical methods. We will use $T = \mathbb{Z}_+$ and find upper bound as maximum forecasted travel time

$$\overline{w}(u,v) = \max\{w_i \mid (t_i, w_i) \in F_{uv},\ 1 \leq i \leq |F_{uv}|\} \tag{4.2}$$

where $F_{uv} = \{(t_i, w_i)\}$ are forecasted travel times. The lower bound is, in our case, simply the static travel time $\underline{w}(u, v) = w(u, v)$ because of the assumption mentioned in section 3.6.2 that travel time is lower bounded by the static travel time. From the travel time bounds we can define two static networks, the *most pessimistic*, and the *most optimistic*, respectively.

$$\overline{G} = (V, E, \overline{w}) \qquad \underline{G} = (V, E, \underline{w}) \tag{4.3}$$

Until now we have used t as both time variable and as a common name for target node. From now on we will use τ as indication of target node and t as time.

4.2 Hierarchy Based Methods

No success was achieved adapting highway hierarchies to a dynamic scenario, the same seems to be the case for the time-dependent problem. The hierarchy is constructed using some static information that determines the importance of edges. A time-dependent search may be possible in the hierarchy, but as the hierarchy dictates when to move up in the hierarchy, the result may not be optimal. Furthermore highway hierarchies are inherently bidirectional, further complicating an adaption. Adapting separator based approaches suffers from the same problems as highway hierarchies. Separator nodes are chosen using static information and they dictate a subsequent search. When travel time changes over time, the separation may not yield an optimal solution. Generally it seems as if hierarchy based methods are not adaptable to the time-dependent problem. Common for all these methods is the construction of a new graph to be searched in subsequent queries. This graph is constructed using static travel times and edge compression is performed replacing a set of edges with one new edge.

Notice that it is possible to combine edges in the time-dependent problem by allowing an ordered set of multiple arrival time functions for one edge. For example for existing edges (u, x) and (x, v), the corresponding arrival time computation for new edge (u, v) will be $a_{uv}(t) = a_{xv}(a_{ux}(t))$. Figure 4.1 shows an example of a contraction in a highway hierarchy.

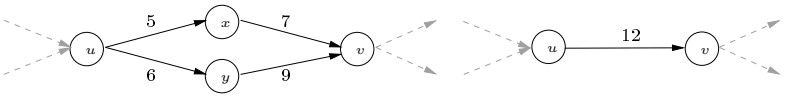


Figure 4.1: A contraction in a highway hierarchy.

The contraction replaces paths uxv and uyv with edge (u, v). In the static problem this contraction does not change optimality because a shortest path going through u and v will always use path uxv with weight 12. To guarantee optimality for the time-dependent problem the new arrival time function has to be

$$a_{uv}(t) = \min\{a_{xv}(a_{ux}(t)),\ a_{yv}(a_{uy}(t))\} \tag{4.4}$$

Unless bounds yield that one path will always be used, so one of the paths can be discarded, the above contraction gives no speed-up because four arrival time functions still needs to be computed.

Transit node routing is not directly a hierarchical method, but it has close ties to hierarchy based methods. It is not applicable for the time-dependent problem, as appropriate transit nodes can change over time.

Edge-labels is another method that is not directly hierarchical, but uses some hierarchy in the network. It seems difficult to adapt edge labels as labels may change over time. For example an edge labeled to indicate that it is used on a shortest path from one cluster to another. If such a label should be used for pruning, there has to be guarantees on when the label is valid.

4.3 Goal Direction

As for the static problem we would like to limit the search space by using goal direction techniques. Intuitively these seem more adaptable, as it is possible to find valid travel time bounds for the time-dependent problem.

4.3.1 A* search

Delling and Wagner [13] give a short description on how to use A* search to solve the time-dependent problem. Here we give the details on how to extend Dijkstra's algorithm.

Assume we have a feasible potential $\pi_\tau(v)$ as lower bound on travel time from v to τ. That is, $w_{uv}(t) \geq \pi_\tau(u) - \pi_\tau(v)$ for any t and edge $(u,v) \in E$. We cannot make a straightforward generalization using *reduced travel time* $w^\pi_{uv}(t) = w_{uv}(t) - \pi(u) + \pi(v)$ and solving a time-dependent shortest-path problem in the network with reduced travel time. This does not produce a correct result, as we need real departure times to find correct travel time. Instead we maintain two distance labels $d(u)$ and $k(u)$, where $d(u)$ is the normal tentative arrival time at node u and $k(u) = d(u) + \pi_\tau(u)$. Recall that in the static problem using $k(u)$ as priority in the heap of nodes, is the same as searching the network with reduced weights. Similarly we use $k(u)$ as priority, but use $d(u)$ to find arrival time. Algorithm 4.1 shows time-dependent unidirectional A* search.

Algorithm 4.1 Time-dependent unidirectional A* search

```
A* SEARCH(G, s, τ, t)
 1  d(v) ← ∞   ∀v ∈ V
 2  k(v) ← ∞   ∀v ∈ V
 3  d(s) ← t
 4  π_τ(s) ← LOWERBOUND(s, τ)
 5  k(s) ← d(s) + π_τ(s)
 6  Q ← V
 7  while Q ≠ ∅
 8        do u ← argmin{k(v) | v ∈ Q}
 9           δ_su(t) ← d(u)
10           if u = τ
11              then return
12           Q ← Q\{u}
13           for each v ∈ adjacent(u)
14                do d(v) ← min{d(v), a_uv(d(u))}
15                   π_τ(v) ← LOWERBOUND(v, τ)
16                   k(v) ← d(v) + π_τ(v)
```

The correctness of the algorithm follows intuitively from the fact that it is actually finding a shortest path in the network with reduced travel time. As we maintain correct tentative arrival times we get the correct travel times as the search progresses.

4.3.2 Landmarks

Using lower bounds obtained from landmarks in the most optimistic network $\underline{G}$ will produce a feasible potential. This can be proven as we know the following to hold for all edges (u,v) in the most optimistic network due to the triangle inequality.

$$\underline{w}_\pi(u,v) \geq 0 \quad \Rightarrow \quad \underline{w}(u,v) + \underline{\delta}(v,l) \geq \underline{\delta}(u,l) \tag{4.5}$$

As $w_{uv}(t) \geq \underline{w}(u,v)$, the inequality also holds for $w_{uv}(t)$. The same applies if bounds are obtained from the shortest-path distances from landmarks.

The quality of the bounds depends on the actual time-dependent travel times for a specific query. If travel times are close to the most optimistic, the bounds will be tighter than if travel times are much higher. We assume the lower bounds to be close to actual travel times for most of the time. High travel times will presumably only occur in short periods and only for specific roads with congestion problems.

A way to obtain tighter bounds would be to find most optimistic networks for periods $T_i \subset T$ of time where the travel time does not vary much. If a query has departure time within a specific period $t \in T_i$ and arrival time can be guaranteed to also be within the period, bounds from this period could be used. Figure 4.2 shows the principle. Guaranteeing that arrival time is within a period requires upper bounds on travel time. Notice that all lower bounds have to be obtained within the same period because otherwise the potential cannot be guaranteed to be feasible. The problem with the approach is mainly space consumption. A single set of landmarks already requires a substantial amount of space so it is prohibitive to store additional sets of bounds.

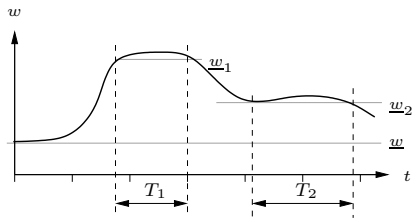


Figure 4.2: Different lower bounds for different periods of time.

As described by Delling and Wagner, the landmarks technique can be extended to the dynamic shortest-paths problem. Assuming travel time updates do not result in lower travel times than in the most optimistic network, landmarks can also be extended to the dynamic time-dependent shortest-paths problem.

We implement a version of the ALT algorithm using a slightly modified version of farthest landmark technique to find landmarks. Firstly we define landmark distance by lowest number of hops instead of shortest-path distance. As described by Goldberg et al. [19] this gives slightly better results because dense regions are favored. Secondly k landmarks are selected by the following steps:

1. An initial landmark is selected randomly.
2. Farthest landmarks are added until $2k$ landmarks have been selected.
3. The initial landmark is removed.
4. Landmarks are removed randomly until $k/4$ landmarks remain.
5. Farthest landmarks are added until k landmarks exist.

The above approach seems to give slightly better results than just choosing k farthest landmarks. Finding a farthest landmark, given a number of existing landmarks, can be done by growing a single shortest-path tree rooted in an added node v' connected to all existing landmarks by zero weight edges.

A detailed description of a farthest landmark algorithm is not given here as it is nearly identical to Algorithm 5.1 given in section 5.2.2. This algorithm finds cluster centers for a k-center clustering where centers are the same as farthest landmarks.

Figure 4.3 shows a shortest path in Denmark found using ALT with the 16 landmarks shown and constant travel time functions. Compare the number of settled nodes (blue) with that of Dijkstra's algorithm given in Figure 3.15. Chapter 8 gives a detailed experimental evaluation of time-dependent ALT.

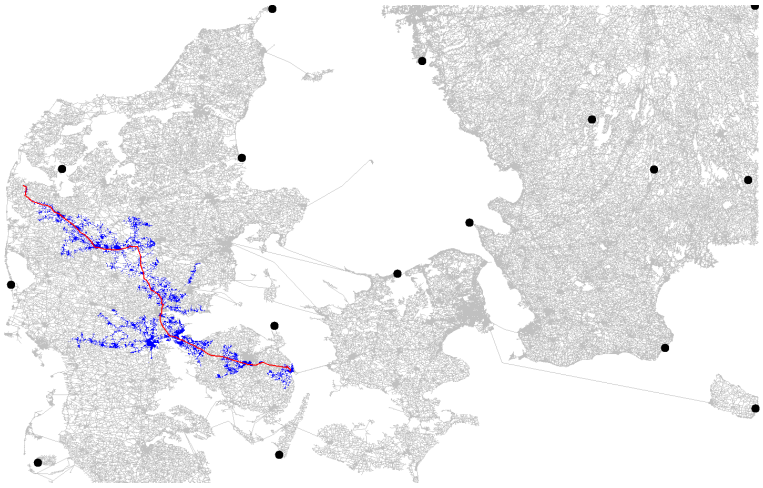


Figure 4.3: A shortest path in Denmark found using ALT with the 16 landmarks shown. Blue nodes indicate settled nodes.

4.3.3 Precomputed Cluster Distances

Recall the PCD method described in section 2.4.5. It maintains lower and upper bounds for the shortest path from s to τ to achieve goal direction. We have just seen that bounding can be used to speed-up the time-dependent problem, so it is natural to try to adapt PCD. We are mainly faced with two differences compared to the static problem. The search needs to be unidirectional, and we need both upper $\overline{\delta}(U, W)$ and lower bounds $\underline{\delta}(U, W)$ on precomputed cluster distances. That is, $\overline{\delta}(U, W)$ is the most pessimistic and $\underline{\delta}(U, W)$ the most optimistic shortest-path distance between clusters U and W. We now describe a search for the time-dependent problem. You may want to revisit section 2.4.5 and Figure 2.10.

Again assume we have k clusters, and S and T are the clusters of s and τ respectively. We add an extra phase to the search.

1. A normal time-dependent Dijkstra search is performed from s until τ or the first border node s' is settled.

2. A reversed static Dijkstra search is performed from τ, with respect to static edge weights $\underline{w}(u, v)$, until the first border node τ' is settled. By this search we have obtained $\underline{\delta}(\tau', \tau)$, the most optimistic shortest-path distance from any border node τ' to τ.

3. Continue to grow the time-dependent shortest-path tree from s while maintaining upper bound $\overline{\delta}(s, \tau)$ and lower bound $\underline{\delta}(s, w, \tau)$. The edges out of w are pruned if $\underline{\delta}(s, w, \tau) > \overline{\delta}(s, \tau)$. Stop when settling τ.

The upper bound is updated when a node $u \in U$ is settled and u is the starting point of the most pessimistic shortest path from U to T. The following holds

$$\delta(s, \tau) \leq \delta(s, u) + \overline{\delta}(U, T) + \theta(T) \tag{4.6}$$

where $\theta(T) = \max\left\{\overline{\delta}(v_1, v_2) \mid v_1, v_2 \in T\right\}$ is the most pessimistic diameter of T. As $\delta(s, u)$ is known exact, $\overline{\delta}(U, T)$ is an upper bound on the distance from u to T, and $\theta(T)$ is an upper bound on the distance from any border node of T to τ, the bound is valid.

When a node $w \in W \neq T$ is settled the lower bound can be found as

$$\underline{\delta}(s, w, \tau) = \delta(s, w) + \underline{\delta}(W, T) + \underline{\delta}(\tau', \tau) \tag{4.7}$$

The shortest-path distance $\delta(s, w)$ is exact, clearly $\underline{\delta}(W, T)$ is a lower bound on the distance from w to $v \in T$, and $\underline{\delta}(\tau', \tau)$ is the minimum distance from any border node τ' to τ.

The preprocessing stage is a little different. Partitioning can either be done with regard to $\underline{G}$ or $\overline{G}$, and we need two sets of distances and starting and end points for the shortest paths between clusters.

One concern adapting PCD to the time-dependent problem is the quality of the upper bounds. If pessimistic cluster distances are considerably higher than the most optimistic, less pruning is possible. We will try an experimental evaluation. Chapter 5 will give the details on an adaption of PCD to the time-dependent problem.

4.4 Reach-Based Routing

At first sight one may be tempted to think, that the notion of reach is directly adaptable to the time-dependent problem. We work with upper bounds on reach, so maybe we could just find reach bounds $\overline{r}(v)$ in the pessimistic network $\overline{G}$. Recall that we can prune a node v if the following holds

$$\overline{r}(v) < \underline{\delta}(s, v) \text{ and } \overline{r}(v) < \underline{\delta}(v, t) \tag{4.8}$$

where $\underline{\delta}(u, v)$ is lower bound on shortest-path distance, for example obtained from landmarks in the optimistic network $\underline{G}$. We will now show, by an example, that upper bounds on reach found in $\overline{G}$ does not necessarily lead to valid pruning.

Consider the time-dependent network shown in Figure 4.4. The only time-dependent edge is (v, y) that has $\underline{w}(v, y) = 1 - \epsilon$ and $\overline{w}(v, y) = 1 + \epsilon$ for a small ϵ.

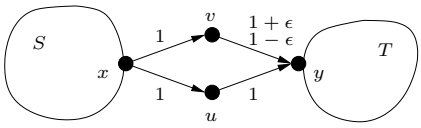


Figure 4.4: Example time-dependent network.

Reach found in $\overline{G}$ will for v give $r(v) = 1 + \epsilon$ because all shortest paths from S to T will use xuy. Assume that $w_{vy}(t) = 1 - \epsilon$ at most times. Clearly all shortest paths will go over xvy in this case. If we use reach based pruning, descent lower bounds will prune node v due to the conditions in equation 4.8. Clearly we will not find a shortest path. The same problem exists when finding reach in $\underline{G}$.

The problem adapting the notion of reach is not surprising, given the similarity with highway hierarchies noticed by Goldberg et al. [17]. Reach also depends on a hierarchy of edges, that change when edge weights change over time.

4.5 Heuristics

In a time-dependent problem, travel time functions may only be forecasts, so optimality may not be very important. Instead of using exact algorithms we may be able to use faster heuristics. For example it may be possible to use a static speed-up technique in a time-dependent setting by accepting suboptimal solutions. The problem with heuristics is that they give no bound on the quality of solutions. The heuristic may be close to optimal nearly all the time, but have large deviations for some queries. Consider for example route planning software that gives a route 20% longer than the optimal. This is clearly unacceptable for the customer, forecasted travel times or not. We have chosen not to consider heuristics in this thesis.

Chapter 5

Precomputed Cluster Distances

In this chapter we give the details on adapting precomputed cluster distances to the time-dependent problem.

5.1 Directed Version

Before we can start an adaption of PCD we need to consider one fundamental difference between the method presented by Maue et al. [30] and the adaption we try to achieve. Maue et al. describe the method in the setting of undirected graphs, which simplifies many tasks. As there is no difference between out- and ingoing edges one can freely search in any direction, and the fact that $\delta(u, v) = \delta(v, u)$ for any pair of nodes can be exploited in the partitioning. We are faced with a challenge regarding how to represent the network. We can either use a simple *adjacency list* representation, that gives access to the outgoing edges of a node. Or we can use a *bidirectional* adjacency list representation, so both in- and outgoing edges can be accessed effectively.

The main motivation for using the simple representation is space consumption, which will be doubled in the bidirected representation. The downside is that we cannot do the reverse search in phase 2 and therefore lower bounds will be less tight. Furthermore partitioning is complicated as we literally have to reverse direction on edges to find border nodes and neighboring clusters. We choose to use a bidirected representation.

5.2 Partitioning

The first step of preprocessing is a partitioning of the road network into k disjoint clusters $V = V_1 \cup \cdots \cup V_k$. We would like the partitioning to result in clusters with low diameter, as this will strengthen the lower bounds found in equation 4.7. This is because the precomputed distance between W and T is used, which does not include the distance between $w \in W$ and the starting node on the precomputed path to T. This distance can be up to the diameter. We would furthermore like the clusters to be of similar size, so phase 1 of a query does not require a large search space for some starting clusters. Maue et al. [30] achieve best results with a k-center clustering, so we will adopt this clustering. We make the clustering in the most optimistic network $\underline{G}$ as we expect random queries to be closer to the most optimistic than the most pessimistic travel times.

5.2.1 k-center Clustering

If we have a cluster of nodes $S \subset V$ we define the *cluster distance* $\delta(v, S)$ from a node $v \in V$ to S as

$$\delta(v, S) = \min \{\delta(v, s) \mid s \in S\} \tag{5.1}$$

For any set $C = \{c_1, \ldots, c_k\}$ of k distinct *centers*, assigning each node $v \in V$ to the closest center will result in a k-center clustering. In the k-center problem one tries to minimize $\max_{v \in V}\{\delta(v, C)\}$. That is, minimize the shortest-path distance from any node to the closest cluster center. This is known to be an $\mathcal{NP}$-hard problem, so we cannot expect to find an optimal solution for large networks.

Before looking at algorithms to determine a good set of cluster centers, we need to address the problem of finding a clustering given a set C of centers. This can be done by growing a single shortest-path tree rooted in an added dummy node v' that is connected to all centers by added zero weight edges. In practice this can be achieved by adding all centers to the heap with zero priority before growing the tree with Dijkstra's algorithm. The tree is grown in the reverse direction so we find distances from nodes to centers. While growing the tree an additional label $c(v)$ is maintained for each node. From the start the labels are initialized with the following values.

$$c(v) \leftarrow v' \ \forall v \in V \backslash C \qquad c(c_i) \leftarrow c_i \ \forall c_i \in C \tag{5.2}$$

When a node v is relaxed, that is a new tentative shortest-path distance has been found, we set the cluster label $c(v)$ to the cluster of the predecessor $c(p(v))$. This way a tree is grown from each center capturing the nearest nodes. Figure 5.1 shows the principle for a graph with three centers.

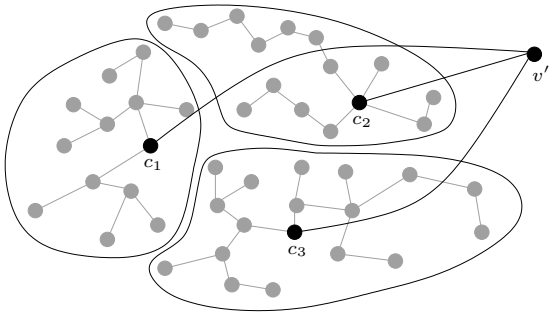


Figure 5.1: Finding a k-center clustering for three cluster centers by growing shortest-path tree from v'.

5.2.2 Greedy Approximation Algorithm

Different approximation algorithms exist for the k-center problem with approximation guarantee 2. Hochbaum [24] proves that no better approximation guarantee exists unless $\mathcal{P} = \mathcal{NP}$. Many of the algorithms have time complexity prohibitive for use in large networks, but Gonzalez [20] proves that a simple greedy algorithm also have guarantee 2. The algorithm is given as Algorithm 5.1.

