Highway Node Routing: increasing flexibility and putting it into practice

Jeroen van Wolffelaar

MSc Thesis

June 8, 2010

INF/SCR-09-82

Universiteit Utrecht

ORTEC
PROFESSIONALS IN PLANNING

Applied Computer Science
Dept. of Information and Computing Sciences
Utrecht University
Utrecht, the Netherlands

Supervisors:
Dr. J.A. Hoogeveen
Dr. J.A.S. Gromicho

Contents

1 Introduction **5**

1.1 ORTEC 5
1.2 History 6
1.3 The problem 7
1.4 Recent developments 8
1.5 Objective and central question 10
1.6 Subquestions and outline 10
1.7 Outline 11

2 Path finding, HH, HNR **12**

2.1 Definitions 12
2.2 Pathfinding 13
2.3 Highway Hierarchies 14
2.4 Highway Node Routing 15
2.5 Many to Many 17
2.6 Dynamic Highway Node Routing 18
2.7 Existing implementation 18

3 Interfacing Highway Node Routing **19**

3.1 Current situation 19
3.2 Current performance 21
3.3 Route queries with HNR deployed 21
3.4 Preprocessing management 22
3.5 Conclusion 23

4 ORTEC network input **24**

4.1 ORTEC map format 25
4.2 Input routine 26
4.3 NodeMapper 27

5 Multi-value path dimensions **29**

5.1 Discussion 29
5.2 Biassed speeds 30

6 Sharing ground network **32**

6.1 Introduction and motivation 32
6.2 Representing the ground network 33
6.2.1 Implementation 33
6.2.2 Getting traveltime 34
6.3 Overlay graphs 36

6.4 Putting the bits together . . . 37
6.4.1 Getting cost . . . 38
6.4.2 Code listing . . . 40

7 Lower memory usage (peaks) 41
7.1 Data structures . . . 41
7.2 Misc . . . 42
7.3 Utility: CompactStruct . . . 43
7.4 Utility: BigVector . . . 45

8 HNR and forbidden areas 48
8.1 The situation . . . 48
8.2 Naive Implementation . . . 48
8.3 A more workable strategy . . . 49
8.4 Multiple components . . . 50
8.5 Consequences for the implementation . . . 51

9 Experiments and results 52
9.1 Methodology . . . 52
9.2 Time . . . 52

10 Conclusion 54
10.1 Objective and central question . . . 54
10.2 Subquestions and outline . . . 55

A Bibliography 56

Preface

Finally. That is in one word, the feeling I have right now, when this thesis was finished. But now, I am satisfied with the result.

In March 2008, I started on this project at ORTEC, in Gouda. Only a few hours after the official start, Dominik Schultes of the University of Karlsruhe, and the main author of Highway Node Routing, also arrived for a short visit. With this flying start, I began my journey in the world of routing algorithms.

Fast forward to June 2010. This is the final version of the thesis, and a talk has been delivered on the subject.

First, I'd like to extend my gratitude to Han Hoogeveen, who had a lot of patience with me. We had some interesting discussions about possible solutions to the various questions that arose in the process, and he helped me with the structure of the thesis, et cetera.

I would like to thank ORTEC for providing me with this unique opportunity. There are not many places outside of the academic world where one could work on this kind of graph algorithms. I wanted to take a look outside of the academic world, to not just develop really interesting software, and think of interesting ways to solve difficult problems, but also seeing how it is going to be used in the "real" world.

I enjoyed being able to work together with people with a passion for algorithms: my colleagues at the algorithmic department of ORTEC's software development division. Especially I want to thank my supervisor and the algorithm czar at ORTEC: Joaquim Gromicho. Joaquim, thanks for your persistence in being my supervisor, it took a lot of lunches...

I'd also like to thank the many other ORTEC colleagues for their help and support: Jelke, Gerhard, Oscar, and others: thanks for your assistance and moral support. Finally, the colleagues of the GIS team have always been very helpful in achieving the integration, and teaching me about the ORTEC map format et cetera. Gerben, Szabi and Huib: thanks!

I would also like to thank those who supported me throughout this project. At times it was very difficult for me to keep being motivated in order to finish of writing down my findings in a manner that is understandable to more than just myself. In order of nagging the most: my parents, Eefje and the colleagues that I already mentioned. Especially Eefje provided me with textual support, and helped to achieve a clear structure. And to my father: thanks for your genuine interest in the subject.

I hope you enjoy reading this thesis.

Jeroen van Wolffelaar; Utrecht, June 2010

Summary

Software for logistic planning needs to have fast access to exact travel-distances and -times over the road. Dijkstra's algorithm becomes too slow on such large networks, and pre-computing and caching take too much storage. Highway Node Routing promises consistent high performance for road networks.

ORTEC aquired the implementation of Highway Node Routing created by Dominik Schultes et al of the University of Karlsruhe. This implementation was used in demonstrating the viability of HNR, and to determine actual performance results. However, the implementation cannot be used as-is in the context of ORTEC's logistics software.

The central question in this thesis is: How can Highway Node Routing (HNR) be integrated, implemented and adapted for optimal use by scheduling software?

To answer this question, the HNR algorithm has been used as the foundation for an alternative routing system for ORTEC, an ICT consultancy company in the Netherlands. The following issues were adressed:

1. **Integration** HNR has been made suitable for use as component in ORTEC ComTec architecture: required interfaces were added and the ORTEC map format was linked to the implementation.

2. **Implementation** Multiple path dimensions are now supported during calculation. HNR is made to be more memory efficient, amongst others by sharing data among HNR overlay graph instances.

3. **Adaptation** HNR is made to work sensibly with forbidden areas. Also, some research has been done to prepare HNR for dynamic queries with respect to extra restrictions.

As a result, HNR is now a full replacement of the former routing system at ORTEC, including most if not all existing functionality. HNR is 100 to 1000 times faster than the former routing system, while preserving its accuracy. It is now possible to support more vehicle profiles simultaneously with less powerful hardware. It has been made easier to make future improvements regarding queries with small additional restrictions.

1 Introduction

1.1 ORTEC

ORTEC B.V. is a consultancy company located in Gouda, The Netherlands. It was founded in 1981 as a provider of expertise and Operations Research tools. These tools are so sophisticated that ORTEC builds them itself, and therefore ORTEC includes an important software division. Today, ORTEC develops and sells software to transport and distribution companies of many sizes. Customers include DHL and Maersk, but also Coca Cola and Tesco (largest British retailer), who use ORTEC's products for their own distribution networks.

One of the most used software solutions of ORTEC are those for "Vehicle Routing & Dispatch", with the application suites ORTEC Transport & Distribution, and ORTEC Shortrec. Using this software, a team of planners can create a schedule which brings together trucks, trailers and drivers with actual orders that need to be brought from A to B. Many constraints need to be satisfied, including working hours of drivers, ample time for loading and unloading, the size and capacity of trucks and trailers, et cetera. Also emerging environmental concerns, such as limiting CO_2 emissions, are included in the solutions offered. Using scheduling software makes it easier to create a schedule that can be realized; the software can check that all constraints are met. In addition to this manual planning process, it is possible to let the computer generate (a part of) a schedule. This component, internally known as "COPS VRP", tries to make an optimal solution for a sizable number of transport orders and trucks/drivers.