Algorithm 5.1 Greedy algorithm for k-center

```
GreedyCenter(G, k)
1  Pick random node c_1
2  C ← {c_1}
3  while |C| < k
4        do v ← argmax{δ(v, C) | v ∈ V} ▷ Farthest node v from C
5           C ← C ∪ {v}
6  return C
```

Finding the farthest node v in line 4 can be done by finding the corresponding clustering as described above. The last node settled is the farthest node from any of the cluster centers. The

running time of the algorithm is $O(kn \log n)$ for a road network. For very large networks this can give quite long preprocessing time, but no worse than finding cluster distances.

The approximation algorithm may have a guarantee on the quality of solutions, but it seems that solutions are often close to this guarantee instead of close to optimal. Figure 5.2 shows a greedy clustering of Denmark and Southern Sweden, and the Netherlands with $k = 64$.

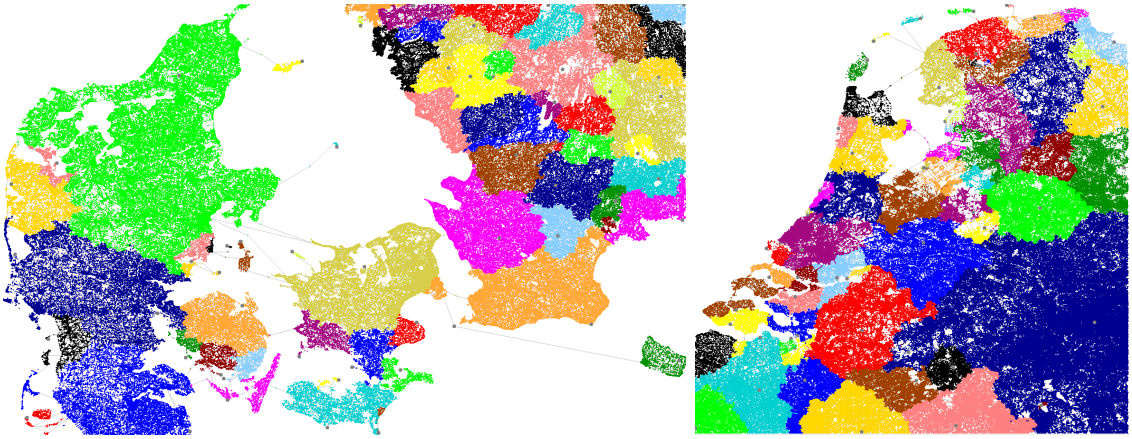


Figure 5.2: Greedy clustering of Denmark and Southern Sweden, and the Netherlands with $k = 64$.

Notice that clusters have very uneven size and cluster centers tend to be placed on small remote islands as this is the typically where farthest nodes are located. The greedy approximation does not deliver a satisfactory clustering.

5.2.3 k'-oversampling Heuristic

Maue et al. [30] propose a k'-oversampling heuristic to find a k-center clustering. A sampling set C' of $k' \geq k$ centers are chosen randomly and iteratively centers are removed until $k' = k$. The algorithm is given as Algorithm 5.2.

Algorithm 5.2 k'-oversampling heuristic

```
OVERSAMPLING(G, k')
1  Make random sampling C' of k' distinct nodes
2  Make clustering based on C'
3  while |C'| > k
4        do Remove a center c_i, C' ← C'\{c_i}
5           Remake clustering based on C'
6  return C'
```

The decisive point in the algorithm is the selection of the center to remove in line 4. Different selection rules can be used to achieve different objectives. As noted earlier we are interested in clusters of roughly the same size and with low diameters. Maue et al. [30] propose three different rules:

MinRad Remove the cluster with minimum radius.

MinSize Remove the cluster with minimum size.

MinRadSize Alternate between removing with MinRad and MinSize.

Other rules are also possible, for example to weight radius and size to make the selection. It is also possible to use more elaborate techniques, where the cluster removed is the one that results in a new clustering with minimum maximum size, or minimum diameter. This technique requires more work as resulting clusterings have to be examined for all possible removings. A selection rule is not only motivated by the objective to be achieved, but also by running time restrictions. We work with large networks, so it is infeasible to employ elaborate selection rules. The easiest rule to employ is MinSize for which Maue et al. [30] actually also achieve the best results. They use an oversampling count of $k' = k \log_2 k$. We will also use this strategy, furthermore it is much simpler than to find diameter which is computationally infeasible in the directed case.

The running time of the algorithm depends on how a clustering is reestablished in line 5. Doing a complete remake in each iteration is prohibitive. Notice that removing a cluster and growing a new shortest-path tree from the dummy node, then all shortest paths from nodes in the deleted cluster will go through the neighboring clusters. We can therefore just restart the search by the following steps:

1. Go through all nodes v in the deleted cluster and determine neighboring nodes by looking at the target u of outgoing edges (v, u). If $c(v) \neq c(u)$ then u is a neighboring node. That is, u connects the deleted cluster to a remaining cluster.
2. Add all neighboring nodes to the heap used in the Dijkstra search and label them as discovered (not settled).
3. Reset labels of all nodes in the deleted cluster so they appear undiscovered.
4. Start growing the tree again. Neighboring nodes will be settled and edges from the deleted cluster relaxed.

Figure 5.3 shows the principle of regrowing a shortest-path tree from the neighboring nodes. Figure 5.4 shows two k'-oversampling clusterings with $k = 64$, and respectively $k' = 64$ and $k' = 384$. For the clustering with $k' = 384$ the result is better because cluster sizes are more even.

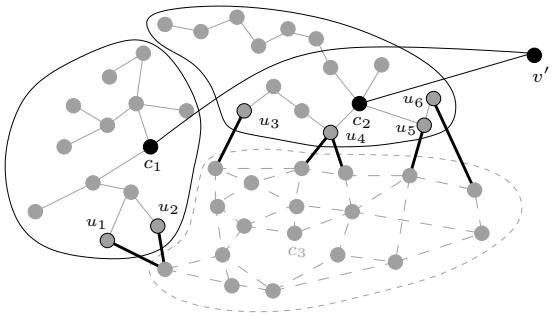


Figure 5.3: Regrowing a shortest-path tree from neighboring nodes $u_1, \ldots, u_6$. Cluster with center c_3 has been deleted.

We know that road networks are nearly planar and degree of nodes is a small constant, so it is fair to assume that clusters have a constant number of neighboring clusters. On average the clusters will have size $O(n/k)$ and smaller when k' is larger than k. It is fair to assume that reestablishing a clustering is much faster than a complete remake, and the whole algorithm faster than $O(kn \log n)$ for $k' = k \log_2 k$. Experiments verify this, as each iteration in the algorithm only takes around 3 milliseconds for the road network of Denmark and Southern Sweden.

5.3 Cluster Distances

We now have a clustering so each node v has a cluster label $c(v)$ and for each cluster $1 \leq i \leq k$ we have a set V_i of nodes. Before finding cluster distances we need to determine whether nodes

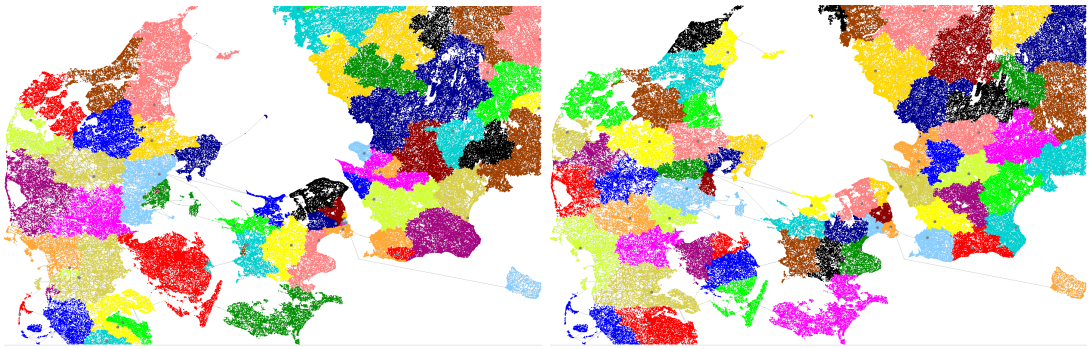


Figure 5.4: Two k'-oversampling clusterings of Denmark and Southern Sweden with $k = 64$, and $k' = 64$ (left) and $k' = 384$ (right).

are border nodes. We do this as above, iterating through all nodes and looking at the target of outgoing edges. For each cluster i we maintain a set of border nodes B_i^+. Notice that there are two kinds of border nodes in the directed case. Those that have incoming edges from other clusters B_i^-, and those that have outgoing edges to other clusters. For now we talk of latter B_i^+, as distances to other clusters have to be found.

For each pair of clusters we need upper and lower bounds on shortest-path distances, and the source and target node of the upper bound shortest path. Figure 5.5 shows the principle.

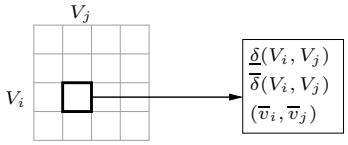


Figure 5.5: Distance matrix for cluster distances.

The cluster distances and source and target node can be found as described by Algorithm 5.3. The running time is dominated by growing shortest-path trees, so the running time is $O(kn \log n)$. Notice that distance to cluster V_j is found when the first node $u \in V_j$ is settled.

Algorithm 5.3 Cluster distances

ClusterDistances(G, k)
1 Add node v'
2 **for** $i \leftarrow 1$ **to** k
3 **do** Add zero weight edges from v' to all nodes in B_i^+
4 Grow SP-tree from v' with regard to $\underline{w}$ and maintain $\underline{\delta}(V_i, V_j)$ as search progresses
5 Grow SP-tree from v' with regard to $\overline{w}$ and maintain $\overline{\delta}(V_i, V_j)$ as search progresses
6 Furthermore maintain $\overline{v}_j \in V_j$ as target node for each shortest path
7 For each target $\overline{v}_j$ use predecessor graph to find corresponding source node $\overline{s}$
8 Remove added edges from v'
9 Remove node v'

5.4 Target Cluster Travel Time Bounds

The only remaining task in preprocessing is finding the most pessimistic upper bound $\phi(T)$ on travel time from a border node b in cluster T to target node $\tau \in T$. We need this to be able to compute the upper bound on travel time from s to τ. We have different choices on how to obtain $\phi(T)$. Until now we have only mentioned the most pessimistic diameter $\theta(T)$ used by Maue et al. [30]. In the undirected case they find the radius $r(T)$ of clusters in the partitioning phase, and use $2r(T)$ as diameter bound, if the exact distance has not been found in the reverse search direction. In this section we discuss different ways to obtain $\phi(T)$ in the directed unidirectional case.

5.4.1 Cluster Node Distances

Before discussing how to find $\phi(T)$ we need to realize the fact that shortest paths between two nodes in the same cluster S do not necessarily lie within S. We know that the two nodes are closest to same center in $\underline{G}$, but this does not mean that shortest paths between them are guaranteed to be solely inside S. Finding shortest paths between nodes of the same cluster cannot be done by pruning nodes of other clusters. Doing so may give an upper bound, but problems occur if there is no path between the nodes inside the cluster. It is also important to realize that when shortest paths are found in $\overline{G}$ the shortest paths from nodes to centers may have changed, and may not even lie within S.

In quering a search is performed backward from the target node to the first border node with incoming edges from other clusters. This search stops when this node is settled and perform no pruning based on cluster.

5.4.2 Diameter Upper Bound

Finding the exact diameter $\theta(T) = \max\left\{\overline{\delta}(u_1, u_2) \mid u_1, u_2 \in T\right\}$, or the longest shortest path, is computationally infeasible. Instead we can find an upper bound as the sum of the longest shortest-path distance from the cluster center c_i to any other node in the cluster v': $\overline{\delta}(c_i, v')$, and the longest shortest-path distance from any node u' in the cluster to c_i: $\overline{\delta}(u', c_i)$. Figure 5.6 shows the principle.

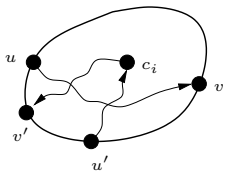


Figure 5.6: Diameter $\overline{\delta}(u, v)$ and upper bound $\overline{\delta}(c_i, v') + \overline{\delta}(u', c_i)$.

This upper bound can be found by a forward and a reverse search from each cluster center. As mentioned before this search cannot be made effective by just searching inside the cluster. Instead of growing complete shortest-path trees a counter can be used to detect when all nodes of the cluster have been settled. The actual number of settled nodes is hard to estimate because the cluster can contain distant nodes on ferry connected islands. The shortest-path tree grown can become large before the last cluster nodes are settled.

The quality of the bound depends on how the center is placed in the cluster. The bound may not be good if the cluster center is not evenly distanced to the cluster nodes. This is often the case if a cluster contains ferry connections. These have much higher travel times, and the diameter bound will approach the double of the actual diameter.

5.4.3 Border Node Bound

We really don't need the diameter, but the longest shortest path from any border node $b \in B^-(T)$. We can find this by for each cluster, and for each border node, compute the longest shortest-path distance to any other node within the cluster. Again we can limit the search by keeping a counter to record when all cluster nodes have been settled, but expect that the search space can be quite larger than the cluster size, for at least some queries. The running time seems prohibitive for this approach, as the number of border nodes is quite high.

5.4.4 Access Node Bound

When finding most pessimistic cluster distances we have found a set of target nodes $A(T)$ for each cluster T. That is, the nodes $v \in T$ that are the target of the shortest paths from the other clusters. We call this set the *access nodes*. Inspired by transit nodes, we believe the number of access nodes for each cluster to be small. Notice that many clusters will share an access point in a cluster T. Clearly $A(T) \subseteq B^-(T)$, so all access nodes are border nodes. The idea is now to find the longest shortest-path distance from any access node.

$$\phi(T) = \max\left\{\overline{\delta}(a, T) \mid a \in A(T)\right\} \tag{5.3}$$

This can be done as for the border nodes. Find longest shortest paths from each access node and select the longest. We now prove that using this bound will give an upper bound on the most pessimistic distance $\overline{\delta}(u, \tau)$ from a node $u \in U$ to $\tau \in T$. Figure 5.7 shows the situation before trying to find an upper bound on travel time from s to τ. We need to show that $\overline{\delta}(U, T) + \overline{\delta}(a, T)$, where a is the access node starting the longest shortest path in T, is an upper bound for $\overline{\delta}(u, \tau)$.

$$\begin{aligned} \overline{\delta}(u, \tau) &\leq \overline{\delta}(U, T) + \overline{\delta}(\tau_{UT}, \tau) \\ &\leq \overline{\delta}(U, T) + \overline{\delta}(a, T) \end{aligned} \tag{5.4}$$

The last inequality holds because $\overline{\delta}(\tau_{UT}, \tau) \leq \overline{\delta}(a, T)$ when $\tau_{UT} \in A(T)$ and $a \in A(T)$ is the farthest access node.

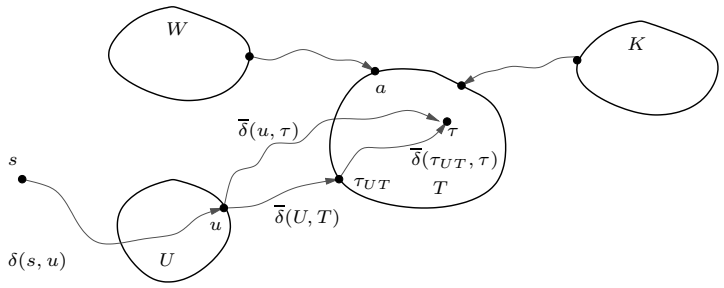


Figure 5.7: Using farthest access node a to find upper bound on $\overline{\delta}(u, \tau)$.

The bound we obtain is at least as tight as the one obtained considering all border nodes, as fewer border nodes are considered. Assuming a small number of access nodes, it is much faster to find the bounds for each cluster.

Another possibility giving tighter bounds would be to store longest shortest-path distances from each access node and using the appropriate when finding upper bound. For example the one starting in τ_{UT} when finding upper bound from u.

5.4.5 Query-Time Access Node Bound

To achieve the tightest possible bounds, the distances $\overline{\delta}(a, \tau)$ from all $a \in A(T)$ could be found at query time. This requires an extra phase (2b) in the quering, where a reverse search is done from

τ in $\overline{G}$. This way $\overline{\delta}(\tau_{UT}, \tau)$ will be known exact for all clusters U. The bounds are tighter, but at the cost of an extra reverse search from τ.

To find exact distances the reverse search would have to grow a shortest-path tree until all access nodes have been settled. As discussed, this can require settling more nodes than just the cluster, and sometimes considerably more if we look for distant nodes. But notice that access nodes are important nodes that are on the shortest paths from other clusters. Therefore we don't expect them to be remotely located.

If we relax the constraint that distances have to be exact, we can use another strategy, where only the inside of the cluster is searched. This clearly delivers an upper bound, but only if there is a path from each access node to the target inside the cluster. We will use the following method to find distances.

1. Grow a reverse shortest-path tree from τ, pruning nodes outside the cluster. If all access nodes have been settled we are done.

2. Grow a reverse shortest-path tree from τ. Stop when all access nodes have been settled.

Preliminary experiments shows that step 2 is very rarely necessary, and on average twice the number of nodes are settled in this step. The distance found in step 1, if available, is only seldom different from the exact distance. Preliminary experiments also shows that a good upper bound on travel time within the target cluster is very important for the speed-up of PCD. Finding the bound at query time gives by far the tightest bounds, and the time of the reverse search is negligible. We therefore choose this approach.

5.5 Quering

Quering is a pretty straightforward extension of the normal Dijkstra algorithm. Phase 1 and 3 are implemented as the same search except for a flag indicating whether the search is in S or not. If not, updating bounds and pruning are performed. The reverse searches in phase 2 are entirely separate searches, performed when the first border node s' is settled. The reason it is after phase 1, is to spare the search if the target node is in the source cluster. Updating bounds is done just after settling node u. If the lower bound is smaller than the upper bound, the edges out of u are pruned.

Between queries labels need to be reset. Instead of iterating through all n nodes and resetting labels, those updated in a query could be saved on a stack and only those reset after the search.

Figure 5.8 shows a time-dependent shortest path in Denmark found using precomputed cluster distances with $k = 1024$ and constant travel times. Compare the number of settled nodes (blue) with that of Dijkstra's algorithm given in Figure 3.15. Chapter 8 gives a detailed experimental evaluation of time-dependent PCD.

Precomputed cluster distances can easily be combined with ALT. Pruning will be performed on the correct distances found, while reduced travel time will control the order of settling nodes. Therefore the combination can be seen as ALT with some possible PCD pruning. In Chapter 8 an experimental evaluation is done to determine if a combination can produce an additional speed-up.

5.6 Dynamic Update

Assuming that updates do not affect the lower bounds on travel time, the lower bounds on cluster distances are not affected either. As clusters are found based on lower bounds the quality of clusters are also not affected. The upper bounds, on the other hand, can invalidate the pessimistic cluster distances. An increase in upper bound on travel time for an edge can affect cluster distances, and start and end nodes on the shortest paths. It seems difficult to make an effective update of these attributes. A simple approach could be to add the increased travel time of an edge to all pessimistic cluster distances. This makes the upper bounds valid but less tight. This approach is

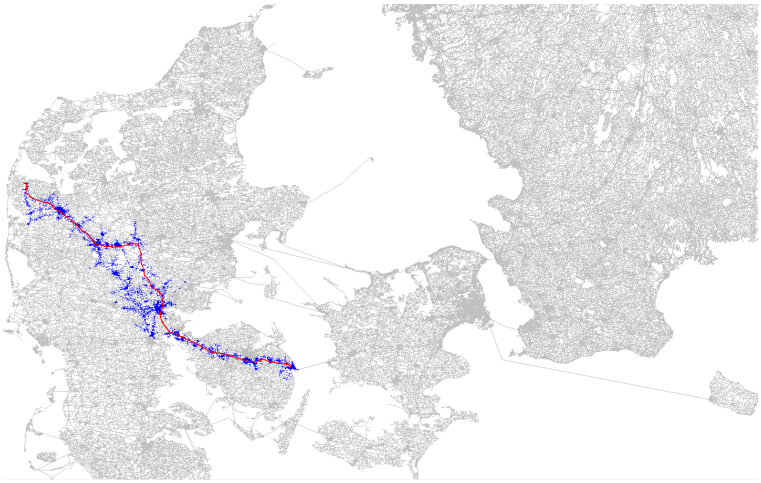


Figure 5.8: A shortest path in Denmark found using precomputed cluster distances with $k = 1024$. Blue nodes indicate settled nodes.

only viable if the changes are only temporary, and travel times will return to normal at a given time.