For calculating the timing of the actions in the schedule it is required to know how long it takes to travel among the addresses in the schedule: if a planner decides that a truck should bring some orders from A to B, the schedule needs to determine at what time the truck will arrive at B. We call this a one-to-one or point-to-point query.

In order for COPS VRP to suggest a part of the schedule, we need even more: the input to the optimisation algorithm consists of some set of orders, and there is no a priori route to pickup and deliver those orders. COPS VRP cannot try to create a suggestion before it knows the travel time among each of the addresses: it needs a complete matrix of travel times. We call this a many-to-many query.

To conclude, the ORTEC software is able to compute both one-to-one- and many-to-many-queries. One-to-one queries (and one-to-many queries) are computed by using the Dijkstra algorithm. Many-to-many queries are indirectly solved by Dijkstra: a caching component uses Dijkstra to complement the schedule for every new one-to-many query, and the schedule as a whole is used to solve

a many-to-many query. ORTEC only uses the one-to-many query to compute many-to-many queries, because the latter cannot be solved directly by the Dijkstra algorithm. In the next subsection, Dijkstra's algorithm will be described, as well as attempts for enhanced ways to solve the shortest path problem.

1.2 History

The problem of optimally solving routing queries has received attention for at least 50 years. Edsger Wiebe Dijkstra was the first to publish[Dij59] a comprehensive algorithm for solving general routing queries, in 1959, better known as the shortest path problem in graphs. The mathematical notion of a graph is typically used to model a real-life road network in the computer, and is also the model which will be used throughout this thesis. Dijkstra's algorithm is able to solve the following question: given two points A and B in a graph with non-negative edge lengths, what is the shortest (or fastest, or generally: best) path to get from point A to point B? The algorithm involves scanning all points around the query points that are close by. For a large distance, this can involve a substantial part of the road network, and therefore this can easily take 1-2 seconds per query even on modern hardware. An interesting side effect is that the same effort leads to one-to-many solutions.

Floyd-Warshall[Flo62] and Johnson[Joh77] have published algorithms to solve the all-pairs shortest path algorithm: how to compute a complete distance matrix in a graph. If it were possible to calculate this in acceptable time, and it were possible to store this matrix (table) on disk, this would solve the routing problem. However, it can easily be shown that the storage requirement of this table is prohibitively large even for moderately sized graphs with a million nodes, such as a map of the Netherlands, let along a map of Europe.

Another approach would be to exploit certain properties of the road network graph: for example, sparsity, the spatial properties (often coordinates are available), and the fact that generally only few of the road segments are usable for long-haul stretches. The $A*$ algorithm[HNR68] is the archetypical example of this class: using geographic coordinates and a heuristic function exploiting the fact that a route via the road cannot be shorter than the straight line distance. In practice, however, $A*$ is at best only a few times faster than Dijkstra's algorithm, and offers no practical advantage for many-to-many queries.

The above mentioned examples are the most famous ones. More examples can be given, but these are not generally known and only usable in specific situations. None of these specific examples offers a promising alternative for Dijkstra, at least not for ORTEC.

1.3 The problem

Traditionally, Dijkstra's algorithm[Dij59] is used to calculate travel times, for both the one-to-one and (via repeated application) the many-to-many queries. Invented more than 50 years ago, this algorithm yields optimal results, within a very predictable timespan.

The problem is that this timespan is too long for many of ORTEC's software products. On the one hand, maps are getting more and more detailed, and Dijkstra's algorithm performs relative to the amount of nodes and edges (road segments). On top of that, Dijkstra's algorithm can only perform a one-to-many or many-to-one query (the latter when reversing the edge direction logic), not directly a many-to-many. In order to fill a matrix with travel times, Dijkstra's algorithm should be performed as many times as there are rows (or columns) in the matrix.

To give an indication, on nowadays' hardware, a Dijkstra search on a reasonably detailed map of Europe takes one to two seconds. A 1000 by 1000 query would therefore take more than 15 minutes, a 10 000 by 10 000 query more than a day.

To alleviate this, ORTEC introduced a caching service in front of the routing services. All routing queries are directed to this service, instead of directly to the actual routing services. The caching service remembers for some set of addresses the travel times among each of these addresses. The storage requirement for this is obviously quadratic, so the amount of addresses in this caching service is limited. The scheduling and optimisation services ask the caching service for travel times. The caching service composes its answer in two phases:

First, it will ensure that all addresses in the query are cached. If not, the caching service will ask the routing service to use Dijkstra's algorithm twice for each address: once to calculate the travel time and distance from each of the present addresses to the missing address, and once to calculate the travel time and distance from the missing address to each of the already present addresses. These travel times and distances are then stored in the cache. Note that the underlaying network is directed and generally not symmetric, therefore, the double query for row and column.

Second, it will return the desired travel times and distances by simply looking up all the desired entries in the cache.

The second phase is very quick, the execution time is dominated by communication overhead. The first phase, however, depends greatly on the amount of addresses that were not previously known to the caching service: each missing address takes a few seconds. If a batch of new orders is added, and the optimisation service is asked to schedule those, this might be very fast (if all addresses

were already present in earlier orders), or take a lot of time (if a lot of addresses have never been seen before).

The central problem with the current routing solution at ORTEC is that it is sometimes too slow. An additional problem is that the present solution consumes too much memory (both in RAM and on disk), in addition, cache consistency in the presence of changes to any of the calculation parameters is difficult to guarantee.

1.4 Recent developments

So far, we have seen three approaches in addressing the routing problem: full calculation (using Dijkstra's algorithm), pre-computing all possible results (and caching it), and using particular features of the road network graph. The first solution is too slow, especially if many results (for example, 1000 by 1000 addresses) are needed. The second is infeasible due to the amount of computing power to (pre-)calculate, and the amount of storage required. Also knowing which addresses to pre-compute is not always possible up-front. This is the case in dispatching routing applications such as taxis. Several algorithms in the third class of approaches are significant improvements to Dijkstra's algorithm, but none are as fast as is required for an adequate many-to-many query algorithm.

In the remainder of this section, we will be looking for a solution that constitutes a combination of these three general approaches.

Much research has been done to algorithms which combine exploiting certain network properties with a certain amount of pre-calculation to speed up queries. ALT[DSSW06], REAL[GKW06], Component Hierarchies (see below) and many more are results of such research. See [Sch08, Related work] for a more complete overview of such work.