Part III

Practical

Chapter 6

Problem Instances

To test the algorithms of this thesis we need time-dependent road networks. Unfortunately real reliable forecasts on travel time are not available at a large scale at the moment. In this chapter we describe how real forecasts can be found, what data we have available, and how simulated forecasts could be used in the lack of real forecasts. Finally we give a practical approach using some rough assumptions to find simulated forecasts that have some touch of realism.

6.1 Real Forecasts

Real reliable forecasts may be lacking, but progress is being made. In Denmark the roads are monitored and maintained by The Road Directorate under the Ministry of Transport and Energy [34]. Christensen [6] from the directorate has informed us, that currently measuring delay is done only for a number of roads, primarily those with frequent congestion problems. The measuring is done at a number of stations with magnetic sensors in the road, able to measure the current speed of traffic in intervals of 15 minutes. This approach is good to determine if a road is congested or not, but not a very good way to find the delay of the whole road. When traffic is at a total stop over the sensors, this does not mean that the delay is infinite for the whole road. Another problem with the approach, is that it is impossible to employ for a large percentage of roads in a network.

A more ideal way to obtain precise forecasts would be to equip vehicles with devices using GPS to record travel time of road segments and time of day, and then send this information to a central database. For only a few percentage of all vehicles, this would generate an enormous amount of data and enable generation of accurate forecasts for road segments. The method is currently being employed in Great Britain, and in Denmark there are plans to start experimenting. We expect that other European countries and the United States have similar plans.

6.2 Available Data

In this section available data and methods to deduce additional data are described.

6.2.1 Road Networks

Occasionally an implementation challenge on different shortest-paths problems is held by DIMACS [15]. On the homepage for the 9th DIMACS Implementation Challenge [1] there is different road networks from USA available in a simple format. The road networks are directed graphs with the following attributes for nodes and edges.

Edge distance For each edge $(u, v) \in E$ we have the edge distance $d(u, v) \in \mathbb{Z}_+$ in meters.

Edge travel time For each edge $(u, v) \in E$ we have the static edge travel time $w(u, v) \in \mathbb{Z}_+$ in 1/10 seconds.

Node coordinate For each node $u \in V$ we have the longitude and latitude $(x_u, y_u) \in \mathbb{Z} \times \mathbb{Z}$; one unit corresponds to 1/100000 degree, and positive and negative numbers stand for the Eastern and the Western hemisphere, respectively.

Table 6.1 shows 12 USA road networks with number of nodes n and number of edges m.

Name	Description	Nodes n	Edges m
USA	Full USA	23,947,347	58,333,344
CTR	Central USA	14,081,816	34,292,496
WST	Western USA	6,262,104	15,248,146
EST	Eastern USA	3,598,623	8,778,114
LKS	Great Lakes	2,758,119	6,885,658
CAL	California and Nevada	1,890,815	4,657,742
NE	Northeast USA	1,524,453	3,897,636
NW	Northwest USA	1,207,945	2,840,208
FLA	Florida	1,070,376	2,712,798
COL	Colorado	435,666	1,057,066
BAY	San Francisco Bay Area	321,270	800,172
NY	New York City	264,346	733,846

Table 6.1: USA road networks with number of nodes and edges.

The German company PTV AG [2] have provided European road networks available for DIMACS Challenge participants and others. The format is the same as the USA networks. Use of the road networks requires signing a license agreement stating that the use of the networks is only for scientific use. PTV provides road networks for 17 European countries and the whole of Europe. Unfortunately these networks lack the travel time attribute for edges. Universität Karlsruhe has worked with PTV to provide the networks. They have used edge categories given in the original data, and assumed average speeds for the 13 different categories, to make an additional road network for the whole of Europe with travel time on edges. From this big European road network, smaller parts can be cut out by a cutting polygon. We have coordinates for all countries, so the convex hull of all intersections could be used to cut out countries. We will use the simple approach of using a rectangle as cutting polygon. We will use country names for the road networks even though they may include large parts of neighboring countries.

Figure 6.1: The road network of Denmark and the network cut out from the big European network.

An application has been developed to cut out from the big European network, taking as parameter a cutting rectangle. Notice that cutting may not produce a *strongly connected network*. To handle this, the application finds the *maximum strongly connected component*. On average, 0.2% of the nodes were not strongly connected. Figure 6.1 shows the original road network of Denmark and the maximum strongly connected component cut from the large European network.

Notice that the network now includes Southern Sweden and the Northern tip of Germany. Table 6.2 shows the networks cut out from the big European network with the cutting application.

Name	Description	Nodes n	Edges m
EUR	Full Europe	18,010,173	42,560,279
FRA	France	5,875,289	14,048,754
DEU	Germany	5,411,065	13,222,264
ITA	Italy	2,574,398	6,024,188
GBR	Great Britain	2,285,562	5,252,190
NOR	Norway	1,925,597	4,331,668
SWE	Sweden	1,845,878	4,249,810
NLD	Netherlands	1,556,207	3,819,094
AUT	Austria	1,387,977	3,284,231
ESP	Spain	1,028,343	2,367,177
CHE	Switzerland	899,128	2,138,838
DNK	Denmark	843,310	1,949,506
BEL	Belgium	833,108	2,018,231
PRT	Portugal	180,905	427,404
LUX	Luxemburg	49,107	116,166

Table 6.2: European road networks with number of nodes and edges.

From the attributes given for the road networks we can deduce a speed attribute in km/h for all edges $(u, v) \in E$ as

$$\nu(u, v) = \frac{d(u, v)}{w(u, v)} \times 10 \times 3.6 \tag{6.1}$$

From the speed we can give each edge a category $\kappa(u, v)$ as shown in table 6.3.

$\kappa(u, v)$	$\nu(u, v) \in$
Fast Motorway	$[130; \infty[$
Motorway	$[110; 130[$
Highway	$[90; 110[$
Road	$[80; 90[$
Small Road	$[60; 80[$
Street	$[50; 60[$
Small Street	$[30; 50[$
Gravel/Forest Road	$[0; 30[$

Table 6.3: Edge category $\kappa(u, v)$ from edge speed $\nu(u, v)$

The categories may not be fully suitable for all countries, but for the purpose in this thesis, they are sufficient. We fail to capture edges being part of ferry connections. These will be categorized as gravel road or small street, depending on the speed of the ferry. We will accept this, but keep it in mind when developing methods to find edge travel time functions.

6.2.2 Delay Forecasts

From the Danish Road Directorate [34] we have obtained some measurings from a number of stations on approach roads to Copenhagen. The measurings are for a specific direction. For each road we get a number of time intervals $[t_i, t_{i+1}[$, varying between 15 minutes and one hour. For each time interval we get a number of average speed and count pairs (v_i^k, c_i^k). From this we find the average speed v_i in each interval as

$$v_i = \frac{\sum_k v_i^k c_i^k}{\sum_k c_i^k} \tag{6.2}$$

We would like to work in terms of delays, so we set the "optimal" speed to $\overline{v} = \max\{v_i\}$. Delay for time interval i can now be defined as

$$d_i = \frac{\overline{v}}{v_i} \tag{6.3}$$

Figure 6.2 shows plots for a number of roads with a few days on each plot.

Notice that because the delays come from the average speeds of vehicles, delay does not only originate from congestion but also from other reasons of reduced speed. It seems that for all roads there are even small delays during the night. This is suspected to be because the roads are used by a higher percentage of slower vehicles like trucks. Notice that for all instances high delays only occur on weekdays. Below the delays for each road are analyzed.

Road 13 is one of the most congested approach roads to Copenhagen. The measuring station is at the end of the road which is often compared to a funnel. All delays are negligible except for the very high delays in the morning. The high delay is in a three hour period from 6 to 9 in the morning. It is clear that congestion can appear and disappear rapidly. In the afternoon there is a single delay for one of the week days. It is suspected that this could be because of an accident or the like, rather than a general trend.

Road 14 is another approach road with congestion problems in the morning, but not in the same degree as road 13.

Road 3 does not suffer any significant delay in north bound direction, but a little around the end of work day. In the south bound direction there are big delays for two out of five week days. This indicates that delay can happen suddenly when the amount of traffic reaches a critical level.

Road 10 has some delay in the morning in the inbound direction to Copenhagen. In the outbound direction there are congestion situations all through the day. The delay is most consistent in the afternoon when the commuters return from work.

Road 30 is an example of a road with no significant congestion. Notice that in the morning and in the afternoon there seems to be the least delay. This indicates that commuters represent travellers going at the highest speed.

The obtained delay forecasts have given us an idea of what real forecasts look like. They will form the basis of the delay forecasts used when constructing simulated forecasts.

6.3 Simulated Forecasts

Given that we do not have enough real forecasts, we could try to simulate forecasts by modelling traffic. Traffic modelling is a big area, and outside the scope of this thesis, but we will give a brief overview of the basic concepts to enable us to make a simple practical simulation.

In traffic modelling one tries to find the amount of traffic on roads at different times. This is usually done using Origin-Destination (OD) matrices. Entry (i, j) in an OD-matrix gives the amount of traffic that needs to go from location i to location j in some time interval. There will be different OD-matrices for different time intervals, and different matrices for different kinds of traffic. This could be for commuters going to and from work; for sales men going from customer to customer; for vans picking up and delivering goods; and so forth. From the OD-matrices different statistical methods are used to estimate the amount of traffic on each road at different times. The simplest method would simply be to assume that the statically fastest path will always be used. Other methods include minimizing the sum of all travellers costs, or by minimizing the cost of each traveller, where the cost function could include different things like time, distance, road type etc.

Detailed OD-matrices require an enormous amount of data and they are difficult to obtain. Instead they are usually constructed by using a sample of travellers, or by matrix estimation. In

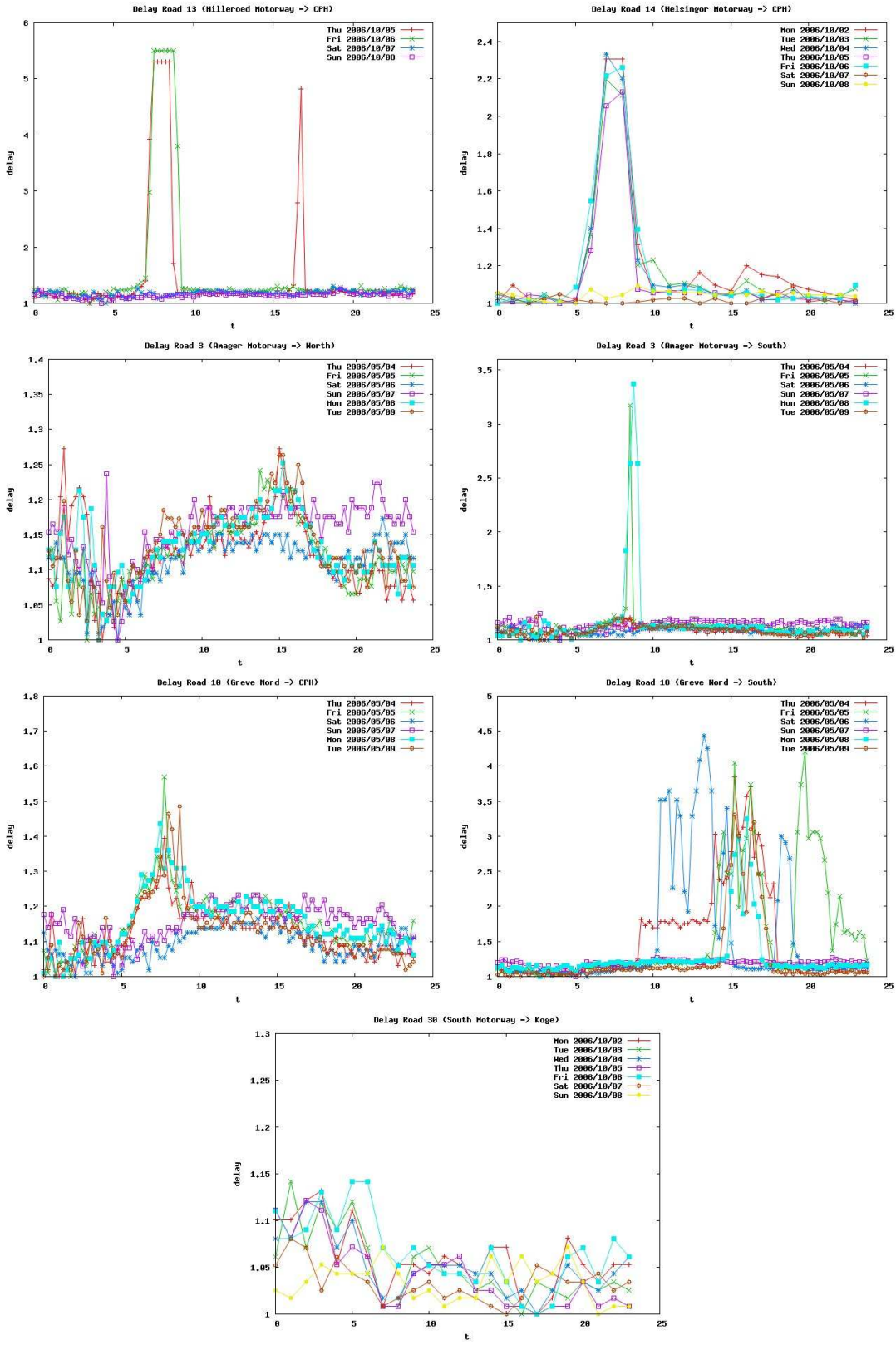


Figure 6.2: Plots of delays for a number of roads and number of days.

matrix estimation traffic counts at cuts, and maybe older information on the distribution of traffic, are used to estimate traffic.

Traffic modelling can find the amount of traffic on roads, but this does not tell if congestion will occur and how much the delay will be. Congestion has a tendency to suddenly happen when the amount of traffic reaches a critical level. Mapping the amount of traffic to delay requires a complicated function that depends on a number of factors, for example the capacity of the road. Whether congestion will occur will probably be determined by using historical data of the amount of traffic when congestion started. It would for example be possible to use the data from the previous section to see what the traffic count is when congestion occurs.

6.4 Forecasts Using Simulated Traffic

Obtaining and handling real forecasts or OD-matrices for the whole of Europe, or even whole countries, is not realistic. So in this thesis we will rely on a method to develop time-dependent networks with some touch of realism.

In this section we present a method to construct simulated OD-matrices using a series of very rough assumptions. The matrices are used to find traffic counts for each edge. Finally we use the traffic counts to assign delay forecasts to a number of edges.

6.4.1 Simulated OD-matrices

In our simulation we will only work with two OD-matrices: One for commuters going to work in the morning A, and one for commuters going home from work in the afternoon H. We assume that commuters go straight home from work so the matrices are each other transposed $H = A^T$. The obtained forecasts in Figure 6.2 seem to indicate that commuting is the main source of congestion. Furthermore is it expected that other kinds of traffic is more equally distributed over the whole day. This kind of traffic is not so important, because we will use the traffic counts for edges, found in the next subsection, in a relative way.

To make OD-matrices we need to determine the origin of commuters. We will use the rough assumption that a fixed number of commuters live in each intersection. This is a way to distribute commuters by the density of intersections. This approach may not be very accurate in reality, and notice that not all nodes in the networks are real intersections but only represent turns on a road. As we categorized edges with $\kappa(u, v)$ we can categorize nodes. We will define node category in the following way

$$\kappa(u) = \max\{\kappa(u, v) \mid (u, v) \in E\} \tag{6.4}$$

That is, the category of node u is the maximum edge category of any out-going edge. When distributing commuters we must expect that more people live in nodes with higher category. We will use the following function $\eta(u)$ to determine the number of commuters in a node u.

$$\eta(u) = \begin{cases} 2 & \kappa(u) = \textit{Gravel/Forest Road} \\ 15 & \kappa(u) = \textit{Small Street} \\ 30 & \kappa(u) = \textit{Street} \\ 15 & \kappa(u) = \textit{Small Road} \\ 10 & \kappa(u) = \textit{Road} \end{cases} \tag{6.5}$$

The reason to use this function is that it captures that most people live at streets and small roads. No one lives by roads bigger than highway, and only a few people live by small gravel roads. The specific numbers is more or less arbitrary based on the network of Denmark and Southern Sweden (DNK) shown in Figure 6.1. It is known that there is approximately 4 million passenger cars in this area. By analyzing the categories of all nodes in the network we find that the resulting number of commuters will be around 3.4 million. Similarly for other instances, the used number of commuters is somewhat proportional to the number of citizens. Again we must stress that we only try to find OD-matrices with some touch of realism.

We know the sum $\eta(u)$ of each row in the matrix A. We need to distribute the commuters of each node over the columns. That is, we need to determine destinations for the commuters. We will make the assumption, that all commuters work within a neighborhood $N(u, \lambda) \subseteq V$ defined as all nodes reachable within the time λ from the source u. We choose $\eta(u)$ destinations randomly from this set. We will use $\lambda = 45$ minutes. Statistically this means that the average travel time is $45/\sqrt{2} \approx 32$ minutes, because this is the radius that splits the area of a circle in two equally sized areas.

6.4.2 Finding Traffic Counts

We now have two simulated OD-matrices, A and H, from which we would like to get two traffic counts for each edge. The first $c_A(u, v)$ is the number of commuters using the edge on the outward trip. The second $c_H(u, v)$ is the number of commuters using the edge on the home trip.

We will make the assumption that all commuters use the statically fastest path between their origin and destination. The last assumption is that the reversed path will be used for the home trip. This path may not be the fastest, and it may not even exist if one-way streets have been used, but the assumption greatly simplifies finding traffic counts. If an edge on the reverse path does not exist, we will just ignore it. Algorithm 6.1 shows how finding the OD-matrices and traffic counts can be wrapped up into a single algorithm.

Algorithm 6.1 Simulating traffic in a road network

```
SimulateTraffic(G, λ)
1  c_A(u,v) ← 0, c_H(u,v) ← 0   ∀(u,v) ∈ E
2  for each u ∈ V
3       do (S,p) ← GrowShortestPathTree(u, λ)
4          D ← PickRandom(S, η(u))
5          for each v ∈ D
6               do for each edge (v1,v2) on shortest path p(u,v)
7                       do c_A(v1,v2) ← c_A(v1,v2) + 1
8                          c_H(v2,v1) ← c_H(v2,v1) + 1
9  return (c_A, c_H)
```

Finding the neighborhood in line 3 of the algorithm is simply done by growing a shortest-paths tree from u with Dijkstra's algorithm, and stopping when settling a node v with $\delta(u, v) > \lambda$. The neighborhood is the set of settled nodes S. When growing the shortest-paths tree a predecessor graph p is made which allows traversing the shortest path in line 6.

The running time of the algorithm is $O(n^2 \log n)$. For large networks this will be computationally infeasible. This can be handled by only using a subset of nodes, equally distributed over the network. For example by skipping a fixed number of nodes in each iteration. This reduces the traffic counts but the counts become less accurate within the model.

An alternative approach could be to select $\eta(u)$ destinations randomly from a geographical neighborhood, defined as nodes within some euclidean distance λ' from u. These could be found by first building a 2-dimensional *kd-tree* containing all nodes by their coordinates. And then make query to the tree with a square of side lengths λ'. Too many nodes would be selected, as a square instead of a circle is used, but that is unimportant within this context. Fast point-to-point query methods from Chapter 2 could now be used to determine the path to destinations. As this would require implementing one of the best speed-up techniques, the slower approach has been chosen.

Figure 6.3 shows the 3 % highest outward count edges from the DNK network. Only every 10th node was used. Finding the counts was done on a laptop with a 2 Ghz 64 bit Intel Dual Core CPU, taking approximately 80 minutes. It takes approximately the same time for each node, so for all instances it takes around 57 milliseconds to process a node. Counts were also found in 13 hours by using all nodes. The result did not differ significantly from the shown. Table 6.4 shows

the CPU time to find traffic counts for the European networks with indication of how many nodes j that are included in the generation. All but the last three was found using the laptop. The last three was found using a single core in a machine with an Intel Xeon Quad Core 2.66 GHz CPU and 8 GB RAM.

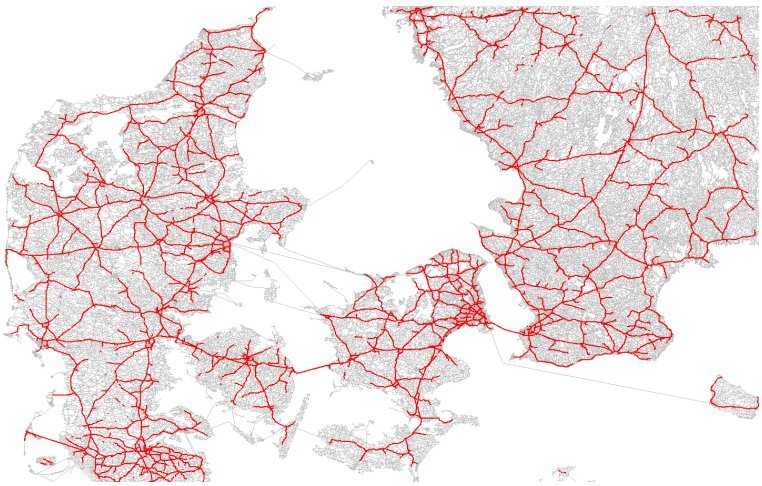


Figure 6.3: The 3 % highest outward count edges in the DNK network.

Network	Every jth node	CPU time
EUR	300	8.5 hours
FRA	40	13 hours
DEU	30	16 hours
ITA	30	4 hours
GBR	30	4 hours
NOR	30	112 minutes
SWE	30	110 minutes
NLD	25	240 minutes
AUT	20	117 minutes
ESP	15	79 minutes
CHE	15	95 minutes
DNK	1	13 hours
BEL	15	100 minutes
PRT	5	14 minutes
LUX	1	10 minutes

Table 6.4: CPU and skip count used when generating simulated traffic counts.

As can be seen from Figure 6.3, the high counts are distributed over the network by the density of street/road nodes. The number of high count edges is low in Sweden, because of the high percentage of forest roads.

Many roads with known congestion problems fall within the 3 % highest count roads, but also a lot of roads in rural areas with no congestion problems do. This shows the weakness of using a fixed number of commuters in each intersection. Notice that the high count roads found by the model are all bigger roads because the fastest path is used between origin and destination, and because a rather large neighborhood is used. Notice that the roads we have found resembles the important roads found in highway hierarchies.

6.4.3 Assigning Delay Forecasts

We now have some simulated traffic counts for each edge, we would like to use this to assign delay forecasts to each edge and thereby a travel time function. To simplify matters we will use a number of fixed delay forecasts based on the forecasts given in Figure 6.2. Figure 6.4 shows the five delay forecasts for outward and inward traffic that we will use. An additional forecast will be used with no delay at any time.

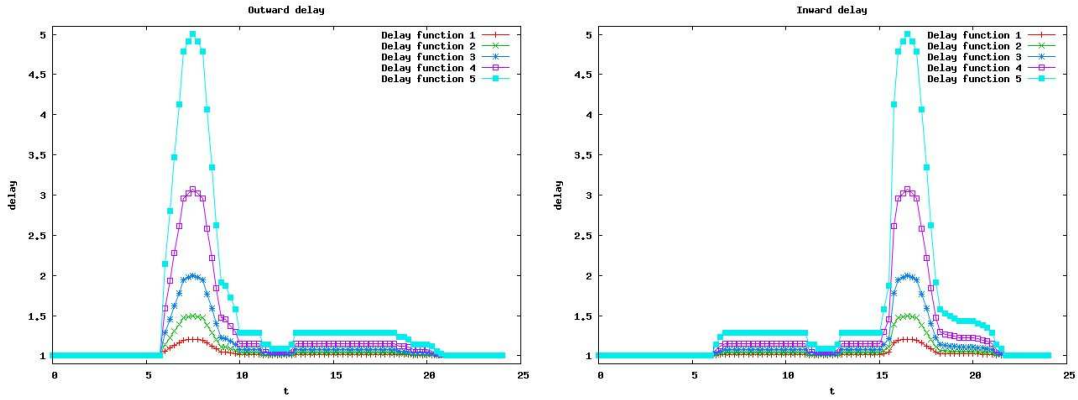


Figure 6.4: Five outward and inward delay forecasts.

The outward delays resembles the ones observed in Figure 6.2. The inward delays were less consistent, so we choose to use the same pattern as for the outward traffic. Notice that both families of delays include some delay during the day. This resembles what was noticed in Figure 6.2, and it is more fair when testing the algorithms, because otherwise the arrival time function would be over simplified. We will denote the outward and inward delay forecasts $D_1^A, \ldots, D_5^A$ and $D_1^H, \ldots, D_5^H$ respectively.

We will assign an outward delay forecasts F_{uv}^A to the 10 % highest outward count edges and an inward delay forecast F_{uv}^H to the 10 % highest inward count edges. We estimate that 10 % is a high figure compared to reality, but we will use this as a worst case example. Later we will make other levels of delay by removing delay from edges until a certain percentage remain with delay. Thereby we can make more levels of delay for each instance, which will be used for experimentation.

Figure 6.5 shows the 10 % highest count outward edges for Denmark and Southern Sweden. The green color gradient indicates the level of delay, with the darkest being the most delay. A zoom of the greater Copenhagen area and similar figures for other networks, indicating outward traffic counts, are given in Appendix A.1.

The actual assignment of delay forecasts to edges is done by the following functions.

$$F_{uv}^A = \begin{cases} D_1^A & (u,v) \text{ is in top } [0\%; 1\%[\\ D_2^A & (u,v) \text{ is in top } [1\%; 3\%[\\ D_3^A & (u,v) \text{ is in top } [3\%; 5\%[\\ D_4^A & (u,v) \text{ is in top } [5\%; 7\%[\\ D_5^A & (u,v) \text{ is in top } [7\%; 10\%[\\ \emptyset & \text{otherwise} \end{cases} \qquad F_{uv}^H = \begin{cases} D_1^H & (u,v) \text{ is in top } [0\%; 1\%[\\ D_2^H & (u,v) \text{ is in top } [1\%; 3\%[\\ D_3^H & (u,v) \text{ is in top } [3\%; 5\%[\\ D_4^H & (u,v) \text{ is in top } [5\%; 7\%[\\ D_5^H & (u,v) \text{ is in top } [7\%; 10\%[\\ \emptyset & \text{otherwise} \end{cases} \tag{6.6}$$

We do this by sorting two lists of all edges descending by $c_A(u,v)$ and $c_H(u,v)$ respectively.

For each edge with a delay forecast we construct a travel time function $F = \{(t_i, w_i)\}$ as described in section 3.6. If an edge has been assigned both inward and outward delay forecasts we need a way to merge two travel time functions. Algorithm 6.2 gives a generic way to do this merging.

The merging is done by iterating through the points in both F^1 and F^2 ordered by time. Each point will create a new point in F. If two points from F^1 and F^2 have the same time they will

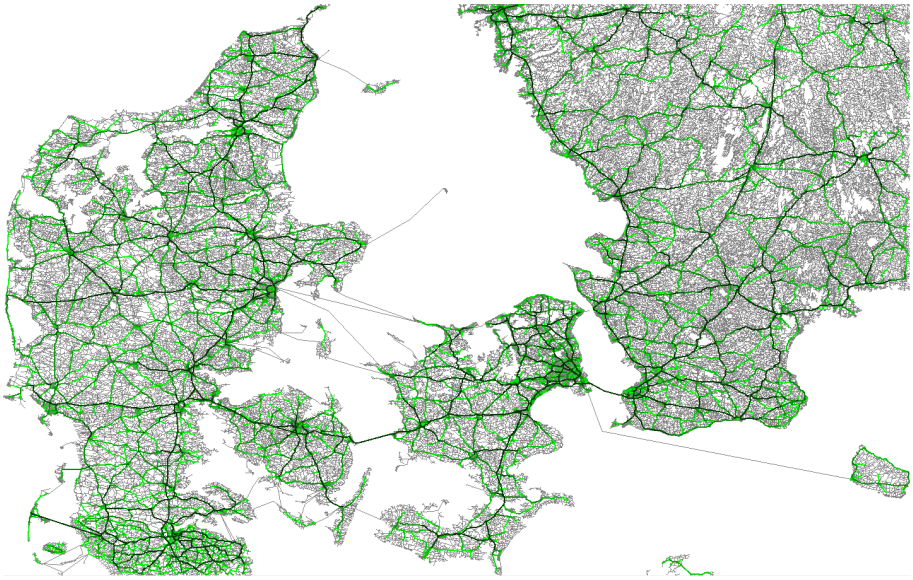


Figure 6.5: Assigned outward delay forecasts for Denmark and Southern Sweden, indicated by a gradient of green.

Algorithm 6.2 Merging Travel Time Functions

TravelTimeMerge(F^1, F^2)

```
1  F ← ∅
2  i ← 1, j ← 1
3  while i ≤ |F^1| and j ≤ |F^2|
4        do t ← min{t_i^1, t_j^2}
5           w ← max{w^1(t), w^2(t)}
6           F ← F ∪ (t, w)
7           if t_i^1 ≤ t_j^2
8              then i ← i + 1
9           if t_i^1 ≥ t_j^2
10             then j ← j + 1
11 return F
```

only create one new point in F. The maximum travel time for F^1 and F^2 is used as travel time for new points. Notice that the algorithm requires that the last point in both F^1 and F^2 have the same time. The number of linear pieces in the new travel time function will be bounded by $|F| \leq |F^1| + |F^2|$.

Most shortest-paths found in the networks will be subject to delay, because important edges will be used. In reality delay is quite rare and much more restricted to certain areas, so the simulation gives a somewhat worst case scenario for algorithms trying to speed-up time-dependent shortest-paths queries. This should be taken into account when evaluating experimental results for the algorithms.

When assigning delay forecasts to edges, edge category could be taken into account. Higher category edges have larger capacity and therefore gets congested at higher traffic counts than lower category edges. No success was achieved by incorporating category into assigning delay. The tendency was that either bigger roads was assigned large delay or none at all. More elaborate techniques are needed, which seems difficult to apply without more data on the road networks. We will use the simple approach described above.

6.4.4 Different Levels of Delay

For experimentation we would like to have more levels of delay than just the single one we have found so far. Specifically we would like for each instance to have levels of delay so 2, 5 and 10 percent of the edges have delay. We obtain such levels of delay simply by randomly removing delay from edges for the set obtained above. This way we actually have four time-dependent instances for each static instance INST. We will denote the instances INST(d) with $d = 0, \ldots, 3$ where $d = 0$ is the static instance with constant travel time on all edges, and $d = 1$ corresponds to delay on 2 percent of the edges, and so forth.

6.5 Space Complexity Issues

Representing large road networks requires a considerable amount of space. When we work with time-dependent networks an additional amount of space is required to store travel time functions for edges. In this thesis we will store whole networks in main memory, not partly on disk, so we need to consider how to limit memory usage. In this section we analyze memory usage and ways to reduce it.

6.5.1 Travel Time Pruning

A graph representation of the core network is described in Chapter 7. This representation requires a fixed amount of space that we cannot hope to improve considerably. Introducing travel time functions gives an additional memory usage that may impede practical usage of time-dependent networks.

Consider using a travel time function F_{uv} for each edge $(u, v) \in E$ with a point for every 15 minutes. Assuming that time is represented by 32 bit (4 bytes) integers, then each travel time function will require $24 \times 4 \times 2 \times 4 = 768$ bytes. For the DNK network this requires approximately 1.5 GB and for the whole European network EUR approximately 33 GB. This is clearly unacceptable.

We handle edges with no delay by an empty travel time function not taking any additional space. The instance with most delay has a travel time function on 10 % of the edges. This reduces the required amount of memory for the two networks above to approximately 154 MB and 3.3 GB respectively. This may still be unacceptable.

Notice that when the travel time function does not change much, for example in the night, using a point for every 15 minutes may be excessive. Algorithm 6.3 shows how to prune points when the slope of two neighboring linear pieces is below a threshold ρ.

Algorithm 6.3 Travel Time Pruning

```
TravelTimePruning(F, ρ)
1  while |F| > 2
2        do i ← argmin {|Δ_i^w − Δ_{i+1}^w| : 1 ≤ i ≤ |F| − 2}
3           if |Δ_i^w − Δ_{i+1}^w| < ρ
4              then F ← F\(t_{i+1}, w_{i+1})
5              else return F
6  return F
```

When we use threshold $\rho = 10^{-9}$, analysis shows that the average number of points in the travel time functions is reduced from 96 to 42. Notice that because of the very low ρ, only points between linear pieces with practically the same slope are removed. This is the points in the night and during periods of the day between rush-hours. The additional memory usage of the two networks are now reduced further to approximately 67 MB and 1.4 GB respectively. Hopefully this is acceptable.

Notice that excessive pruning declines the accuracy of the travel time functions, and may also result in non-FIFO travel time functions. Further notice that pruning does not only reduce memory consumption, but also reduces the time it takes to calculate travel time, because of reduced lookup time.

If it becomes impossible to store travel time functions in memory, because they are defined for a high percentage of edges and/or because the level of detail is high, other methods are needed. This could be storing travel time functions on disk, for example in a database able to cache the latest requests, or the like. This would greatly increase the solution time of the algorithms, and is not a subject that will be handled further in this thesis.

Chapter 7

Implementation

In this chapter we describe the main aspects and considerations concerning the implementation. The implementation is object oriented and written in C++ using the *Standard Template Library* (STL) and the *Boost Graph Library* (BGL). The source code can be downloaded at

`http://www.diku.dk/hjemmesider/studerende/oleborup/tdsp/tdsp.tar.bz2`

and the documentation browsed at

`http://www.diku.dk/hjemmesider/studerende/oleborup/tdsp/`

The amount of instance data is substantial, and therefore not included. Furthermore the European instances are not to be made publicly available.

7.1 Boost Graph Library

When designing graph algorithms one have a choice between using an existing representation or making a new one from scratch. Making a new one allows a representation that is tailored towards the specific needs of the algorithms, and it gives complete control over the graph structure, which allows tuning, for example of memory usage. But it requires substantially more work and debugging, and no existing algorithms can be used.

We have chosen to use the Boost Graph Library [29, 40]. It is a highly generic library that gives easy graph traversal of nodes and edges, and has the strong concept of node and edge *properties*. The concept of properties allows adding different attributes to nodes and edges. As default, weighted graphs have an edge weight property, but any property can be added to edges. In our case for example, the property of arrival time functions on edges. A graph class can be constructed with template arguments that control the type of graph, weighted and directed, properties, and also some of the internal representation of nodes and edges. We have chosen to use a bidirected adjacency list representation, which means that both out- and ingoing edges of nodes can be traversed. The graph representation we use has been tuned towards fast access to nodes and edges, instead of faster insertion and deletion, as we load the graph once, without changing the node and edge set later.

Boost has a set of ready made graph algorithms, including an implementation of Dijkstra's algorithm. We have chosen to make our own implementation, as it was difficult to adjust the Boost version towards our special needs, and an experimental evaluation showed that our version was two times faster. The only algorithm we have used from the library is one that finds the maximum strongly connected component of a graph.

Using Boost has proven to give simple and efficient access to the graph when implementing algorithms. But one major drawback was encountered when testing algorithms on larger graph instances. The memory usage, without any added edge properties, is considerable compared to the amount of memory that is needed at a minimum to represent the graph. Two causes has been identified for the memory usage. The adjacency list representation uses STL container `vector` to store both node and edge lists. For each node two lists are used, one for outgoing, and one for

ingoing edges. Compared to a simple array, an instance of `vector` stores additional data controlling the object. The other main cause is the property map of the graph. How it is represented could not be determined, but it uses quite a lot of memory for large node and edge sets.

The memory usage, without edge arrival time functions, is approximately 100 bytes per node and edge for all available instances. For the European network this results in a memory usage of approximately 6 GB. Adding arrival time functions to edges and preprocessed data like landmark and cluster distances, makes it impossible to use the instance on the available hardware. Given that the subject of the thesis was solving time-dependent shortest-paths problems on large road networks, this discovery has been disappointing. In the next chapter experimentation is described. A largest possible subgraph has been extracted from the European network and used instead. It is around 61 percent the size of the whole European network.

Performance problems when using existing graph libraries is not new. In a lately discovered article by Sanders and Schultes [38] on engineering fast route planning algorithms, they conclude that when implementing for large networks, implementing a tailored graph representation is adviceable due to performance problems in existing libraries when handling millions of nodes and edges.

7.2 Dijkstra Implementation

Dijkstra's shortest-path algorithm is the basis for all quering algorithms, and a key ingredient in all preprocessing algorithms. Therefore effort has been put into finding the most efficient practical implementation. Algorithm 7.1 shows the practical implementation that gave the best results.

Algorithm 7.1 Practical Dijkstra implementation

```
DIJKSTRA(G, s, t)
 1  for each node v ∈ V
 2       do color[v] ← WHITE
 3  Q ← ∅
 4  color[s] ← GRAY
 5  d[s] ← t
 6  p[s] ← s
 7  INSERT(Q, s)
 8  while Q ≠ ∅
 9       do u ← EXTRACTMIN(Q)
10          for each v ∈ adjacent(u)
11               do if color[v] = WHITE ∨ w(u, v) + d[u] < d[v]
12                    then d[v] ← w(u, v) + d[u]
13                         p[v] ← u
14                         if color[v] = WHITE
15                            then color[v] ← GRAY
16                                 INSERT(Q, v)
17                            else if color[v] = GRAY
18                                    then DECREASEKEY(Q, v, d[v])
19          color[u] ← BLACK
```

Three different labels are used, distance d, predecessor p and color. The color indicates undiscovered, discovered and settled, for white, gray and black, respectively. First of all, notice that nodes are only added to the heap when they are discovered the first time. This makes heap operations more effective, but requires a check of color to see if an insert or decrease key operation should be used when relaxing. Further notice, that we only initialize the color in line 1, which is possible because we always relax undiscovered nodes in line 11, without considering distance. A simple binary heap is used, that has access directly to the distance labels and uses this to

determine the current tentative distances of nodes.

All the variations of Dijkstra's algorithm we use build on this implementation. In quering the the search is stopped when the target node is encountered, and pruning may be performed. Rank ordering may be determined, the last settled node saved or the algorithm restarted with some alternative initialization, for example when finding oversampled PCD clusters.

In quering we need to restart the algorithm consistently, therefore we need effective reinitialization. As described we only need to initialize the color label. One approach is to store all discovered nodes on a stack or the like, and then before a query only initialize nodes with changed color. Practical experiments surprisingly showed, that it was actually more effective to iterate through all color labels, which is represented as a simple array, as nodes are represented by integers in the range $\{0, \ldots, n-1\}$. This is suspected to be because the compiler does loop-unrolling, which makes such an iteration very effective.

7.3 Classes

In this section we describe the main classes of the implementation.

7.3.1 Time-Dependent Graph

The class `TimeDepGraph` represents a time-dependent graph. It inherits from the Boost adjacency list graph, templated to meet our needs. Nodes have a coordinate property and properties for edges include weight, length and arrival time function. The weight is the most optimistic static travel time, and length is included to find edge category $\kappa(u, v)$. Edges are represented as a pair of integers representing source and target node. When using the graph, only the necessary parts are loaded. So for normal quering only edge weight and arrival time functions are loaded. Under construction of delay forecasts, and when cutting from bigger networks, the rest of the properties are loaded.

The graph class extends the functionality of the base Boost class by a number of helper methods to ease the use of the graph. It includes, among others, methods to load and save a graph from and to disk, find upper bounds on travel time, finding arrival time for an edge given departure time, and easy access to properties.

Arrival time functions for edges are represented by a pointer to an instance of the class `ArrivalTime`. When an edge has no delay the pointer is null. Notice that each edge points to a unique arrival time instance, not just some static delay forecasts, converted to arrival time for a specific edge at query time. That was considered not to accurately describe the intended use with real unique forecasts for each edge. The arrival time class is basically an ordered collection of forecast points (t_i, w_i). Quering for arrival time finds the linear piece to use and calculates the travel time. The class also includes a method to find the maximum travel time, method to determine if the function is FIFO, and a method to make the function FIFO.

Arrival time functions for edges are represented in a file for each graph and for each of the three levels of delay. The delay is represented with number $0, \ldots, 4$ for edges with delay indicating one of the four fixed delay functions. When loading forecasts the static delay functions are converted to an arrival time function for each edge.