In 2007, Anne de Koning investigated Component Hierarchies[dK, Tho99] at ORTEC. Thorup et al observed that the reason for Dijkstra's algorithm to have a above-linear complexity in the number of nodes and edges, is related to the necessity to repeatedly find the node with the least distance so far, and only then to settle it. Indeed, to get Dijkstra to perform best, many researchers have looked into the best way to implement such a so-called *delete-min* operation on a set of nodes. The best known implementation so far is the Fibonacci-heap[FT87], with which Dijkstra's algorithm has a complexity of $O(m + n \log n)$. Thorup suggests a way to remove this requirement: by grouping edges into components, it is no longer absolutely necessary to process nodes strictly in priority queue order, reducing the amount of delete-min operations. This grouping can be done in time linear to the number of edges. One can then replace the delete-

min operations by some mechanism based on bucket sorting. Putting these two together, this means that Component Hierarchies is the first algorithm to reach the theoretical algorithmic complexity of the shortest path problem, with as only additional requirement integer edge lengths. The complexity is simply $O(n + m)$[Tho99, Theorem 1][1].

Unfortunately, this algorithm is not usable for many-to-many queries, and therefore not viable to be used at ORTEC. The algorithm also does not provide obvious opportunities for future flexibility and ad-hoc calculations with small modifications to the graph. In addition, the set of implementations by Dominik Schultes et al[Sch08] show that even using a regular heap can exhibit record-breaking performance if search space sizes are minimized, thereby reducing the number of heap operations done (as opposed to merely making them faster).

At the University of Karlsruhe a suite of algorithms has been developed for routing in road networks. *Highway Hierarchies* is the original algorithm, exploiting the hierarchical nature of road networks in such a way that after some pre-computations, queries can be done very fast without compromising correctness.

Highway Node Routing was developed later as a more flexible variation. It has the unique ability to be locally pre-processed again upon minor changes in the network, and in general features fast pre-processing for various cost functions once a graph has been pre-processed for some generic cost function.

Transit Node Routing is yet another variation, with larger pre-processing times and memory consumption, no flexibility, but a record breaking performance. This algorithm leads to winning the 9th DIMACS challenge on Shortest Path algorithms in 2006.

All these algorithms have in common that they allow for an efficient form of many-to-many queries. They combine a limited search space and bi-directional search, allowing a matrix with times and distances to be filled in time roughly proportional to the number of sources and targets in the many-to-many query. If the number of sources is s and the number of targets is t, then obviously the theoretical complexity is at least $O(st)$, the size of the output, but the calculation time is dominated by the forward- and backward searches, which takes merely $O(s + t)$. The $O(st)$ phase (overlapping the search spaces) is negligible.

The flexibility present in Highway Node Routing makes it a good candidate to consider replacing the ORTEC routing service with. However, it cannot be used as-is. The reference implementation created by the University of Karlsruhe needs to be converted into a service and it must be taught to understand

[1]assuming all edgelenghts e_w to be $0 \leq e_w \leq 2^w - 1$ and $n \leq 2^w - 1$ where w is the number of bits in a machine word

the map format in use at ORTEC. Also interfaces need to be added for other subsystems to use it. Other required improvements include: memory usage, especially sharing memory among multiple vehicle profiles and returning more than one dimension of the path 'length'. Also the routes to be returned need to be better: road restrictions (forbidden areas) must become soft restrictions, in order to be able to get reasonable approximations around pedestrian areas forbidden for vehicles such as shopping centers.

1.5 Objective and central question

Ideally, ORTEC would have a service that can answer one-to-one and many-to-many queries across the map in a consistently fast manner, using Highway Node Routing, the most promising algorithm today for road network routing.

The central question in this thesis will be: *How can Highway Node Routing be integrated, implemented and adapted for optimal use by scheduling software?*

1.6 Subquestions and outline

To answer the central question the following subquestions are formulated:

Integration

- How can the interfaces be added that are required for ORTEC's ComTec framework? How can the implementation be integrated so that it can be hosted in a long-running process?
- How can the ORTEC map format be interpreted by the HNR implementation?

Implementation

- How can multiple edge-length dimensions be returned, even for many-to-many queries?
- How can the implementation be made more memory efficient, amongst other by sharing data for various vehicle profiles?

Adaptations of the HNR algorithm

- How can HNR be made to work sensibly with forbidden areas?
- How can HNR be prepared for dynamic queries with respect to extra restricions?

1.7 Outline

In chapter 2 we will explain how Highway Node Routing (HNR) works, and what improvements and adaptations are still necessary and possible.

ORTEC acquired the implementation by Karlsruhe University. In chapter 3 we will discuss how this implementation was integrated: how the HNR implementation was modified to be embedded in the framework used at ORTEC for their products. In chapter 4 we will discuss how the implementation was adopted to deal with ORTEC's map format.

Then, several chapters on the implementation of the HNR algorithm: In chapter 5 a solution is presented on retrieving both traveltime and distance of the calculated routes, even for many-to-many queries. In chapter 6 a split in the main graph datastructure is discussed, allowing for a much more memory efficient situation. In chapter 7 some more memory-saving techniques are discussed.

In chapter 8 we discuss an addition to the HNR algorithm required for routing with forbidden areas. The problem of preparing HNR for dynamic queries, with respect to extra restrictions, is not discussed in this chapter. However, throughout other chapters, this problem will be described.

We conclude with some experimental results in chapter 9, and the conclusion in chapter 10.

2 Path finding, HH, HNR

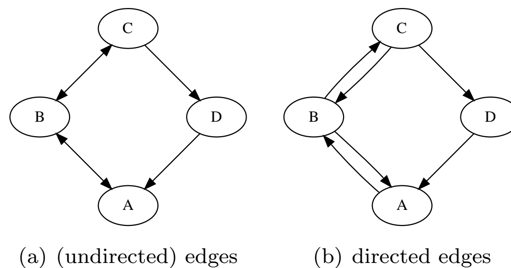
The key issue in this thesis is finding the best path in road networks, and returning the total distance and travel time of the path, and optionally the actual path itself too as a list of nodes. A *road network* is a collection of locations, and public roads connecting those locations. We can easily map the real world concept of a road network to the mathematical concept of a *graph*. Locations become nodes, and road segments become (directed) edges connecting those nodes. In the case of complicated intersections, one can introduce multiple nodes for each intersection and as many directed edges as needed, to fully express any turning restriction.

In graph theory, a *path* in a graph is a sequence of consecutive edges. The *cost* or length of a path is the sum of the costs of the edges it consists of. In this thesis, we will use the term (edge- or path-)cost. A shortest path between nodes u and v is a valid path such that there does not exist another path which has a lower cost. Note that a shortest path in graph theory does not necessarily refer to the path with the minimum travel distance. In real life applications, cost is typically a function of the distance and travel time, and this is also the case in this thesis.

2.1 Definitions

A *graph*, typically denoted with G or H, is a set of nodes and (directed) edges.

While representing a graph in a data structure, there are several ways to express the connections amongst the nodes.



(a) (undirected) edges (b) directed edges

Figure 1: Example graph

Let's consider an example graph, as drawn in two different ways in Figure 1. In the undirected edge view 1(a), there are 4 (undirected) edges, of which two are completely open, and two (CD and AD) are half-open.