7.3.2 Solvers

All solver algorithms are represented by solver classes. They all inherit from the abstract class `TdspSolver`, which defines pure abstract methods `solve(src,tgt,t)` and `preprocess()`. All children must implement these methods. For static and time-dependent Dijkstra the preprocessing is just an empty method. The abstract class have methods common to all solvers, for example time and settle count measuring, and a method to dump the solution to a visualizer file, described shortly.

The solver classes have so much in common that a hierarchy of solvers is made, where they inherit common features from each other. The labels to be used are defined in the static Dijkstra solver `StaticDjkSolver` which all other solvers inherit from. This class gives methods to initialize and clean up before a new query. Figure 7.1 shows the hierarchy.

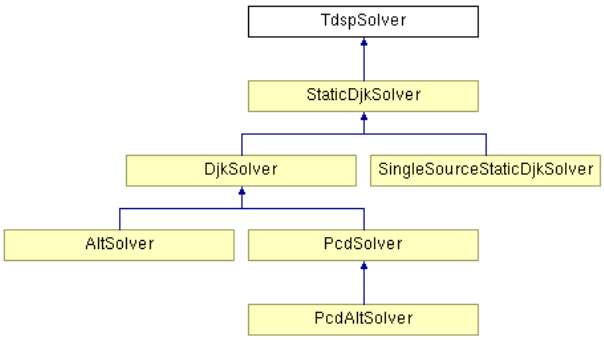


Figure 7.1: Hierarchy of TDSP solver classes.

Having all solvers inherit from `TdspSolver` allows making instances of all solvers as an instance of `TdspSolver`. Quering is now calling the preprocess method once, followed by multiple solve calls, no matter what actual solver is used.

For PCD a clustering is found using the class `PcdClustering`. The class has methods to determine cluster labels, distances, and bounds used in PCD. Instead of doing the preprocessing each time the PCD solver is used, the clustering and other information are stored on disk for the given instance, number of clusters, and delay level.

For ALT, landmarks are represented by the class `Landmarks`. The class has methods to find landmarks and distances, and also finding lower bounds given source and target node.

7.4 Applications

A number of applications, shown in Table 7.1, have been implemented to solve, construct and test the algorithms.

`tdspconvert`	Cut subgraphs from bigger time-dependent graphs using a cutting polygon and save the new time-dependent graph.
`tdsppruner`	Find maximum strongly connected component of a time-dependent graph and save it as a new time-dependent graph.
`tdspgenerator`	Generate time-dependent graphs using simulated traffic.
`tdspsolver`	Solve a time-dependent shortest-path query given various arguments like algorithm, source- and target node, and starttime.
`tdspcorrectness`	Test the correctness of solvers by solving the same set of problems and comparing results.
`tdsptest`	Test performance of different solvers on different instances, and store the result for further processing.

Table 7.1: List of implemented applications.

Other implementations include a number of Perl scripts for processing various data.

7.5 Visualizer

When developing and testing algorithms a visualization tool is indispensable for verification and understanding the algorithms behaviour. A visualizer application has been developed in Java which was considered easier than in C++. Furthermore a similar visualizer existed from a previous project and could be adapted. The visualizer loads a graph into a simple graph like datastructure and displays the graph based on the node coordinates. The graph can be zoomed and centered using the mouse buttons, and node identifiers can be shown.

Using Java disallows the C++ code from easily calling the visualizer. Instead a decorator file format has been developed. An optional decorator file argument can be given to the visualizer, if given, the visualizer will load the decoration after loading the graph. When showing the graph, the decoration can be turned on and off. The decorator file format is very simple, each line can be one of the following.

- `path` *color* v_1 v_2 $\cdots$ v_k
- `vertex`D *color* v_1 v_2 $\cdots$ v_k

A sequence of nodes is given in both cases, color is given in hexadecimal form as also used in HTML. For nodes, D can be specified as the diameter of the circle made to indicate the node. In the implementation decorator files can be created to show specific properties, for example a clustering made for PCD. Figure 7.2 shows the visualizer application with a decorator file for a clustering of North-West Europe. All figures in the thesis showing road networks have been made with the visualizer.

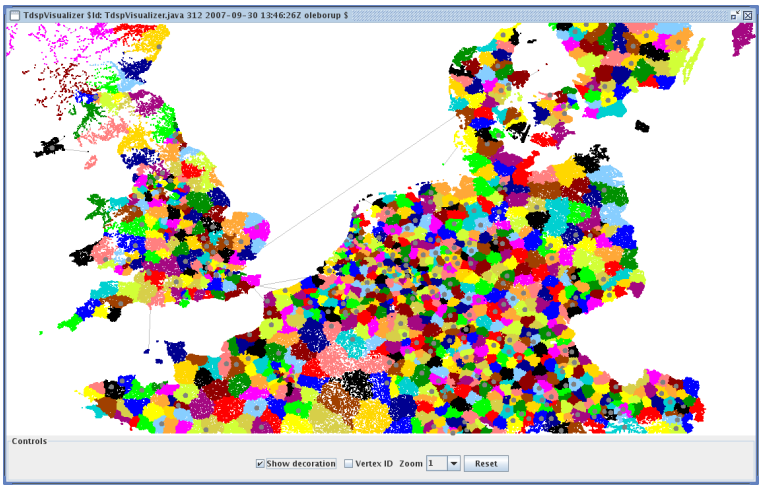


Figure 7.2: Screendump of the visualizer application showing a clustering of North-West Europe.

For large graphs memory consumption becomes a problem, so in these cases only a subset of nodes and edges are shown. Loading graphs from disk are slow, but rendering the graphics is very fast, under one second.

7.6 Parallelization

No parallelization has been used in the implementation, but it is applicable for the preprocessing stages of ALT and PCD. Finding cluster distances is by far the most demanding task in preprocessing of PCD. Finding the different distances is independent, so this task could be done in parallel. The same applies for ALT, but here the preprocessing stage is much less demanding. Quering cannot be made in parallel because the search is unidirectional. In PCD the searches

in stage two could be made in parallel, but it is uncertain if this would be beneficial given the overhead of starting new threads.

Chapter 8

Experiments

In this chapter we present results from an experimental evaluation of the developed algorithms. First we give some prerequisites before presenting a correctness evaluation and key performance results for each algorithm.

8.1 Prerequisites

In this section we describe the instances, algorithms and environment used in the experiments. Furthermore we describe how performance testing is performed.

8.1.1 Instances

We will limit experimentation to the four instances in Table 8.1.

Name	Description	Nodes n	Edges m
MEU(l)	North-West Europe	10,862,697	26,000,041
DEU(l)	Germany	5,411,065	13,222,264
NLD(l)	Netherlands	1,556,207	3,819,094
DNK(l)	Denmark	843,310	1,949,506

Table 8.1: Instances used in experimentation.

Recall that for each instance we have four levels of delay $l = 0, \ldots, 3$, so in practice we have 16 time-dependent instances. Due to the size limitations described in the previous chapter we cannot use the full European network. Instead we have used the cutting application to find the largest possible part of Europe that we can solve. Figure A.4 in Appendix A.1 shows instance **MEU**(3).

When loading delay forecasts and assigning arrival time functions to edges, it is determined if the functions are FIFO. For all instances only two edges in **MEU(3)** were not. They are both ferry connections with travel time over one our.

8.1.2 Algorithms

The algorithms we have available are summarized below.

DJK The time-dependent Dijkstra algorithm (Algorithm 3.3).

ALT A* search (Algorithm 4.1) with 16 landmarks found as described in section 4.3.2.

PCD(k) Precomputed cluster distances algorithm (Chapter 5) using k clusters.

PCDALT Precomputed cluster distances combined with ALT. The number of clusters used is always 1024.

8.1.3 Environment

All experiments were performed on a 64-bit machine with 8 GB main memory and 4 MB L2 cache, using one out of 4 cores on an Intel Xeon Quad-Core processor clocked at 2.66 GHz, running Ubuntu Linux (kernel 2.6.22). The programs were compiled by the GNU C++ compiler 4.1.3 using optimization level 3. Version 1.33.1 of the Boost library was used.

8.1.4 Performance Test Method

We evaluate performance in terms of query time. The simplest way to test would be to perform a series of (s, τ, t) queries on an instance, where all variables are selected at random. This is not a very informative measure, as most queries will generate relatively long shortest-paths. It is unrealistic to assume that real life queries will be uniformly distributed. Instead we take the same approach as used by Sanders and Schultes in [35]. The solution time is measured in intervals based on the Dijkstra rank of the target node, so local queries are measured within big networks. More specifically, we choose a set of sample sources S and for each source $s \in S$ find the targets with rank as a power of two $r = 2^k$. Algorithm 8.1 shows the method, given a specific instance.

Algorithm 8.1 Performance testing

```
PerformaceTest(I, M)
1  K ← ⌊log2 n⌋
2  for i ← 1 to M
3       do s ← RandomNode(V)
4          t ← Random()
5          GrowShortestPathTree(G, s)
6          for k ← K downto K − I
7               do r ← 2^k
8                  Solve(s, v_r, t)
```

For each rank level the algorithm performs M measurements. The variable I determines how many rank intervals will be used, from 2^{K-I} to 2^K, where K is the maximum power two rank level for a given instance. A shortest-path tree is grown from s so the rank r node v_r can easily be found as the r'th settled node. The departure time t is selected at random, so all possible departure times within a whole day is equally likely.

When doing performance testing departure time plays a crucial role. Some departure times, and shortest-paths lengths, mean that no delay will occur on the path. For example for a query at midnight to a node that is reachable before delay starts at 7 in the morning. We use random departure times, but record minimum, maximum and average solution times. This way we capture very long and very short solution times, and base the performance of the algorithm on these figures. When using a number of measurements large enough, it is very unlikely not to hit departure times with high and low probability of delay. For example for an hour around midnight. The probability not to choose a departure time within this hour is $(23/24)^M$. For $M = 300$ this approximately 0.003 percent.

Solution time is the key performance figure, but we will also record the minimum, maximum and average number of settled nodes. A speed-up compared to Dijkstra's algorithm can be found both in terms of solution time and settled nodes. The difference in these two speed-ups is interesting, because they tell how much the added complexity in a speed-up algorithm matters. Solution time is measured as CPU time with `getrusage()` from `sys/resource.h`. The CPU time does not differ significantly from actual time when no other processes use the same core.

8.2 Correctness

To investigate the correctness of the algorithms the following test has been performed. For a given instance we generate a random sequence of 1000 source, target and departure time triples (s_i, τ_i, t_i). Each of the time-dependent solvers are used to solve each triple, and it is verified that they all give the same result. The test was performed with success on the following instances: **DNK**(l), **NLD**(l) and **DEU**(l) for $l = 0, \ldots, 3$. The test is not a proof of correctness, but sufficient to indicate correctness.

8.3 Dijkstra's Algorithm

In this section we test the performance of algorithm DJK on the chosen instances. The results will be used as a basis for testing the other algorithms, as their performance will be expressed in terms of speed-up compared to DJK. Table 8.2 shows the result for all instances. For each instance, and level of delay l, the performance is expressed as average/maximum solution time in milliseconds, and average number of settled nodes.

	r	$l=0$		$l=1$		$l=2$		$l=3$	
DNK	2^{13}	5/8	8192	5/12	8409	6/16	8455	7/32	9042
	2^{14}	10/12	16384	11/28	16703	12/32	17077	13/48	17311
	2^{15}	20/28	32768	21/48	33100	23/60	33875	26/92	34664
	2^{16}	43/56	65536	44/76	66101	47/92	66811	52/164	68103
	2^{17}	91/112	131072	93/176	130793	98/260	133571	108/384	135866
	2^{18}	194/228	262144	197/280	262520	202/416	261944	222/460	267178
	2^{19}	392/440	524288	397/552	524993	409/632	521238	438/688	519576
NLD	2^{13}	7/8	8192	7/12	8354	8/16	8599	9/32	8745
	2^{14}	11/12	16384	12/20	16584	13/28	16992	15/52	17527
	2^{15}	21/24	32768	22/40	32968	24/64	33242	28/136	34472
	2^{16}	42/48	65536	45/104	67181	50/312	67942	56/352	68547
	2^{17}	88/100	131072	92/216	131369	103/244	135079	112/308	132712
	2^{18}	190/220	262144	201/436	265255	215/408	263194	238/588	263386
	2^{19}	413/472	524288	427/664	522177	460/852	524181	505/1020	524990
	2^{20}	890/1020	1048576	918/1160	1044971	987/1320	1047416	1078/1552	1049271
DEU	2^{13}	15/20	8192	14/20	8245	15/24	8473	16/24	8593
	2^{14}	19/28	16384	19/36	16667	20/36	17042	21/32	16777
	2^{15}	28/32	32768	29/48	33361	31/72	33996	32/84	34390
	2^{16}	49/60	65536	49/76	65384	53/104	66177	55/140	66285
	2^{17}	94/112	131072	95/172	132646	104/244	135660	108/236	134384
	2^{18}	193/212	262144	195/352	262688	207/488	262614	219/480	264701
	2^{19}	412/456	524288	417/580	523665	443/776	526578	466/868	529348
	2^{20}	898/1004	1048576	923/1336	1055008	955/1904	1045143	1008/1760	1056196
	2^{21}	1968/2264	2097152	2006/2596	2096683	2092/3276	2101279	2172/3168	2097562
	2^{22}	4198/4820	4194304	4266/5180	4190735	4435/5192	4193497	4603/5360	4185807
MEU	2^{13}	26/32	8192	26/36	8383	26/32	8488	27/40	8555
	2^{14}	31/36	16384	30/44	16704	31/44	16736	33/84	17326
	2^{15}	41/48	32768	40/56	33226	42/68	33101	45/88	33912
	2^{16}	62/76	65536	62/84	65968	65/104	66437	72/236	67387
	2^{17}	109/132	131072	109/160	130311	118/204	134048	129/488	135298
	2^{18}	213/268	262144	214/272	262222	230/360	267058	249/484	265204
	2^{19}	441/540	524288	442/636	524160	474/712	530781	516/1064	532430
	2^{20}	943/1132	1048576	948/1600	1051534	1009/1536	1062765	1086/2464	1061489
	2^{21}	2047/2520	2097152	2044/2956	2094025	2134/3228	2096982	2299/3896	2095932
	2^{22}	4481/5464	4194304	4473/5292	4180546	4632/5936	4200627	4996/7120	4202669
	2^{23}	9240/10620	8388608	9197/9953	8389517	9541/10400	8396605	10258/11193	8392626

Table 8.2: Average and maximum solution times in milliseconds, and average number of settled nodes for DJK.

For small rank values the solution times are acceptable, under one second, but for the larger instances and larger rank values the solution times become substantial. No bigger difference between average and worst solution times are seen for delay $l = 0$.

As delay is imposed, the solution times change, as solving now requires actually evaluating arrival time functions. The average solution times seem consistently to decline with around 11

percent from $l = 0$ to $l = 3$. As random departure times are used, many queries will actually not suffer from delay, so the worst case solution times are a better figure to evaluate running time, when delay is imposed. The decline in running time is up to around 50 percent between worst running times for $l = 0$ and worst running times for $l = 3$. This seems to be an acceptable raise, as arrival time evaluation is used for a large number of edges. Recall that important edges suffer from the most delay, so longer shortest paths will contain a large number of edges with delay.

The number of settled nodes is naturally 2^k for $l = 0$, but changes as delay is imposed. The rank ordering was determined in $\underline{G}$, and naturally it changes when the network is perturbed at different departure times. The change is minor for all instances and levels of delay.

The longest solution times for the biggest instance **MEU** are over ten seconds. This is unacceptable for interactive use, so we now turn to the developed speed-up algorithms to see how much they can improve the solution time.

8.4 ALT

8.4.1 Preprocessing

Preprocessing for ALT is growing $2k + 3k/4$ complete shortest-path trees to find landmarks, and $2k$ complete trees to find shortest-path distances to and from landmarks. For $k = 16$ this is 76 tree computations. Table 8.3 shows the preprocessing times in seconds. Notice that as lower bounds are found, the bounds are the same for all levels of delay.

DNK	46	**NLD**	95	**DEU**	355	**MEU**	789

Table 8.3: Preprocessing times in seconds for ALT.

8.4.2 Quering

Measuring performance of the speed-up algorithms is done in terms of speed-up compared to the average solution times and average number of settled nodes of DJK at the same level of delay. Table 8.4 shows the results for ALT.

Before evaluating the results we will give a description of the table format. For each instance and each level of delay, the first column is the minimum solution time and minimum settled nodes speed-ups. The second column gives the average solution time and average settled nodes speed-ups. Finally the third column gives the maximum solution time and maximum settled nodes speed-ups. The speed-ups are rounded to whole figures, as this is considered to give a sufficient level of detail. Some solution time speed-ups could not be measured, as the solution times are zero or very close to zero, these are indicated with a $-$. The solution time speed-ups are generally quite imprecise for very low solution times, which can result in a large difference between solution time speed-up and settled nodes speed-up. Notice that low rank levels indicate quite short shortest paths, that are fast even for DJK, so the importance of speed-up for these levels are not as consistent and important as for higher rank levels.

The method ALT is only dependent on lower bounds, so if a query is at a time with low or no delay, the speed-up will be the same as for the static network. Figure A.6 and A.7 in Appendix A.2 show how the number of settled nodes is affected by level of delay and departure time. For **NLD**(3) and departure time midnight, the number of settled nodes is the same as for **NLD**(0). For departure time 08:00 the number of settled nodes grow with the level of delay, as the quality of the lower bounds worsens.