In the directed view 1(b) there are 6 directed edges (sometimes also called arcs). For straightforward, regular pathfinding, like with Dijkstra, only outgoing

edges for a given node are of interest, so a representation with 6 entries for (outgoing) arcs associated with nodes will do. For a backward search from target to source, on the other hand, the opposite representation is required,

node	outgoing arcs
A	$\{B\}$
B	$\{A, C\}$
C	$\{B, D\}$
D	$\{A\}$

(a) Forward

node	incoming arcs
A	$\{B, D\}$
B	$\{A, C\}$
C	$\{B\}$
D	$\{C\}$

(b) Backward

Figure 2: Arc list representation of a graph

However, most connections between nodes have no direction limitation: one way roads and highways are rare compared to the vast amount of bidirectional city and rural roads. Because of this asymmetry, a more memory efficient approach would be to store every (undirected) edge only twice, once for each endpoint, along with two boolean flags to indicate the direction restrictions.

node	edges
A	$\{B^{\leftrightarrow}, D^{\leftarrow}\}$
B	$\{A^{\leftrightarrow}, C^{\leftrightarrow}\}$
C	$\{B^{\leftrightarrow}, D^{\rightarrow}\}$
D	$\{C^{\leftarrow}, A^{\rightarrow}\}$

Using this representation, 8 entries in the edge list are required, instead of the 12 otherwise needed for simultaneously storing the forward and backward versions.

When talking about the size of a graph, we will use the number of (undirected) edges. So the graph in Figure 1 has 4 edges (and therefore, typically requires 8 entries to store all edges, if both forward and backward traversal are desired).

2.2 Pathfinding

The problem of finding a shortest path in a graph has been covered extensively for about 50 years. In 1959, Dijkstra published[Dij59] the de-facto standard algorithm for shortest path calculations in a general graph. Running time for Dijkstra's algorithm is close to the theoretical minimum running time of $O(E)$, where E is the number of edges (if not all edges are visited, there might exist a shorter path involving one of those edges). Therefore, finding a shortest path between two given nodes is possible in polynomial time, close to linear time in the number of edges. However, in real-world scenarios, graphs representing

road networks can easily have tens of millions of edges, and even on the fastest computers, a query can take around a full second, or several seconds on more common hardware.

Road networks are continuously changing: new roads are built, existing roads are upgraded or sometimes closed. However, these changes take a long time, and on the scale of a day or week, we can assume a road network to be static. More specifically, updates to the road network are provided batch wise by road information suppliers, and updates are rolled out to customers periodically. This allows for a strategy where initially some time is spent building auxiliary data structures for a given map, which then later can be used for faster queries. As an extreme example, a complete distance table could be calculated and stored, for near instant 'calculation' of routes. However, storage requirements would be prohibitive except for in the smallest of maps. Furthermore, extracting the path as the list of traversed edges would not profit from this caching.

Two techniques (amongst others) were devised by Dominik Schultes et al of Karlsruhe University: *Highway Hierarchies* and *Highway Node Routing*. Both techniques rely on an algorithm to create a relatively compact preprocessed data structure, and an algorithm to do point to point shortest path queries, exploiting this auxiliary data structure. Also many to many queries can be performed with the same data structure. We stress that these algorithms are optimal, deterministic algorithms, without compromising exactness or correctness. Highway Node Routing is the main subject of this thesis, Highway Hierarchies are merely used as a necessary preprocessing step. The next section can therefore be skipped by the impatient reader.

2.3 Highway Hierarchies

For a more complete discussion of Highway Hierarchies, see [SS05], [Sch05] and [Sch08]. Highway Hierarchies is not used in this thesis, this description merely serves as an introduction to hierarchical shortest path algorithms, and as such describes preliminaries for Highway Node Routing

Highway Hierarchies (HH) is based on the key observation in the previous section that it pays to spend time in advance in some preprocessing step where an auxiliary data structure is created. HH exploits some specific properties of graphs representing road networks, not generally present in graphs. Road graphs are generally very sparse, mostly planar, and have a hierarchical nature. This means we can try to come up with preprocessed data structures which are expected to perform best under such conditions (but are correct in all cases).

Let $G = (V, E)$ be a directed graph, modelling a road network. The key observation with Highway Hierarchies is that comparatively few edges occur in

the middle of any long-haul shortest path. To put it the other way around: most of the edges do not occur in the middle section of any shortest path.

Definition 1 *The neighbourhood of size N of some node u consists of the N nodes closest to u. The edges in the search tree finding these nodes are called the neighbourhood edges.*

In a directed graph, there is not just one neighbourhood for each node, but two: the *forward neighbourhood* are the nodes closest when following the edges in their regular configuration. The "backward neighbourhood" are the closes nodes when doing a backward search: interpreting the directed edges with inverse direction, corresponding to a search started from the target node, instead of the source node.

Definition 2 *The middle section of a path is the set of edges on the path that are neither in the forward neighbourhood of the source, nor in the backward neighbourhood of the target.*

With "middle section" defined, we can define the reduced set of edges. We call this set of edges "highway edges".

Definition 3 *Highway edges: Given a graph G, the set of highway edges is the union of middle sections of the shortest routes of any pair of nodes.* $E_h = \bigcup_{s \in V} \bigcup_{t \in V} middlesection(shortestpath(s,t))$

The highway edges form a (possibly unconnected) graph that is typically much sparser than the original graph, but seldom empty. The same procedure can be repeated, thus getting multiple layers of graphs, increasingly sparse.

When searching in the original graph, one can now speed up the search considerably by relying on the definition of these highway graphs: outside of the neighbourhood of the source node, we know that all relevant edges are present in the next level, and therefore only need to consider edges in that sparser graph. Applying the same algorithm backwards, from the target node until both half-queries meet, yields a fast, hierarchical, query algorithm.

2.4 Highway Node Routing

In contrast to the previous section on Highway Hierarchies, in this section we will present some theorems which underline the workings of Highway Node Routing (HNR), because, unlike HH, HNR is used directly and as a base in this thesis.

Highway Node Routing[Sch08] is based on the same premisses as Highway Hierarchies, namely on the observation that in the middle of long haul routes comparatively few edges can occur.

A key assumption for Highway Hierarchies is that the basegraph is constant, and not changing. However, for certain applications this is not true. Examples include:

- We want calculations to be done for different vehicle types, for example, both vans and big trucks. Both vehicles have a different speed profile, and hence different traverse times and hence edge cost associated for nearly all edges. The only solution would be to completely duplicate the datastructures, which whould have a considerable memory usage impact.
- Some road segments are forbidden based on the loading of the vehicle: some road are inaccessible when hazardous materials are transported, when a truck is loaded above a certain tonnage or height, when a trailer is present, or any combination thereof.

With Highway Hierarchies, such changes involve major structural changes to the hierarchies, which cannot easily be locally updated.

A solution to this problem is Highway Node Routing. Instead of having a predefined neighbourhood, and constructing the hierarchies based on this definition, we work the other way around: the nodes that are to be part of the hierarchies are fixed in advance, and what is considered the neighbourhood is determined from this input.

For this to work, three things need to be considered:

1. How to determine what nodes span the hierarchy?
2. How to execute the concept of neighbourhood? More precisely formulated, how to determine when to elevate to the next hierarchy level?
3. How to obtain a data structure for a higher hierarchical level where searching should be faster than in the base graph?

Ad 1, what nodes are to be part of higher hierarchies is assumed to be given. As for correctness of HNR, no assumption is made on this subset, even the empty set or the complete set of nodes would yield a correct algorithm, albeit an algorithm that can degrade to Dijkstra performance. The selection of sensible subsets is, however, of significant impact to the performance of HNR. Highway Hierarchies with an average vehicle profile shows to be an adequate method, but research on other methods is coarse.

Ad 2, the concept of neighbourhood. In contrast to Highway Hierarchies, where the hierarchy was carefully constructed to allow for a simple elevation metric, we now assume nothing at all. Instead, the search continues on a higher hierarchical level precisely when the search encounters a node that belongs to

such higher hierarchy. Other branches of the search tree continue in their old level. The search is only completely elevated to a next level once the complete search tree is covered by those higher level nodes. In order to prevent this search tree from growing too much, the search tree is pruned by a technique called stalling – which will not be further detailed here as it is out of scope.

Ad 3, the higher hierarchy data structure. Once the search is elevated to a higher level, like in HH, we would like search faster. For this to work, we are looking for a data structure with which we can answer the question: "How to get from A to B fastest" faster, but no less accurate, than when we would use the original graph. One very fast way would be storing a full distance table, but for any sizeable set of nodes, the $O(n^2)$ storage requirement is prohibitive. In general, we are looking for a graph, dubbed overlay graph, which preserves shortest paths. A complete graph representing the distance table is a valid overlay graph, but a sparser graph yields a better trade-off.

The preprocessing for HNR now consists of creating those overlay graphs, given the base graph and given the hierarchy.

To construct an overlay graph, a search is started from each node v in the overlay. Once the search tree is covered by nodes in the overlay, the seardch is stopped, and for each covered node u that is also in the set of overlay graph nodes, an edge (v, u) is added to the overlay graph under construction.

This process is considerably faster than the HH process, because only from overlay nodes a search is started, not from all nodes. This is possible thanks to the hierarchy already given. Of course, generating the hierarchy itself was needed too, but that could be done once for the graph, regardless of vehicle profile.

Moreover, this process is more locally independent, because the hierarchy never changes, and edges in the overlay graph need to change only if the corresponding shortest path changed. So, minor changes to the base graph would not cause many escalating changes in the overlay graph.

2.5 Many to Many

When attempting to optimise a delivery schedule, it is required to have full matrices with distances and times of some set of nodes $S \times T$. Instead of doing $|S| \times |T|$ point to point queries, the reduced search space of HNR allows us to do better.

Recall that the query process for HNR involves searching from both sides, until the search spaces overlap. Even if a search is not aborted when such overlap is detected, but continued until no more nodes can be found, the search space will be comparatively small: when higher level nodes are encountered, the

search is only continued on that level, and so on.

2.6 Dynamic Highway Node Routing

Faced with the issue of a small number of changes to a graph, two solutions are noted in the PhD thesis of Schultes: the server scenario and the mobile scenario.

The server scenario basically involves destructively modifying the overlay graph data structure based on the new information to the point where it becomes identical to the overlay graph that would be obtained if it were directly constructed. This is done in a "redo every step of the construction that has seen a bit of information that's now (possibly) changed". Only parts of the construction which can be shown to not rely on changed data in the graph are preserved. To aid in this effort, a data structure involving affected node sets is used to find out what nodes are susceptible to change in such a scenario.

The mobile scenario involves no changes to the overlay graph at all. Instead those areas in the overlay graph that would be revisited if the server scenario would be in use, are marked and ignored (as if the set of nodes for a particular level were smaller).

For neither technique correctness is formally proven. Moreover, the mobile scenario technique is conflicting with the many to many technique, there is nothing published yet about this issue or about techniques to make this possible.

2.7 Existing implementation

As part of the research at the University of Karlsruhe, Highway Node Routing (amongst others) has been implemented as a proof of concept. The HNR implementation was done in the C++ programming language, as a command line executable on a Linux system. It could run a fixed set of commands based on a configuration file. This is adequate for a proof of concept, but lacks key functionality when HNR is to be integrated in a larger software package, when its functionality needs to be available as a service.

We adapted the code to work on the windows operating system. In addition, an interface was added: a shared library using which code can be run continuously and hooked in other subsystems. It supports loading a graph from an ORTEC network file, returning a handle. Using this handle, any number of point-to-point and many-to-many queries can be performed.

3 Interfacing Highway Node Routing

ORTEC acquired an implementation of Highway Node Routing (HNR) from the University of Karlsruhe. However, just having such an implementation is not enough. In order to be able to deploy HNR successfully, the code needs to be *integrated*.

There are basically two sides to the integration. On the one hand, there are the components that require the services of HNR: knowing distances and times among certain nodes, sometimes many of them as a matrix, sometimes also returning the actual shortest path. On the flip side there is the data that HNR requires to function: map data, including per-edge information on what classes of vehicles are allowed there, and also average speed information for the vehicles that are supported.In this chapter we will focus on the former side of the integration (interfaces), while in chapter 4 the latter side (map data) will be covered.

In this chapter we will explore what was needed to do the integration, and what choices were made, including:

- Making the code compile and function correctly on the operating system used at ORTEC (Microsoft Windows)
- Making the functionality available via a shared library, and eventually via the remote procedure call mechanism of choice
- Manage the lifetime of all the various data structures: from the distribution of nodes across level as a result of preprocessing, to the reuse of priority queues in memory
- Manage availability of HNR to the outside world: triggering preprocessing when needed, and making sure routing services are available in the same way as in the pre-HNR situation.

3.1 Current situation

ORTEC acquired the Highway Node Routing implementation for using it in its Logistics Suite (OLS). Routing and route information is an integral part of OLS.

The framework used for OLS is called ComTec. In the ComTec framework, most software components are spread out across multiple services. This allows for good separation of responsibilities, and also allows scalability: services can be spread over multiple servers, and some type of services can employ load balancing. The map and routing functionality is likewise distributed across several

services. Examples include visualisation (drawing maps), geocoding (translating coordinates and addresses into node ids) and a network data provider which can give all sorts of information about the map, including path list retrieval.

The actual routing functionality is in “ShortrouteCalculator”, and there is a different service “ShortrouteMatrix” which can serve as a cache in front of ShortrouteCalculator.