We now evaluate the speed-up results one delay level at a time.

l = 0 The first noticeable result is the extremely high speed-ups for minimum solution time and minimum settled nodes. For all rank levels there are speed-ups over 100 and up to thousands

	r	$l=0$ min	$l=0$ avg	$l=0$ max	$l=1$ min	$l=1$ avg	$l=1$ max	$l=2$ min	$l=2$ avg	$l=2$ max	$l=3$ min	$l=3$ avg	$l=3$ max
DNK	2^{13}	– 167	**2 6**	1 2	– 183	**2 5**	1 2	– 59	**2 4**	1 1	– 91	**2 4**	1 1
	2^{14}	– 321	**3 6**	1 2	– 143	**3 5**	1 1	– 145	**3 5**	1 1	– 182	**3 4**	1 1
	2^{15}	– 250	**5 7**	1 2	– 133	**4 6**	1 2	– 87	**3 5**	1 1	– 124	**3 4**	1 1
	2^{16}	– 377	**6 8**	2 2	– 369	**4 6**	1 2	– 185	**4 5**	1 1	– 234	**3 4**	1 1
	2^{17}	– 565	**8 9**	2 2	23 115	**5 7**	1 2	– 270	**4 5**	1 1	– 333	**3 4**	1 1
	2^{18}	– 967	**11 12**	3 3	49 262	**5 7**	1 2	50 192	**4 6**	1 1	56 979	**3 4**	1 1
	2^{19}	– 1331	**10 10**	3 3	99 1076	**5 6**	1 2	102 875	**3 5**	1 1	– 1319	**3 4**	1 1
NLD	2^{13}	– 390	**2 6**	1 2	– 116	**2 6**	1 2	– 200	**2 5**	1 1	– 115	**2 4**	1 1
	2^{14}	– 153	**2 7**	1 2	– 154	**2 7**	1 2	– 156	**3 5**	1 1	– 325	**2 5**	1 1
	2^{15}	– 655	**4 9**	1 2	– 122	**4 7**	1 2	6 244	**3 5**	1 1	– 331	**3 5**	1 1
	2^{16}	10 195	**5 12**	2 3	11 574	**5 9**	2 3	12 324	**4 7**	1 2	– 126	**4 6**	1 1
	2^{17}	– 767	**8 13**	2 3	23 238	**7 9**	2 2	– 580	**5 7**	1 1	28 275	**4 6**	1 1
	2^{18}	48 1101	**11 16**	3 3	50 524	**8 11**	2 2	54 770	**6 7**	1 1	– 1558	**5 7**	1 1
	2^{19}	103 2009	**15 17**	3 4	107 533	**9 11**	1 2	115 607	**6 7**	1 2	126 222	**5 6**	1 1
	2^{20}	– 3318	**17 18**	4 4	230 1333	**9 11**	2 2	247 4457	**5 6**	1 1	270 957	**5 6**	1 1
DEU	2^{13}	2 178	**1 6**	1 2	2 275	**1 5**	1 1	2 51	**1 5**	1 1	1 94	**1 4**	1 1
	2^{14}	2 159	**1 7**	1 2	2 132	**1 6**	1 1	2 198	**1 5**	1 1	2 146	**1 4**	1 1
	2^{15}	2 254	**2 7**	1 2	4 269	**2 6**	1 2	3 119	**2 5**	1 2	3 72	**2 4**	1 1
	2^{16}	4 282	**3 9**	1 2	4 442	**3 7**	2 2	4 49	**3 6**	1 2	5 264	**2 4**	1 1
	2^{17}	8 514	**5 10**	2 3	8 518	**4 8**	1 2	9 264	**4 7**	1 2	9 140	**3 5**	1 1
	2^{18}	16 319	**7 12**	3 4	16 440	**5 9**	2 2	17 313	**4 6**	1 2	14 137	**3 5**	1 1
	2^{19}	52 3084	**11 14**	3 3	35 172	**7 9**	2 2	37 673	**6 7**	1 2	39 735	**4 5**	1 1
	2^{20}	75 2072	**15 16**	3 4	77 1241	**9 10**	2 3	80 1197	**6 8**	1 1	50 150	**4 6**	1 1
	2^{21}	164 3848	**19 17**	5 5	125 722	**9 10**	2 2	262 5360	**7 8**	1 2	136 443	**5 6**	1 1
	2^{22}	350 2297	**20 16**	5 5	267 1110	**8 9**	1 2	370 5390	**6 7**	1 1	288 2068	**4 5**	1 1
MEU	2^{13}	1 137	**1 5**	1 1	1 83	**1 5**	1 1	1 78	**1 4**	1 1	2 87	**1 3**	1 1
	2^{14}	2 303	**1 5**	1 2	2 211	**1 5**	1 1	2 68	**1 4**	1 1	2 193	**1 4**	1 1
	2^{15}	2 209	**2 6**	1 1	2 200	**2 5**	1 2	2 261	**2 5**	1 1	2 271	**2 4**	1 1
	2^{16}	3 993	**2 6**	1 1	3 165	**2 6**	1 1	3 373	**2 5**	1 1	4 253	**2 5**	1 1
	2^{17}	5 612	**3 7**	1 1	5 437	**3 6**	1 1	6 428	**3 5**	1 1	5 189	**3 4**	1 1
	2^{18}	11 926	**4 8**	1 2	11 120	**4 7**	1 2	10 224	**4 6**	1 1	12 319	**3 5**	1 1
	2^{19}	18 384	**6 9**	1 2	22 1219	**5 7**	1 2	20 409	**4 6**	1 2	26 1439	**4 5**	1 1
	2^{20}	39 568	**8 9**	2 3	34 146	**6 8**	1 2	50 1839	**5 6**	1 2	45 671	**4 5**	1 1
	2^{21}	85 1539	**12 12**	2 2	85 547	**8 9**	2 2	107 2770	**6 7**	1 1	96 786	**4 5**	1 1
	2^{22}	224 1755	**15 12**	3 3	140 413	**8 8**	2 2	105 159	**6 7**	1 2	114 146	**4 5**	1 2
	2^{23}	462 10243	**18 13**	5 3	383 2686	**6 7**	1 2	159 221	**4 5**	1 2	285 661	**3 4**	1 1

Table 8.4: Quering speed-ups for ALT.

with regard to the number of settled nodes. The speed-up in solution time is hard to measure for these queries, because they are so fast that query time approach zero, resulting in imprecise measurements. The reason that very high speed-ups exists, is because the speed-up depends on the placement of source and target with regard to landmarks. If either source or target is a landmark, the lower bound is the actual shortest-path distance. Also if source and target are placed near or in front of landmarks, the lower bounds will be of very good quality.

The average speed-ups grow to around 20 for increasing rank level. The speed-up for low rank levels is less, because the overhead of initialization and quering for lower bounds dominate. This can also be seen as the speed-up in settled nodes is higher than the speed-up in solution time. For all instances the average speed-ups results in queries that are faster than one second.

The worst case speed-ups are quite disappointing. They also grow for increasing rank level, but for **MEU** they result in query times of up to two seconds. This shows that for some queries the landmarks are not able to deliver strong bounds. Maybe a better landmark selection strategy could improve the worst case speed-ups.

l = 1 The highest speed-ups are still very large, which is not surprising, since some queries will have a departure time that results in low delay. For the largest instances and large rank levels the highest speed-ups are smaller, which is because the shortest-paths will be so long, that they cannot be entirely without some delay.

The average speed-ups falls to around 10 for high rank levels. There is still an increase in speed-up for increasing rank levels, but it seems to flatten. This is especially obvious for **DEU** and **MEU**, where the speed-up declines for the highest rank levels. This is again suspected to be, because long shortest-paths will always encounter some delay.

For the maximum solution time and maximum number of settled nodes, the speed-up is only up to two in some cases, but mostly there is no speed-up. The speed-up does not fall below one, or 0.5 because the speed-ups are rounded, so the overhead does not "punish" ALT too much compared to DJK.

l = 2, 3 The best speed-ups, as expected, remain very high. The average speed-ups further decline to around 4 to 6. In the worst case there is virtually no speed-up for any rank level.

A comparison of average speed-ups with regard to solution time and settled nodes show, that sometimes the speed-up in solution time is actually higher than speed-up in settled nodes. This is quite surprising, as there is a time overhead in ALT to find lower bounds compared to DJK. The reason seems to be that the fastest queries are so fast, that they are difficult to measure, and result in zero or near zero solution time. For these queries the number of settled nodes is more precise.

In conclusion ALT has very good best case behaviour, but also poor worst case behaviour. The average speed-up is quite good as long as the level of delay is low. For high levels of delay there is still some speed-up, but notice that this is partly because a portion of queries have departure times that results in no or low delay.

8.5 PCD

8.5.1 Preprocessing

Table 8.5 shows the preprocessing times in seconds of PCD for the four different cluster sizes.

	Instance	Time	Instance	Time	Instance	Time	Instance	Time
$k = 64$	DNK(0)	83	NLD(0)	176	DEU(0)	701	MEU(0)	1630
	DNK(1)	80	NLD(1)	171	DEU(1)	671	MEU(1)	1500
	DNK(2)	79	NLD(2)	165	DEU(2)	662	MEU(2)	1470
	DNK(3)	82	NLD(3)	168	DEU(3)	667	MEU(3)	1450
$k = 128$	DNK(0)	177	NLD(0)	349	DEU(0)	1390	MEU(0)	3030
	DNK(1)	159	NLD(1)	341	DEU(1)	1320	MEU(1)	2960
	DNK(2)	156	NLD(2)	328	DEU(2)	1300	MEU(2)	2900
	DNK(3)	161	NLD(3)	330	DEU(3)	1310	MEU(3)	2860
$k = 512$	DNK(0)	653	NLD(0)	1380	DEU(0)	5500	MEU(0)	12200
	DNK(1)	632	NLD(1)	1340	DEU(1)	5250	MEU(1)	11800
	DNK(2)	618	NLD(2)	1300	DEU(2)	5240	MEU(2)	11400
	DNK(3)	645	NLD(3)	1350	DEU(3)	5180	MEU(3)	11300
$k = 1024$	DNK(0)	1310	NLD(0)	2800	DEU(0)	11000	MEU(0)	24000
	DNK(1)	1227	NLD(1)	2690	DEU(1)	10700	MEU(1)	23600
	DNK(2)	1250	NLD(2)	2600	DEU(2)	10400	MEU(2)	23100
	DNK(3)	1280	NLD(3)	2680	DEU(3)	10500	MEU(3)	23200

Table 8.5: Preprocessing times for PCD in seconds.

In the preprocessing some obvious optimization options have not been used. For the same number of clusters and same instance, the same clustering have not been used. Doing so would limit the construction phase to once for each number of clusters, and finding lower bounds would only need to be done once for each instance.

Preprocessing is highly dominated by finding cluster distances, which is growing $2k$ complete shortest-path trees. For the largest instance **MEU** and number of clusters $k = 1024$, finding a clustering with k'-oversampling just takes 331 seconds out of the total 24,000 seconds.

It was expected that the number of access nodes for each cluster would be low. This turned out to be correct, as the average number is under six for all instances and all number of clusters.

8.5.2 Quering

For ALT the departure time has influence on the speed-up, because the speed-up is largest when the actual shortest-path distance is close to the most optimistic. For PCD it is different, the actual shortest-path distance to a node is used in both lower and upper bounding, so they somewhat cancel each other. Figure A.8 and A.9 in Appendix A.2 give an example using random source and target node. The figures suggest that speed-up for PCD may be limited when delay is imposed. Table 8.6 gives speed-up figures for PCD(1024). How the number of clusters affect speed-up will be shown later for a single instance.

		$l=0$			$l=1$			$l=2$			$l=3$		
	r	min	avg	max	min	avg	max	min	avg	max	min	avg	max
DNK	2^{13}	1 27	**1 3**	0 1	1 10	**1 2**	0 1	2 9	**1 2**	0 1	2 7	**1 1**	0 1
	2^{14}	2 55	**2 4**	1 1	3 20	**1 3**	1 1	3 13	**1 2**	1 1	3 19	**1 2**	1 1
	2^{15}	5 43	**3 6**	0 2	5 24	**2 3**	1 1	6 12	**1 2**	0 1	6 11	**1 2**	1 1
	2^{16}	11 58	**5 8**	1 1	11 41	**3 3**	1 1	6 9	**2 2**	1 1	6 16	**1 2**	1 1
	2^{17}	23 139	**8 10**	3 2	12 110	**3 3**	1 1	8 20	**2 2**	1 1	27 48	**2 2**	1 1
	2^{18}	48 141	**11 12**	3 2	25 30	**3 3**	2 2	17 29	**2 2**	1 1	6 7	**2 2**	1 1
	2^{19}	98 239	**14 12**	3 3	10 10	**2 2**	1 1	8 8	**2 2**	1 1	3 3	**1 1**	1 1
NLD	2^{13}	1 17	**1 2**	0 1	1 10	**1 2**	0 1	1 4	**1 1**	0 1	1 7	**1 1**	0 0
	2^{14}	1 18	**1 3**	0 1	2 13	**1 2**	0 1	1 13	**1 2**	0 1	1 7	**1 1**	0 1
	2^{15}	3 42	**1 4**	0 1	2 14	**1 3**	1 1	2 10	**1 2**	1 1	2 10	**1 2**	1 1
	2^{16}	5 52	**3 5**	1 2	4 42	**2 3**	1 1	4 8	**2 2**	1 1	5 8	**1 2**	1 1
	2^{17}	11 62	**5 8**	2 2	8 16	**3 3**	1 1	5 6	**2 2**	1 1	4 5	**2 2**	1 1
	2^{18}	24 136	**8 11**	2 2	17 26	**4 4**	2 2	13 22	**2 2**	1 1	5 4	**2 2**	1 1
	2^{19}	34 178	**13 13**	3 2	15 16	**4 3**	2 1	9 8	**2 2**	1 1	6 6	**2 2**	1 1
	2^{20}	74 486	**20 17**	4 3	16 13	**4 3**	2 2	8 6	**2 2**	1 1	4 3	**2 1**	1 1
DEU	2^{13}	0 5	**0 1**	0 1	0 5	**0 1**	0 1	0 2	**0 1**	0 1	0 3	**0 1**	0 0
	2^{14}	1 10	**0 2**	0 1	0 10	**0 1**	0 1	0 3	**0 1**	0 1	0 3	**0 1**	0 0
	2^{15}	1 14	**1 2**	0 1	1 6	**1 2**	0 1	1 4	**1 1**	0 1	1 4	**1 1**	0 1
	2^{16}	1 28	**1 3**	0 1	1 13	**1 2**	0 1	1 8	**1 2**	1 1	1 8	**1 1**	0 1
	2^{17}	2 26	**2 4**	1 2	2 10	**1 2**	1 1	2 5	**1 2**	1 1	2 4	**1 2**	1 1
	2^{18}	5 23	**3 5**	2 2	4 8	**2 3**	1 1	3 7	**2 2**	1 1	2 4	**1 2**	1 1
	2^{19}	9 46	**5 7**	2 2	7 12	**3 3**	2 2	5 6	**2 2**	1 1	4 5	**2 2**	1 1
	2^{20}	20 83	**8 8**	3 2	13 14	**4 3**	2 2	7 7	**2 2**	1 1	3 3	**2 2**	1 1
	2^{21}	41 88	**14 11**	5 4	13 11	**4 3**	2 2	6 5	**2 2**	1 1	3 3	**2 2**	1 1
	2^{22}	81 131	**19 13**	3 2	12 9	**3 3**	2 1	21 18	**2 2**	1 1	3 3	**1 1**	1 1
MEU	2^{13}	1 4	**0 1**	0 1	1 5	**0 1**	0 0	1 4	**0 1**	0 0	1 4	**0 1**	0 0
	2^{14}	0 10	**0 1**	0 1	0 3	**0 1**	0 1	0 5	**0 1**	0 1	0 7	**0 1**	0 0
	2^{15}	1 10	**0 2**	0 1	1 5	**0 1**	0 1	1 3	**0 1**	0 1	1 6	**0 1**	0 1
	2^{16}	1 9	**1 2**	0 1	1 6	**1 2**	0 1	1 6	**1 1**	0 1	1 7	**1 1**	0 1
	2^{17}	1 10	**1 3**	1 1	1 14	**1 2**	1 1	1 4	**1 2**	1 1	2 6	**1 1**	1 1
	2^{18}	3 19	**2 4**	1 1	2 9	**1 2**	1 1	2 5	**1 2**	1 1	2 5	**1 1**	1 1
	2^{19}	5 23	**3 5**	1 2	5 13	**2 3**	1 1	4 10	**2 2**	1 1	6 37	**1 2**	1 1
	2^{20}	12 72	**5 6**	2 2	9 16	**3 3**	1 1	6 7	**2 2**	1 1	6 7	**2 2**	1 1
	2^{21}	23 56	**9 7**	2 1	14 15	**3 3**	2 1	8 9	**2 2**	1 1	6 6	**2 2**	1 1
	2^{22}	45 65	**12 9**	3 2	15 12	**4 3**	2 1	8 6	**2 2**	1 1	5 4	**2 2**	1 1
	2^{23}	80 83	**14 9**	3 2	9 7	**3 2**	2 1	6 5	**2 2**	1 1	4 4	**1 1**	1 1

Table 8.6: Quering speed-ups for PCD(1024).

We will again evaluate for each level of delay and finish with some general observations.

l = 0 Similar to ALT, the best speed-ups are quite good, at least for large rank levels. These are achieved for queries where the source and target are favourably placed with regard to clusters, and when the number of near optimal paths are limited. When there is many paths between source and target that are near optimal, PCD may have to explore in the direction of these paths, if bounding is not tight enough to prune. A source node can be favourably placed if it for example is the source node on the shortest-path from source to target cluster. An upper bound will then quickly be obtained.

The largest speed-ups in the average case is for the largest rank levels, and comparable to those of ALT. The same number of clusters are used for all instances, so for the large instances the clusters will contain more nodes, and a good speed-up is first obtained for long shortest-paths.

The worst case speed-ups are not very good. For larger rank levels they are between 2 and 5, while for lower rank levels they are even less than 0.5. We will get back to explaining speed-ups under 1.

l = 1 The speed-up rapidly declines when delay is imposed. The best case speed-ups are no more than the static average ones. The average speed-ups are between 2 and 4 for larger rank levels. There is practically no speed-up for worst case queries.

l = 2, 3 The highest speed-up in the average case is 2, and in the worst case there is no speed-up at all.

It is clear from the table, that for shorter queries there is an overhead by using PCD. When source and target nodes are in the same cluster no pruning is possible at all, and when the source and target cluster are close, the searches in phase 2 become noticeable in the speed-up. For large instances, clusters will contain many nodes, so phase 1 of the search will have to settle many nodes. Furthermore the two searches in phase 2 will have to search a large cluster. The memory usage of PCD is low compared to ALT, so we could just use more clusters, but this requires a lot of preprocessing for the large instances, as seen from Table 8.5. The hardware used in experimentation was not available for that period of time.

The speed-up of PCD is quite disappointing when delay is imposed. The gap between upper and lower cluster distances becomes so large, that pruning is limited. As mentioned before, the delay we impose on edges are quite harsh, because many important edges will be given delay. For the example source and target pair in Appendix A.2, the most optimistic travel time is 138 minutes. For departure time 07:00 and delay level 3, the travel time is 245 minutes. This is a 56 % increase which seems plausible for a trip around rush hour in the Netherlands, so we cannot just blame low speed-up on too pessimistic delays. But we can argue that too many upper bound cluster distances are affected by high delay. Not every important road suffers from congestion on a normal day.

One way to strengthen the cluster distances would be to use time-dependent cluster distances. An approach similar to the one described for landmarks in section 4.3.2. The time horizon is separated into periods of time where the travel time does not vary much. For each of these periods most optimistic and pessimistic cluster distances are found. So between clusters there is a set of distances that apply to specific intervals of departure time. The approach uses more memory and more preprocessing time, and the worst case queries will still have low speed-up because they can occur when the gap between bounds is large.

8.5.3 Cluster Number Influence

We would like to evaluate how number of clusters affects the speed-up of PCD. We do the evaluation for a single instance **NLD**. Table 8.7 shows the result.

For constant travel time the speed-up improves with increasing number of clusters. A significant speed-up is first obtained for $k = 512$ clusters. The worst case speed-ups remain around three for $k = 2048$, so it is still possible to find queries that give limited pruning. For low rank levels the speed-up remain limited for all number of clusters, but it improves for the high cluster counts.

When delay is imposed the speed-ups rapidly decline. Only a small improvement is obtained going from $k = 1024$ to $k = 2048$. For $l = 1$ an average speed-up around 4 is obtained for high rank levels, while for higher levels of delay the speed-up remains around 2.

It can be concluded that without delay, the method of PCD is able to deliver significant speed-ups for high rank levels at a very low memory cost compared to ALT. The preprocessing time is on the other hand much larger for PCD, and the use of upper bounds requires finding new bounds if the travel times change.