Naturally, there needs to be some form of communication among those software components. Multiple protocols are in use, but for the Shortroute services an XML *SOAP*[2] interface is used. SOAP is a protocol for Remote Procedure Calls (RPC), a concept where named functions are called with some arguments and return value via a network protocol. In the case of SOAP, the network protocol is HTTP on top of TCP/IP. The caller and callee can be on the same host (by using localhost addressing), but can also be on distinct hosts. The SOAP protocol will encode the function call arguments in an XML document, send it over HTTP to the callee (or *SOAP server*). The server will then reply with an XML document representing the function call status and if successful, the return values.

Both the ShortrouteCalculator and the ShortrouteMatrix service provide a SOAP interface. Services that want time/distance information for routes, typically use the caching ShortrouteMatrix, even for one-to-one queries. Services, such as the map GUI, that want to know the actual path from some node A to some node B use ShortrouteCalculator, as that is the only service of the two that can provide this information. ShortrouteMatrix itself cannot calculate anything, it will ask ShortrouteCalculator for all information that is not available in its cache. See Figure 3 for an impression of how the services interact.

ORTEC already has programmatic interfaces to an existing routing component. There are actually three different components providing those services, each with a different focus. One problem is that the existing architecture was made on the assumption of Dijkstra-like routing algorithm: $1 \times N$ and $N \times 1$ calculations are basic steps, and not notably slower than just 1×1, and on the other hand, doing a $N \times M$ with $N, M > 1$ cannot be done faster than multiple single-source/single-target calculations. With HNR, the performance properties

[2] `http://en.wikipedia.org/wiki/SOAP`

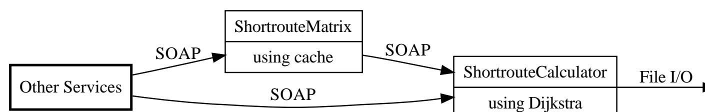


Figure 3: High-level call graph of shortroute services

are different, prompting for a reconsideration of the existing distribution of the functionality. However, in order to keep deployment as painless as possible, changes visible to the outside should be kept to a minimum.

3.2 Current performance

Customers of ORTEC get increasingly global, and they want to plan and optimise transports in large geographic areas. And on the flip side, it would also be beneficial to ORTEC if fewer maps needed to be supported. In the current situation, many maps of Europe are supported, each with a different subset of countries, but none with all countries included. A major obstacle to rectify this situation and deploy bigger maps is the performance of the route calculation components.

ShortrouteCalculator supports one-to-one path queries for time and distance, with and without returning the actual path. This is implemented using Dijkstra's algorithm. Because Dijkstra's algorithm is target-oblivious, one to many and many to one queries (the latter using the backward graph) are also possible without significantly more calculation time. The calculation times varies, but can be a couple of seconds for long haul routes.

ShortrouteMatrix maintains a square matrix with distances and times of some 10 000 nodes: those nodes that are most recently used. Many to many queries are answered by looking up results in this matrix. The problem is in the prepare step, where ShortrouteMatrix needs to ensure that all the nodes mentioned in the query are actually in the matrix. For every node not present, it performs two RPC calls to ShortrouteCalculator, one one-to-many query to fill the new row and one many-to-one query to fill the new column. The result is that ShortrouteMatrix takes time proportional to two Dijkstra queries per node that's not yet in the cache. Especially after a configuration change which required the cache to be flushed, only 500×500 query can easily take half an hour.

For optimising schedules, such many to many queries are often needed. Such long waiting times prompted to research alternatives to the current solution.

3.3 Route queries with HNR deployed

By using HNR instead of the matrix-cache and Dijkstra services we hope to improve the expected performance considerably, although it is impossible to beat the matrix-cache solution if the matrix is completely filled with the nodes that are queried. But by removing the cache altogether, we do eliminate the possibility of cache-misses. We trade excellent best-case performance for predictable and acceptable performance.

Highway Node Routing supports efficient many-to-many queries, but only if the query is done in one go. The existing caching matrix solution queries rows and columns from ShortrouteCalculator. We decided to completely drop the caching matrix infrastructure, and create a new service that implements both the ShortrouteCalculator and the ShortrouteMatrix SOAP interfaces, based on Highway Node Routing. See Figure 4 for an illustration of the HNRCalculator setup. The actual work is divided in three components: HNRCalculator implements the SOAP server protocol, and loads `HNR.dll` to do the actual work. It will also load `NodeMapper.dll` to do the node labelling translations required, see section 4.3 for more details.

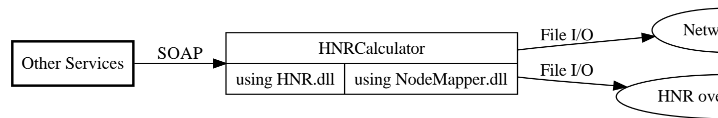


Figure 4: High-level call graph of HNRCalculator service

3.4 Preprocessing management

The HNRCalculator is the new service that provides routing services, previously provided by ShortrouteMatrix. ShortrouteMatrix had a near instantaneous startup: it only needs to open the persisent cache, and enable the interface.

For HNRCalculator, more things are required: the distribution in overlay levels by Highway Hierarchies, the loading of the nodemapper, the creation (if required) of the overlay graphs, and the loading of those overlay graphs. Only once all of these steps are performed, routing queries can be addressed. Other services within ComTec may attempt to get routing information before everything is up and running. In order to remain close to the old behaviour, we have chosen to register the routing interface immediately upon startup, and queue all incoming requests while the HNRCalculator is still starting up.

Apropos the HNRCalculator startup, the following actions are performed in order:

- All required code and shared libraries are loaded
- The configuration is parsed, and a list of profiles is retrieved from the operational database
- Interface handlers are registered for each profile

- The preprocessing state of the basegraph is checked, and if no assignment of nodes to levels has yet been made, Highway Hierarchies is invoked to create such mapping
- The nodemapper is loaded for the graph, using the Highway Hierarchies preprocessing result
- For each profile the preprocessing state is checked. If no preprocessing was done, or the preprocessing doesn't match the Highway Hierarchies preprocessing, or doesn't match the profile (anymore), it is (re)done at this time.
- The basegraph is loaded from disk
- All overlays are loaded from disk
- The interfaces are unblocked, so that all pending queries are dealt with

3.5 Conclusion

This section describes the main architecture and interface decisions taken to give HNR a place within the ComTec framework. The result obtained enables swapping the old and the new shortest path technologies on a running system and even to run them both simultaneous by taking advantage of properties of the data.

4 ORTEC network input

As a starting point for any graph algorithm, it is required to have some means of reading the graph information (structure and per-edge-properties) from permanent storage, typically some files on a file-system. There are several standardized formats for this purpose, and the Karlsruhe implementation can use one of those (Graph Description Language (GDL)[3]) as input. However, GDL is a textual format, and as such has a lot of overhead. For huge graphs like those to be used in map routing algorithms, it is ill-suited.

Furthermore, ORTEC already has a file format for route graphs, providing everything needed. Data from each map data supplier is converted into this format, so that there is a common base for the ORTEC components to work from without needing to encode supplier-specific code everywhere.