8.6 PCDALT

The last algorithm to evaluate is PCDALT. Table 8.8 shows the results.

k	r	$l=0$ min	$l=0$ avg	$l=0$ max	$l=1$ min	$l=1$ avg	$l=1$ max	$l=2$ min	$l=2$ avg	$l=2$ max	$l=3$ min	$l=3$ avg	$l=3$ max
64	2^{13}	2 6	**0 1**	0 1	2 3	**0 1**	0 1	2 2	**0 1**	0 0	2 4	**0 1**	0 0
	2^{14}	1 7	**0 1**	0 1	1 4	**0 1**	0 0	1 5	**0 1**	0 0	1 3	**0 1**	0 0
	2^{15}	1 4	**1 1**	0 1	1 3	**1 1**	0 1	1 5	**1 1**	0 0	1 2	**1 1**	0 0
	2^{16}	2 7	**1 1**	0 1	1 5	**1 1**	0 1	2 4	**1 1**	0 0	1 2	**1 1**	0 0
	2^{17}	3 9	**1 2**	1 1	3 5	**1 1**	1 1	2 3	**1 1**	0 0	3 4	**1 1**	1 1
	2^{18}	6 13	**2 2**	1 1	4 4	**2 2**	1 1	3 3	**1 1**	1 1	2 2	**1 1**	1 1
	2^{19}	9 13	**3 3**	1 1	6 6	**2 2**	1 1	4 4	**1 1**	1 1	3 2	**1 1**	1 1
	2^{20}	17 16	**4 3**	1 1	6 5	**2 2**	1 1	4 3	**1 1**	1 1	3 3	**1 1**	1 1
128	2^{13}	1 10	**0 1**	0 1	1 3	**0 1**	0 1	1 3	**0 1**	0 0	1 2	**0 1**	0 0
	2^{14}	1 8	**0 1**	0 1	1 5	**0 1**	0 1	1 4	**1 1**	0 0	1 3	**1 1**	0 0
	2^{15}	2 11	**1 1**	0 1	1 8	**1 1**	0 1	1 4	**1 1**	0 0	1 4	**1 1**	0 0
	2^{16}	3 14	**1 2**	0 1	2 4	**1 2**	0 1	2 4	**1 1**	1 1	2 3	**1 1**	0 0
	2^{17}	6 24	**2 3**	1 1	5 11	**1 2**	0 1	4 6	**1 1**	1 1	3 3	**1 1**	0 1
	2^{18}	8 22	**3 3**	1 1	6 8	**2 2**	1 1	6 7	**2 2**	1 1	3 3	**1 1**	1 1
	2^{19}	17 30	**5 4**	2 1	8 7	**3 2**	1 1	4 4	**2 2**	1 1	3 3	**1 1**	1 1
	2^{20}	28 29	**6 5**	2 2	8 7	**2 2**	1 1	4 3	**2 2**	1 1	3 3	**1 1**	1 1
512	2^{13}	1 6	**1 1**	0 1	1 5	**1 1**	0 1	1 5	**1 1**	0 0	1 6	**1 1**	0 0
	2^{14}	1 18	**1 2**	0 1	2 16	**1 2**	0 1	2 4	**1 1**	0 1	1 5	**1 1**	0 0
	2^{15}	3 14	**1 3**	1 1	2 7	**1 2**	0 1	2 6	**1 2**	0 1	2 6	**1 1**	0 1
	2^{16}	5 28	**2 4**	0 1	4 15	**2 3**	1 1	3 6	**2 2**	1 1	4 5	**1 1**	1 1
	2^{17}	7 34	**4 5**	1 2	6 18	**3 3**	1 1	4 7	**2 2**	1 1	5 6	**2 2**	1 1
	2^{18}	24 109	**6 6**	2 2	10 15	**3 3**	1 1	6 6	**2 2**	1 1	7 7	**2 2**	1 1
	2^{19}	34 142	**10 9**	3 2	21 24	**4 3**	1 1	6 5	**2 2**	1 1	5 4	**2 2**	1 1
	2^{20}	56 159	**16 12**	3 2	13 10	**3 3**	2 1	7 6	**2 2**	1 1	4 3	**2 1**	1 1
1024	2^{13}	1 17	**1 2**	0 1	1 10	**1 2**	0 1	1 4	**1 1**	0 1	1 7	**1 1**	0 0
	2^{14}	1 18	**1 3**	0 1	2 13	**1 2**	0 1	1 13	**1 2**	0 1	1 7	**1 1**	0 1
	2^{15}	3 42	**1 4**	0 1	2 14	**1 3**	1 1	2 10	**1 2**	1 1	2 10	**1 2**	1 1
	2^{16}	5 52	**3 5**	1 2	4 42	**2 3**	1 1	4 8	**2 2**	1 1	5 8	**1 2**	1 1
	2^{17}	11 62	**5 8**	2 2	8 16	**3 3**	1 1	5 6	**2 2**	1 1	4 5	**2 2**	1 1
	2^{18}	24 136	**8 11**	2 2	17 26	**4 4**	2 2	13 22	**2 2**	1 1	5 4	**2 2**	1 1
	2^{19}	34 178	**13 13**	3 2	15 16	**4 3**	2 1	9 8	**2 2**	1 1	6 6	**2 2**	1 1
	2^{20}	74 486	**20 17**	4 3	16 13	**4 3**	2 2	8 6	**2 2**	1 1	4 3	**2 1**	1 1
2048	2^{13}	1 19	**1 3**	0 1	1 8	**1 2**	0 1	1 8	**1 2**	0 1	1 10	**1 2**	0 1
	2^{14}	1 47	**1 4**	0 1	2 17	**1 3**	0 1	2 10	**1 2**	0 1	2 60	**1 2**	0 1
	2^{15}	3 48	**2 6**	1 1	3 18	**2 3**	1 1	3 42	**1 2**	1 1	4 13	**1 2**	1 1
	2^{16}	5 122	**3 8**	2 2	6 32	**2 4**	1 2	4 9	**2 3**	1 1	4 8	**2 2**	1 1
	2^{17}	11 388	**6 11**	2 2	12 29	**4 4**	2 2	9 19	**3 3**	1 1	6 7	**2 2**	1 1
	2^{18}	24 219	**11 16**	3 3	17 29	**5 5**	2 2	9 8	**3 2**	1 1	7 6	**2 2**	1 1
	2^{19}	52 651	**19 19**	3 2	21 23	**5 4**	2 2	16 12	**3 2**	1 1	5 4	**2 2**	1 1
	2^{20}	111 486	**36 31**	4 3	29 24	**4 3**	2 2	10 7	**2 2**	1 1	6 4	**2 2**	1 1

*Table 8.7: Quering speed-ups for PCD(k) in **NLD**.*

l = 0 For constant travel time the combination of PCD and ALT gives extra average speed-up for high rank levels compared to the two individual methods. For low rank levels the speed-up is limited, which indicates that the overhead of the two methods dominate. This is also indicated by the difference in speed-up for solution time and settled nodes. The speed-up in settled nodes is much higher than for solution time.

The best speed-ups are still very high and attributable to ALT. The worst case speed-ups are like the ones for ALT, which is not surprising, as ALT delivers the best worst case speed-ups compared to PCD.

l > 0 Worst-, best- and average speed-ups are quite like the ones for ALT, which indicates that all speed-up are being delivered by ALT. Comparing the speed-ups for ALT and PCD this is not surprising, as ALT nearly always gives the best speed-ups.

It can be concluded that in the static case PCD can actually prune nodes in the goal directed A* search. But as the quality of PCD falls when delay is imposed, the pruning gets very limited.

	r	$l=0$ min	$l=0$ avg	$l=0$ max	$l=1$ min	$l=1$ avg	$l=1$ max	$l=2$ min	$l=2$ avg	$l=2$ max	$l=3$ min	$l=3$ avg	$l=3$ max
DNK	2^{13}	1 149	**1 7**	0 2	1 159	**1 5**	0 1	– 72	**1 4**	0 1	2 110	**1 4**	0 1
	2^{14}	2 241	**2 9**	0 3	3 246	**2 6**	0 2	3 328	**1 5**	1 1	3 74	**1 3**	1 1
	2^{15}	5 262	**3 12**	1 2	5 259	**2 6**	1 2	6 239	**2 5**	1 2	6 168	**2 4**	0 1
	2^{16}	11 489	**5 15**	2 2	11 365	**4 8**	1 2	12 308	**3 6**	1 2	13 69	**2 4**	1 1
	2^{17}	23 794	**9 19**	2 2	23 849	**5 8**	1 1	24 313	**3 6**	1 1	27 147	**3 4**	1 1
	2^{18}	48 589	**15 25**	4 5	49 562	**6 8**	2 2	50 806	**4 6**	1 2	28 99	**3 4**	1 1
	2^{19}	98 1603	**17 22**	5 6	50 342	**5 7**	1 2	51 646	**3 5**	1 1	110 860	**3 4**	1 1
NLD	2^{13}	1 130	**1 6**	0 2	1 190	**1 6**	0 2	1 246	**1 5**	0 1	1 117	**1 4**	0 1
	2^{14}	1 278	**1 8**	1 3	2 182	**1 6**	0 2	2 347	**1 5**	1 1	2 206	**1 4**	0 1
	2^{15}	3 269	**2 10**	0 2	3 634	**2 7**	0 2	3 443	**2 6**	1 1	4 460	**2 4**	0 1
	2^{16}	5 936	**3 13**	2 3	6 240	**3 9**	1 2	4 393	**3 7**	1 2	7 185	**2 5**	1 1
	2^{17}	11 364	**6 16**	3 4	12 1059	**5 10**	2 2	13 189	**4 8**	1 2	14 497	**3 5**	1 1
	2^{18}	24 1872	**11 24**	3 4	25 347	**7 12**	2 2	27 484	**6 9**	1 2	20 734	**4 6**	1 1
	2^{19}	34 1928	**17 28**	6 7	36 2251	**8 12**	2 3	38 286	**6 8**	1 2	42 502	**4 6**	1 1
	2^{20}	111 4112	**29 37**	7 6	115 3483	**8 10**	1 2	82 894	**6 8**	1 1	67 224	**4 5**	1 1
DEU	2^{13}	2 149	**0 6**	0 2	2 250	**0 5**	0 1	2 109	**0 5**	0 1	1 54	**0 4**	0 1
	2^{14}	2 171	**1 7**	0 2	2 253	**1 6**	0 2	2 141	**1 5**	0 1	3 88	**1 4**	0 1
	2^{15}	1 415	**1 8**	0 2	4 340	**1 7**	0 2	1 78	**1 6**	1 2	1 529	**1 5**	0 1
	2^{16}	2 607	**1 10**	1 3	2 124	**1 7**	1 2	2 389	**1 6**	1 1	2 157	**1 5**	0 1
	2^{17}	3 496	**2 11**	1 4	3 834	**2 9**	1 2	3 122	**2 6**	1 2	3 248	**2 5**	1 1
	2^{18}	6 1417	**4 14**	2 4	5 1663	**3 9**	1 2	6 549	**3 7**	1 2	6 157	**3 6**	1 1
	2^{19}	13 824	**7 18**	3 5	13 966	**5 10**	1 2	11 109	**4 7**	1 2	13 155	**4 6**	1 1
	2^{20}	25 3785	**13 22**	3 4	26 645	**8 11**	2 2	27 2205	**6 7**	1 2	28 4845	**5 6**	1 1
	2^{21}	55 983	**21 25**	6 7	56 3012	**9 11**	2 3	52 406	**6 7**	1 2	49 284	**5 6**	1 1
	2^{22}	131 6223	**31 28**	5 6	107 674	**8 9**	2 2	123 2061	**6 6**	1 2	89 401	**4 5**	1 1
MEU	2^{13}	1 105	**0 5**	0 2	1 182	**0 5**	0 1	1 54	**0 4**	0 1	1 87	**0 4**	0 1
	2^{14}	2 148	**0 6**	0 1	2 159	**0 5**	0 2	2 139	**0 4**	0 1	2 99	**0 4**	0 1
	2^{15}	2 683	**1 7**	0 2	1 116	**1 5**	0 2	2 198	**1 4**	0 1	1 85	**1 4**	0 1
	2^{16}	1 475	**1 8**	0 2	3 434	**1 6**	0 2	3 165	**1 5**	0 1	1 163	**1 4**	1 1
	2^{17}	2 1192	**1 9**	1 2	2 474	**1 7**	1 2	2 971	**1 5**	1 1	2 147	**1 5**	1 1
	2^{18}	3 592	**2 10**	1 2	3 225	**2 7**	1 2	3 288	**2 5**	1 1	4 491	**2 5**	1 1
	2^{19}	6 2521	**4 11**	2 3	6 960	**3 8**	1 2	7 2900	**3 6**	1 2	8 533	**3 5**	1 1
	2^{20}	13 1452	**7 13**	3 4	13 550	**5 8**	1 2	14 706	**4 6**	1 2	15 932	**4 6**	1 1
	2^{21}	28 6513	**12 16**	2 4	27 324	**6 9**	1 2	30 850	**5 7**	1 1	26 128	**4 5**	1 1
	2^{22}	56 426	**20 21**	4 4	62 2271	**8 9**	2 3	50 179	**5 6**	1 2	59 370	**4 5**	1 2
	2^{23}	122 1225	**23 19**	4 3	135 9654	**6 7**	2 2	80 146	**4 5**	1 2	80 154	**4 4**	1 1

Table 8.8: Quering speed-ups for PCDALT.

8.7 Random Delay

When delay is imposed we get limited speed-up for all speed-up algorithms. It has been discussed if this is partly because the used instances have too much delay on important edges compared to reality. In this section we will try to solve two instances made with delay on random edges. We use the same delay functions as before and assign both outward and inward delay to 10% of the edges chosen at random. One of the five delay functions are also chosen at random. Table 8.9 shows the result for the three speed-up algorithms and instances **DNK** and **NLD**. The speed-ups are relative to the average solution time with DJK in the instance with no delay $l = 0$.

Compared to the $l = 3$ results for the instances, the speed-ups are much higher. For ALT they are approximately doubled, for PCD(1024) and PCDALT they are around three times as high. For PCD there is especially a clear improvement, because the gap between most optimistic and most pessimistic cluster distances has been reduced. The shortest paths between clusters clearly does not suffer from the same level of delay.

Assigning delay randomly to 10% of the edges is of course not a realistic distribution of delay. But maybe the realistic assignment of delay is somewhere in between, so usable speed-ups may be possible to obtain for realistic time-dependent shortest-paths quering in road networks.

	r	ALT				PCD(1024)				PCDALT			
		min		avg		max		min		avg		max	
DNK	2^{13}	–	195	**2**	**5**	1	1	1	18	**1**	**2**	0	1
	2^{14}	–	111	**3**	**6**	1	1	2	24	**1**	**3**	0	1
	2^{15}	–	303	**4**	**7**	1	2	5	18	**2**	**4**	1	1
	2^{16}	–	144	**5**	**8**	1	2	11	25	**4**	**5**	1	2
	2^{17}	–	753	**5**	**8**	1	2	23	83	**5**	**6**	2	2
	2^{18}	–	877	**6**	**9**	1	2	48	54	**6**	**5**	2	2
	2^{19}	–	1533	**5**	**7**	2	2	33	44	**5**	**5**	2	2
NLD	2^{13}	–	126	**2**	**6**	1	2	1	22	**1**	**2**	0	1
	2^{14}	–	234	**2**	**6**	1	2	1	13	**1**	**2**	0	1
	2^{15}	–	254	**4**	**8**	1	2	3	15	**1**	**3**	1	1
	2^{16}	–	705	**5**	**9**	2	3	5	29	**2**	**4**	1	1
	2^{17}	–	475	**6**	**10**	2	3	11	64	**4**	**5**	1	1
	2^{18}	–	536	**8**	**11**	2	3	16	61	**5**	**6**	2	2
	2^{19}	–	779	**9**	**12**	2	3	26	67	**7**	**6**	2	2
	2^{20}	222	2804	**9**	**12**	2	3	44	49	**7**	**6**	2	2

	r	PCDALT				
		min		avg		max
DNK	2^{13}	1	120	**1**	**6**	0 2
	2^{14}	2	87	**2**	**6**	0 1
	2^{15}	5	172	**2**	**8**	1 2
	2^{16}	11	383	**4**	**9**	2 3
	2^{17}	23	130	**6**	**10**	2 3
	2^{18}	48	710	**7**	**10**	2 4
	2^{19}	98	638	**7**	**9**	2 3
NLD	2^{13}	1	124	**1**	**5**	0 2
	2^{14}	1	321	**1**	**7**	0 2
	2^{15}	3	241	**2**	**8**	1 2
	2^{16}	5	211	**3**	**10**	1 3
	2^{17}	11	319	**4**	**11**	1 3
	2^{18}	24	1057	**8**	**14**	2 4
	2^{19}	52	3449	**10**	**15**	2 3
	2^{20}	111	1419	**11**	**15**	2 3

Table 8.9: Quering speed-ups using random delay on 10% of the edges.

Chapter 9

Conclusions

We have in this thesis presented the time-dependent shortest-paths problem in the context of large road networks. It has been shown that it is possible to represent time-dependent travel time rather effectively by piece-wise linear functions. The storage consumption is acceptable for the used level of detail, and for the percentage of edges assigned a travel time function.

Existing speed-up techniques for the static shortest-paths problem in road networks have been presented, and it has been discussed if and how these could be adapted to the time-dependent problem. Generally it is difficult to adapt speed-up techniques due to the more dynamic nature of the time-dependent problem. Two goal directed techniques, A* search using landmarks and the triangle inequality, and precomputed cluster distances have been adapted. Furthermore the two methods have been combined with the aim of achieving an additional speed-up. To the best of our knowledge we are the first to have adapted precomputed cluster distances to the time-dependent problem.

To test the performance of the algorithms simulated instances have been developed. The objective was instances with some touch of realism, which proved difficult given the limited available data. The instances became somewhat worst case examples due to the high delay imposed on all important edges.

Experimentation has shown that the adapted speed-up techniques are highly sensible to the level of delay imposed. Speed-up rapidly declines with increasing level of delay, especially for PCD that has low speed-ups even for low levels of delay. ALT seems to be the most versatile method as it only relies on lower bounds and therefore the achieved speed-up is only dependent on the actual delays suffered at a given departure time. Furthermore the method can be used in the dynamic time-dependent problem. Generally both methods have poor worst case speed-ups. The combined method gives an additional speed-up in the static case, but as PCD gives poor speed-ups for higher delay levels, the speed-ups approaches those of ALT.

The experimentation shows that the speed-ups of both methods are considerably better for an instance with delay on randomly chosen edges. Considering the simulated instances as a worst case, this suggests that usable speed-ups may be achievable for realistic instances. The concept of time-dependent bounds are shortly mentioned. These could be a way to improve the speed-ups of both methods, but would require additional storage and preprocessing time.

Progress is being made in the area of shortest-paths in time-dependent road networks. The article by Delling and Wagner [13], giving an adaption of ALT, was published after we independently made the same adaption. It is expected that further progress will be made, as more reliable and accurate route planning will require, that the dynamic characteristics of road networks are also taken into account.

Appendix A

Visualizations

A.1 Simulated Traffic Counts

In this appendix figures indicating outward traffic counts found by the simulation are given for all the instances used in experimentation together with Italy and a zoom of the greater Copenhagen area.

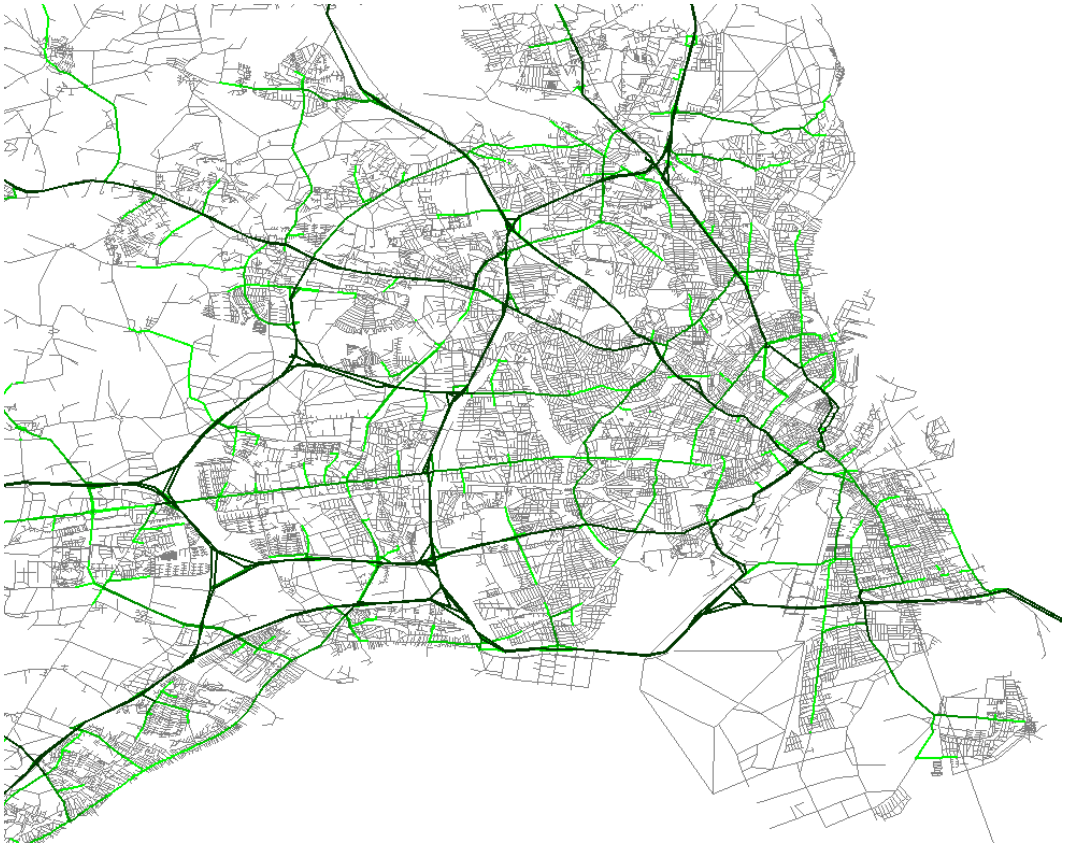


Figure A.1: Assigned outward delay forecasts for the greater Copenhagen area, indicated by a gradient of green.

Figure A.2: Assigned outward delay forecasts for the Netherlands, indicated by a gradient of green.

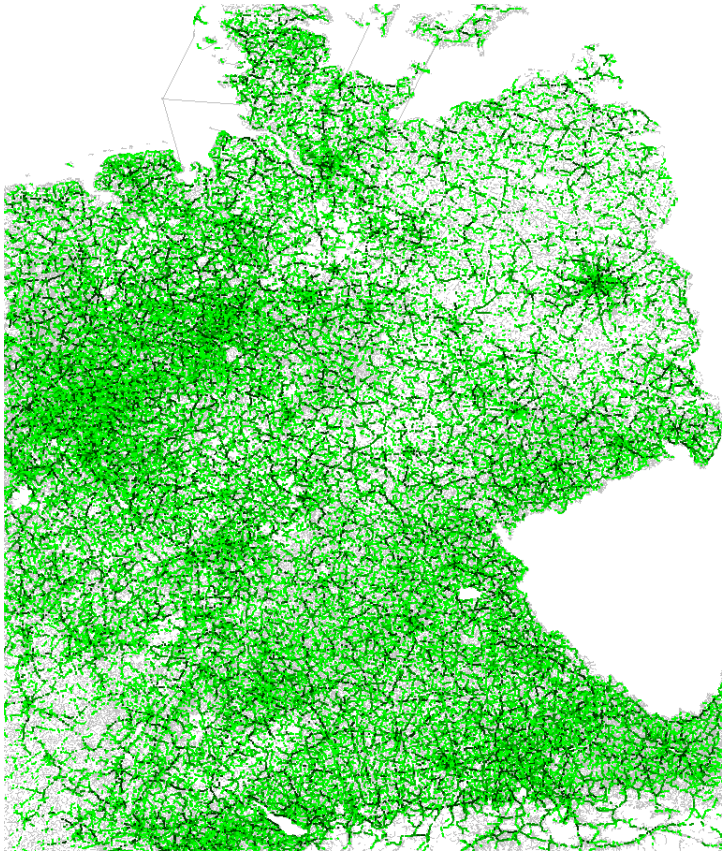


Figure A.3: Assigned outward delay forecasts for Germany, indicated by a gradient of green.

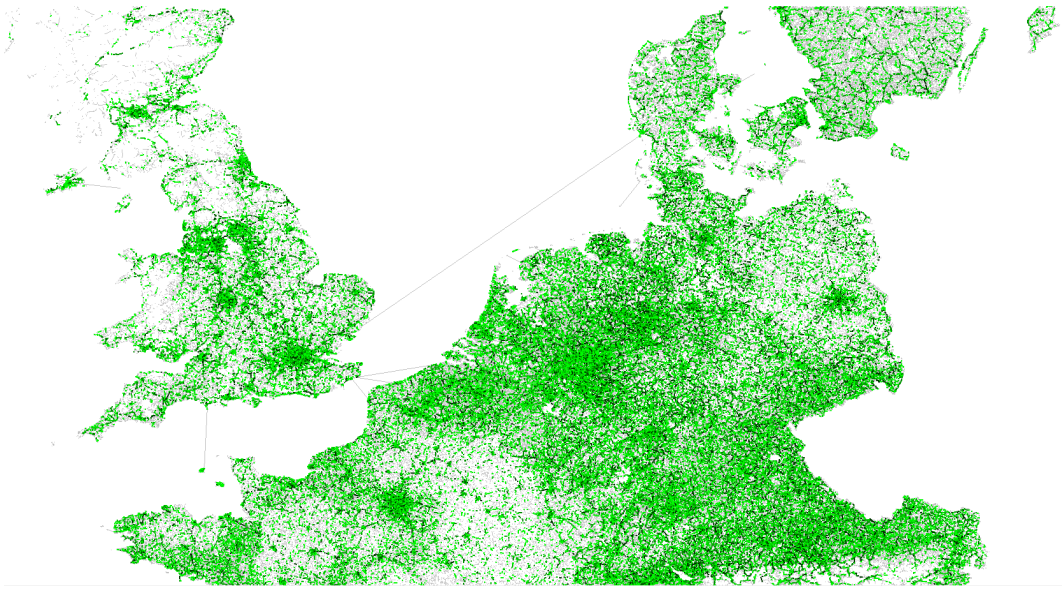


Figure A.4: Assigned outward delay forecasts for North-West Europe, indicated by a gradient of green.

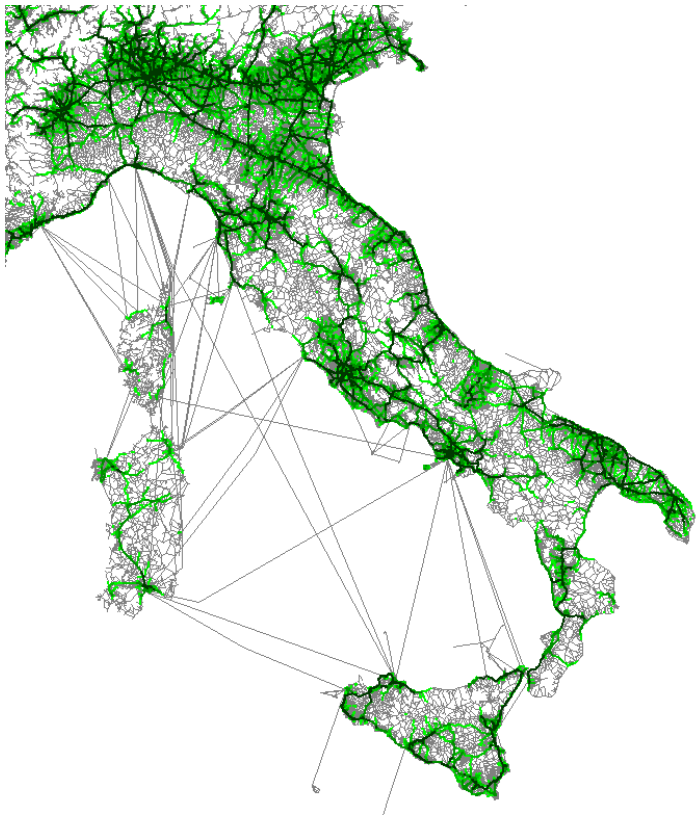
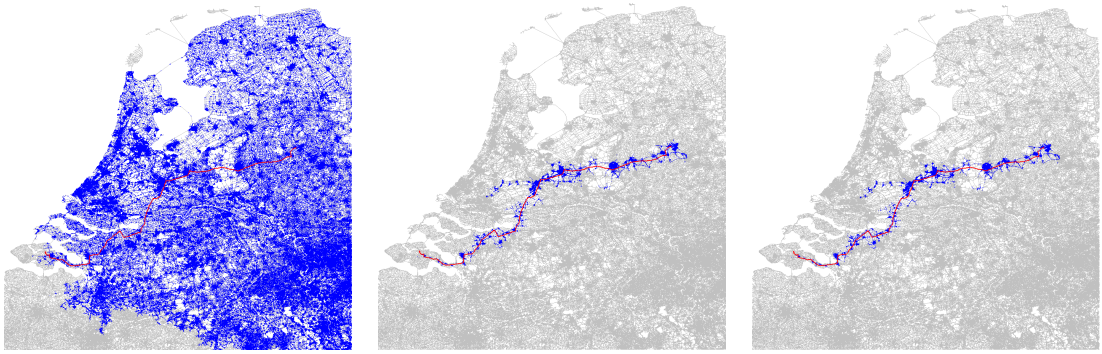


Figure A.5: Assigned outward delay forecasts for Italy, indicated by a gradient of green.

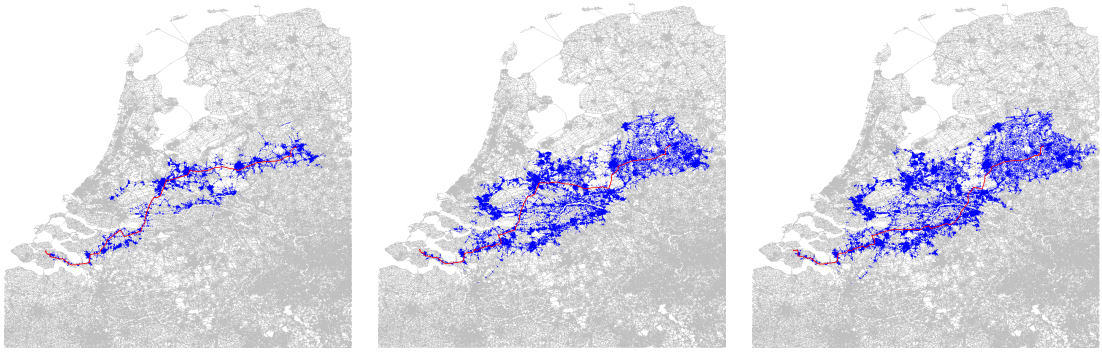
A.2 Solution Examples

Figure A.6 shows that the speed-up of ALT only depends on the delay of a specific query. Figure A.7 shows how the number of settled nodes is affected by different levels of delay. The figures also shows that shortest-path depends on delay and departure time.



(a) **NLD**(0) and DJK. (b) **NLD**(0) and ALT. (c) **NLD**(3) and ALT.

*Figure A.6: Query in **NLD** with DJK and ALT for two delay levels $l = 0$ and $l = 3$. All departure times 00:00.*



(a) **NLD**(1) and ALT. (b) **NLD**(2) and ALT. (c) **NLD**(3) and ALT.

Figure A.7: The same query for three different levels of delay and departure time 08:00.

Figure A.8 shows the settled nodes by PCD(1024) for **NLD**(0), and source and target as above. Notice how PCD settles nodes on paths that are close to optimal, but prune around the candidate paths. Figure A.9 shows how PCD(1024) are affected by different levels of delay and departure time.

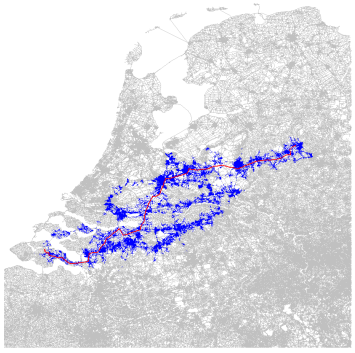
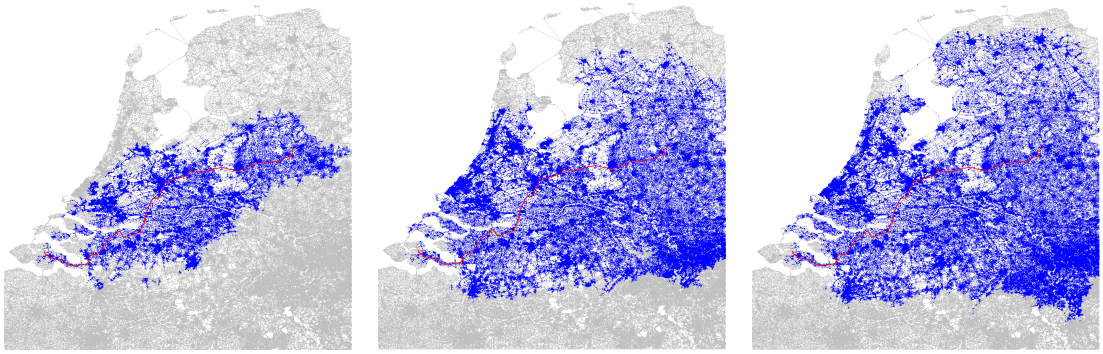
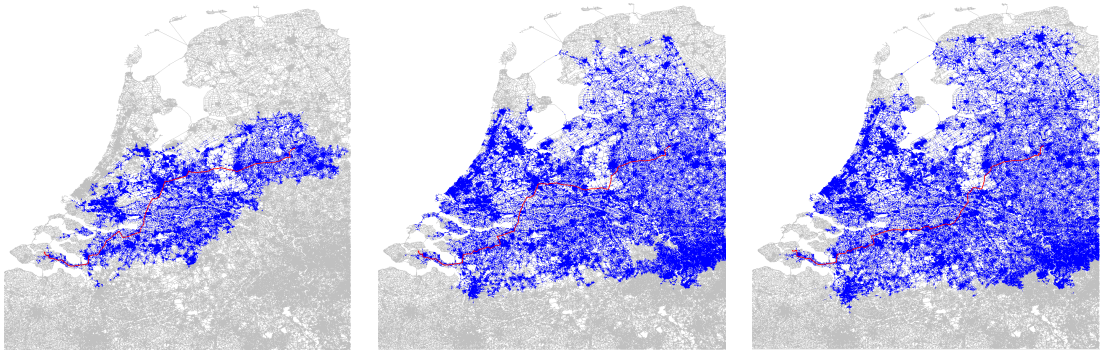


*Figure A.8: **NLD**(0) with PCD(1024)*



(a) **NLD**(1) 00:00 (b) **NLD**(2) 00:00 (c) **NLD**(3) 00:00



(d) **NLD**(1) 08:00 (e) **NLD**(2) 08:00 (f) **NLD**(3) 08:00

Figure A.9: The same query solved with PCD(1024) for three different levels of delay and departure times 00:00 and 08:00.

Bibliography

[1] 9th DIMACS Implementation Challenge on Shortest Paths. `http://www.dis.uniroma1.it/ challenge9/`, 2006.

[2] PTV AG. `http://www.ptv.de`.

[3] Holger Bast, Stefan Funke, Domagoj Matijevic, Peter Sanders, and Dominik Schultes. In transit to constant time shortest-path queries in road networks. In *ALENEX*. SIAM, 2007.

[4] Reinhard Bauer. Dynamic speed-up techniques for dijkstra's algorithm. Master's thesis, Universität Karlsruhe, 2006.

[5] Reinhard Bauer, Daniel Delling, and Dorothea Wagner. 14. experimental study on speed-up techniques for timetable information systems. In Christian Liebchen, Ravindra K. Ahuja, and Juan A. Mesa, editors, *ATMOS 2007 - 7th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems*, Dagstuhl, Germany, 2007.

[6] Allan Christensen. `ac(at)vd.dk`. Road Directorate Danish Ministry of Transport and Energy.

[7] K.L. Cooke and E. Halsey. The shortest route through a network with time dependent internodal transit times. *Journal of Mathematical Analysis Applications 14*, pages 493–498, 1966.

[8] K.L. Cooke and E. Halsey. An appraisal of some shortest-path algorithms. *Operations Research 17*, pages 395–412, 1969.

[9] K.L. Cooke and E. Halsey. Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length. *Journal of the Association for Computing Machinery 37(3)*, pages 607–625, 1990.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[11] Brian C. Dean. Shortest paths in fifo time-dependent networks: Theory and algorithms. *Technical report*, 2004.

[12] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Highway hierarchies star. In *9th DIMACS Challenge on Shortest Paths*, November 2006.

[13] Daniel Delling and Dorothea Wagner. Landmark-based routing in dynamic graphs. In Camil Demetrescu, editor, *Experimental Algorithms, 6th International Workshop, WEA 2007, Rome, Italy, June 6-8, 2007, Proceedings*, volume 4525 of *Lecture Notes in Computer Science*, pages 52–65. Springer, 2007.

[14] Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[15] Center for Discrete Mathematics & Theoretical Computer Science Founded as a National Science Foundation Science and Technology Center. `http://dimacs.rutgers.edu`.

[16] Frigioni, Marchetti-Spaccamela, and Nanni. Fully dynamic algorithms for maintaining shortest paths trees. *ALGORITHMS: Journal of Algorithms*, 34, 2000.

[17] A. V. Goldberg, Haim Kaplan, and Renato Werneck. Reach for a^*: Efficient point-to-point shortest path algorithms. Technical Report MSR-TR-2005-132, Microsoft Research (MSR), October 2005.

[18] Andrew V. Goldberg and Chris Harrelson. Computing the shortest path: a^* search meets graph theory. In *SODA*, pages 156–165. SIAM, 2005.

[19] Andrew V. Goldberg and Renato Fonseca F. Werneck. Computing point-to-point shortest paths from external memory. In Camil Demetrescu, Robert Sedgewick, and Roberto Tamassia, editors, *Proceedings of the Seventh Workshop on Algorithm Engineering and Experiments and the Second Workshop on Analytic Algorithmics and Combinatorics, ALENEX /ANALCO 2005, Vancouver, BC, Canada, 22 January 2005*, pages 26–40. SIAM, 2005.

[20] Teofilo F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38(2–3):293–306, June 1985.

[21] Ronald J. Gutman. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In *Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX'04)*, pages 100–111. SIAM, 2004.

[22] K. Ozbay H. Sherali and S. Subramanian. The time dependent shortest pair of disjoint paths problem: Complexity, models and algorithms. *Networks 31*, pages 259–272, 1998.

[23] Michael T. Heath. *Scientific Computing: An Introductory Survey.* McGraw-Hill, New York, second edition, 2002.

[24] Dorit Hochbaum, editor. *Approximation Algorithms for NP-hard Problems.* PWS Publishing Company, 1997.

[25] Stefan Funke Holger Bast and Domagoj Matijevic. Transit: Ultrafast shortest-path queries with linear-time preprocessing. In Camil Demetrescu, Andrew Goldberg, and David Johnson, editors, *9th DIMACS Implementation Challenge — Shortest Path*, Piscataway, New Jersey, 2006. DIMACS.

[26] Martin Holzer, Frank Schulz, and Dorothea Wagner. Engineering Multi-Level Overlay Graphs for Shortest-Path Queries. In *Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments (ALENEX 2006)*, volume 129 of *Proceedings in Applied Mathematics*, pages 156–170. SIAM, January 2006.

[27] David Kaufman and Robert L. Smith. Fastest paths in time-dependent networks for intelligent vehicle-highway systems application. *Journal of Intelligent Transportation Systems Technology, Planning, and Operations*, 1993.

[28] Myeongki Seong Kiseok Sung, Michael G. H. Bell and Soondal Park. Shortest paths in a network with time-dependent flow speeds. *European Journal of Operational Research, Volume 121, Number 1*, pages 32–39(8), 2000.

[29] BOOST C++ Libraries. `http://www.boost.org/`.

[30] Jens Maue, Peter Sanders, and Domagoj Matijevic. Goal directed shortest path queries using precomputed cluster distances. In *WEA*, pages 316–327, 2006.

[31] Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. Partitioning graphs to speedup dijkstra's algorithm. *J. Exp. Algorithmics*, 11:2.8, 2006.

[32] L. F. Muller and M. Zachariasen. Fast and Compact Oracles for Approximate Distances in Planar Graphs. In *Proceedings of the 15th European Symposium on Algorithms, Lecture Notes in Computer Science 4698*, pages 657–668, 2007.

[33] Giacomo Nannicini, Philippe Baptiste, Gilles Barbier, Daniel Krob, and Leo Liberti. Fast paths in large-scale dynamic road networks, June 27 2007. Comment: 12 pages, 4 figures.

[34] Road Directorate Danish Ministry of Transport and Energy. `http://www.vejdirektoratet.dk`.

[35] Peter Sanders and Dominik Schultes. Highway hierarchies hasten exact shortest path queries. In *ESA: Annual European Symposium on Algorithms*, 2005.

[36] Peter Sanders and Dominik Schultes. Engineering highway hierarchies. In Yossi Azar and Thomas Erlebach, editors, *Algorithms - ESA 2006, 14th Annual European Symposium, Zurich, Switzerland, September 11-13, 2006, Proceedings*, volume 4168 of *Lecture Notes in Computer Science*, pages 804–816. Springer, 2006.

[37] Peter Sanders and Dominik Schultes. Robust, almost constant time shortest-path queries in road networks. In *14 th European Symposium on Algorithms (ESA 06)*, pages 804–816, 2006.

[38] Peter Sanders and Dominik Schultes. Engineering fast route planning algorithms. In Camil Demetrescu, editor, *WEA*, volume 4525 of *Lecture Notes in Computer Science*, pages 23–36. Springer, 2007.

[39] Dominik Schultes and Peter Sanders. Dynamic highway-node routing. In Camil Demetrescu, editor, *Experimental Algorithms, 6th International Workshop, WEA 2007, Rome, Italy, June 6-8, 2007, Proceedings*, volume 4525 of *Lecture Notes in Computer Science*, pages 66–79. Springer, 2007.

[40] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The boost graph library: user guide and reference manual.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[41] Mikkel Thorup. Compact oracles for reachability and approximate distances in planar digraphs. In *FOCS*, pages 242–251, 2001.