Initially, some glue code was added to the HNR implementation to read the complete map in to memory, and process it until it fits the internal representation wished for by HNR. This included a tree data structure for mapping the external NodeIds to the internal node sequence numbers used by HNR. The complete data structure was then, after the preprocessing, serialized in a big file.

This procedure proved to be quite memory intensive, furthermore, it would not scale very well in the presence of a split ground network/overlay graph (see chapter 6). In order to reduce the memory footprint, several approaches were implemented:

1. Structure and edge-information data is read directly from the ORTEC data structure
2. This input process is optimised to require fewer iterations over all data, so that is is adequately fast and not a problematic cost to incur at startup time
3. Node id mapping gains an extra file for the HNR sequence ids, and is split from the core HNR/HH code
4. The tree data structure for node mapping is replaced by new memory- and cache-friendly static hash-map

The main in-memory data structure for HNR, and how to get it smaller, is discussed in chapter 6, this chapter focusses on input time and memory, and on the node mappings.

In this chapter, we will explain how this ORTEC format looks like, and how we manage to read this format in the most efficient way possible.

[3] `http://www.aisee.com/gdl/nutshell/`

4.1 ORTEC map format

The ORTEC map format is composed of a directory with over 50 files, each representing different aspects of the road network. Most files use a binary format, for efficient processing, and quite some information is duplicated, to cover for the various access patterns (for example, there are both forward and reverse representation files for all directed edges: an edge that is open for traffic from both directions, is mentioned four times).

Only a handful of those files are of relevance to calculating shortest paths. Most files have a sequence of unsigned integers, sequentially stored in little-endian binary format. These are the `.net`-files. For example, a `.net`-file representing an array of size 3, with the integers 1, 42 (0x2A), and 2009 (0x07D9) is a 12-byte file:

```
01 00 00 00   2A 00 00 00   D9 07 00 00
```

This simple format has the advantage of random-access, and it is identical to the in-memory layout of an array on little-endian machines, such as all i386-based machines (i.e., PCs). Therefore, it can be memory-mapped when appropriate.

File	#Entries	Bytes*	Description
`org_nr.net`	N	4	External id of a node
`first.net`	$N+1$	4	Per-node index into edgelists
`naar.net`	M	4	Id of target node of an edge
`lengtelong.net`	M	4	Length in meters of edge
`type.net`	M	1	Road type id
`access.net`	M	4	Road accessibility bitmask

* per entry

Table 1: Files composing the ORTEC map format

There are two `.net`-files describing the structure of the graph, `first.net` and `naar.net`. These two files jointly represent an adjacency list, with `first.net` having at each index i, the index j in `naar.net` where the outgoing edge group of node i starts. The entry at $i+1$ represents the end (exclusive) of that edgegroup – the edgegroups are consecutive. `first.net` has 1 more entry than the number of nodes to make this work, where the last entry represents the total number of edges. `naar.net` then has the id of the target node. Every edge is directed, bidirectional edges are listed twice in these files, once for each direction. The graph is not necessarily connected.

There are three more `.net` files having per-edge properties, that are relevant for shortest path calculations. `lengtelong.net` has the length of the edge, in meters. This can be zero, sometimes one road feature is mapped to mul-

tiple nodes, if turning conditions are complicated, or if two maps are stitched together. `type.net` is an exception to the general rule that every entry in a `.net`-file is 4 bytes, it only as one byte per edge: the road type (highway, local road, etc). The road types are used to map speeds to roads, a vehicle profile has one speed for each road type. Lastly, there is `access.net`, which as a bitmap of 32 bits for each edge, giving for up to 32 classes of vehicles whether or not that vehicle class is allowed to use that road. Edges with restrictions include public transport and taxi only roads and pedestrian passages. Example vehicle classes include regular cars, trucks (the latter may not cross certain smaller bridges or city center roads), and emergency vehicles (access to all roads, except narrow pedestrian passages and so on).

`org_nr.net` is a mapping file, one entry per node, which has the external, original node id. Most ComTec components will refer only to nodes by these ids. See table 1 for a listing of all `.net`-files discussed here.

4.2 Input routine

We have reworked the input routine. Previously, all input files were read from disk, and copied and processed multiple times. The goal is to have a associated list representation in memory, where given any node, it is possible to directly find all outgoing and incoming edges. All edges therefore need to be stored twice, once for each endpoint. Two boolean flags keep track whether the data structure actually represents a forward edge, backward edge, or both.

One way of achieving more flexibility with HNR is splitting the overlay graph from the base network (see chapter 6). When HNR is consequently used, the base network will need to be read from disk separately from the overlay graph. One way would be to use the ORTEC network format directly, but then it is important that reading in the network is fast. An alternative would be to store the base network in some serialised form, but this would require additional data management.

The reworked input routine consists of the following steps:

- The edgelist with the for routing related attributes is stored in five binary files, which are all read in parallel. One file has for each node an index in the edgelist where the series of outgoing edges for that node start, thus identifying the source nodes for each edge. The other four files have for each edge the target node id, distance in meters, road category, and an access bitmask. During this single pass over the five files, the values are stored in a sequential block of memory, as sequence of edges (with source/target nodeid, and the edge properties).

- The list of edges is stored in a BigVector (see section 7.4), to eliminate memory peaks, while keeping track of the start of a group of outgoing edges.
- The backward graph is added, by scanning the list of edges, and for each edge, appending the reverse to the end of the edgelist if it wasn't there yet. This typically adds about 5% to the number of edges (implying that the vast majority of edges are two-way).
- Possibly, all sources and targets are translated according to a supplied mapping
- The graph is sorted by source,target
- Parallel edges are detected, just to be sure
- The graph is converted to some form better suited for the algorithm at hand: for Highway Node Routing, this is a compact, static representation for this (base) graph, having distances and roadtypes; for Highway Hierarchies, this is a dynamic datastructure where only costs are stored with the edges: it is not needed to have distances and times there.

4.3 NodeMapper

In the ORTEC logistics suite (OLS), there are two labellings of nodes. One is the record number (RecNo) $0 \leq r < N$, the other is the NodeID, some arbitrary positive integer supplied by the map supplier. With HNR, a third labeling is introduced: The HNR NodeID. This NodeID is constructed such that the nodes are sorted by level, in order to allow for an efficient data structure storing the overlay graphs.

Initially, in the code from Karlsruhe a mapping existed in the UpdateableGraph class. This mapping uses a MSVC++[4] hashtable (from Microsoft's C++ template library). Unfortunately, such implementation of a hashtable is very inefficient: it's slow and takes a lot of memory due to the buckets being implemented as a linked list – lots of fragmentation. Also, there was only one mapping between the NodeID and the HnrNr, the RecNr was not part of this.

Queries via SOAP and the command handler interface are always specified in NodeID. When only time and distance is required, the result has no node references at all. Only if a path is requested, a list of nodes is returned. The specification in this case mandates that the nodes are returned as record numbers.

[4] Microsoft Visual Studio C++

Node label type	Source	Domain	Comments
NodeID	Map Supplier	positive integers	Global reference
RecNr	Ortec	$[0, N\rangle$	On-disk order
HnrNr	HH step	$[0, N\rangle$	Ordered by level

Given these requirements, and given that the translation calls would need to be done in the HNR code at all the entry points, we decided to move the mapping out of HNR, and make it the responsibility of the caller of the HNR library to simply work with HNR nodeIDs.

Therefore, a new library was created, NodeMapper, which supports mapping a NodeID to a HnrNr, and a HnrNr to a RecNr. Without assuming anything about the NodeID, we created a static hashmap which performs well, and has minimal overhead. Thus, a hash for nodeid→HnrNr, and an array for HnrNr→RecNr.

However, the NodeID's turn out to be sorted by RecNr. Given this fact, it is possible to have a nodemapping which is even smaller. An array HnrNr→RecNr and an array RecNr→nodeid is not enough, because with a nodeid, one can use a binary search on the second table to obtain a RecNr (this is what happens in Delphi-based ORTEC code). However, in this case we need to store a RecNr→HnrNr mapping too, in the end, still ending up with roughly 12 bytes per node for the mappings.

5 Multi-value path dimensions

The shortest path problem assumes some definition of *edge length*, sometimes also called edge weight or edge cost. In the context of road networks, two metrics for "edge length" are commonly used: the actual distance from source to target (for example, in meters), and the expected traversal time from source to target (for example, in seconds).

The original HNR implementation can work with any integer cost. By chosing either of the two metrics "distance" and "traversal time", one can perform queries for the respectively shortest and fasted routes. Since the chosen cost is the only metric stored in the map data structure, it is impossible to recover the other metric afterwards – the only metric returned is the one optimised for.

In a typical scenario, optimisation is done on a combination of time and distance: generally, the fastest oute is preferred, but for example a 50km detour for 1 minute of time savings, is not considered appropriate. The value that's used to find the optimal path is not enough to compose a schedule: for that, the actual travel time is required to find the optimal composition of travels that serve all customers.

For point to point queries, one solution would be to calculate the shortest path normally, retrieve the actual path, and calculate the time and distance by fetching those properties from all the involved edges. However, this could easily triple the calculation time. Even so, when performing a matrix query, getting simultaneously the best routes from a number of source nodes to a number of target nodes, that workaround is not feasible at all.

In this chapter, we explore how to simultaneously optimise on one value, and return two actual metrics (time and distance) at the same time, without incurring any significant performance penalty.

5.1 Discussion

In ORTEC's logistics products, a combination of these two real world metrics is used. An integer $0 \leq f \leq 100$ defines on a scale the influence of distance versus time in the cost function. $f = 0$ indicates that the cost function is completely determined by the travel time, and $f = 100$ means exclusive influence by distance, so a pure shortest-route cost function. Every other f indicates a weighted cost function. This is implemented by a linear weight function. The cost function at ORTEC is of the form $d_{\text{cost}}(A, B) = \alpha \cdot d_{\text{dist}}(A, B) + \beta \cdot d_{\text{time}}(A, B)$. In effect, this allows to express the optimal shortest path for example as "choose the fastest route, but do not make a 5 km detour for just 1 minute of time saving." In order to make the two distinct metrics comparable when $0 < f < 100$, normalisation of units needs to happen. At ORTEC, the

somewhat arbitrary choice of "one minute of travelling is equivalent to one kilometer travelled" was made. For example, with $f = 10$, that means that every minute costs 90 units, and every kilometer costs 10 units.

Summarising:

$$
\begin{aligned}
d_{\text{cost}} &= \alpha \cdot d_{\text{dist}} + \beta \cdot d_{\text{time}} \\
\alpha &= \frac{f}{100} \cdot \frac{1}{1000} \\
\beta &= \left(\frac{100 - f}{100}\right) \cdot \frac{1}{60}
\end{aligned}
$$

(d_{dist} is in meters, and d_{time} is in seconds)

The solution chosen for this problem is to replace the cost data type in all of the HNR code, previously a single integer, by a triple of numbers: distance, time, and cost. Thanks to C++'s operator overloading, very few changes needed to be made to the code once the basic operators like + and < were implemented on this new data type. The code was immediately functional, but at the cost of a significant raise in memory usage.

In the *UpdateableGraph* data structure, the vast majority of memory is taken by the edges vector. Every edge entry there used to have 4 bytes for the target, creation level, and some flags, and 4 bytes for the edge weight, totalling to 8 bytes. With the new edge weight being a triple of integers, this weight component is now 12 bytes, making the total amount of memory per edge 16 bytes: memory usage just doubled.

5.2 Biassed speeds

As long as cost function can be written as $c(d, t)$, where d is the distance, and t is the time, it is possible to always reconstruct the cost if you keep track of the distance and time in the overlay graph.

The concept of 'biased speeds' is a change in the cost function designed to give bias to some road types. For example, for the purpose of the cost function, the travel time component on highways is reduced by some percentage. Effectively this gives a significant preference (bias) to highways. In general, every road type could have a different factor in the cost function, as a way to prefer or dislike some road types.

Important to note is that the actual travel time that is to be returned in the end, is *not* changed, it is still the original time based on that road type. Therefore, the cost function cannot be expressed as $c(d, t)$.

In general, either the cost value itself should be maintained, or all input parameters, whichever is best for the situation. For example, one could maintain the distances for each separate road type. In that case, it is also not required to maintain the travel time, as it can be reconstructed. There are two stages in transforming the source-quantity "distance" into something different: using the average speed to get travel time, and using the cost function to get the cost.

6 Sharing ground network

6.1 Introduction and motivation

Highway Node Routing's preprocessed data structure consists of a number of graphs: the original graph, and a handful of overlay graphs.

Regardless of the vehicle profile, the original graph has the same structure (nodes, and what nodes are connected to what), and equal distances. In the economic variant of HNR, the total size of the combined overlay graphs is much smaller than the size of the original graph. Thus, if we want to support multiple vehicle profiles, we can have one instance of an original graph data structure shared among all the profiles, and for each profile a set of overlay graphs. This setup would enable multiple instances of HNR without using prohibitively much memory.

Let us consider as big map instance the map of Western Europe (over 12M undirected edges, 23M directed edges). After performing Highway Hierarchies with the merge level 1 setting turned on, it turns out that only 2.1% (206k) of the 10 million nodes belong to level 1 or higher, and only 4.7% (482,000) of the 11 million levelnodes[5] are for level 1 and higher. Of the 24.8 million undirected halfedges, only 2.8 million (11.1%) are in the overlay graph.

Given that the vast majority of the graph data structure is in level 0, we can allow ourselves to use a bit more memory for the overlay edges, as long as we ensure that the ground network (level 0) is lean and mean.

Splitting the base graph off and allowing independent overlay graph creation, loading and unloading, also provides some other practical benefits: creation of an overlay graph is much faster, there is no need to load and save the entire graph every time. The cost of loading the base graph is incurred only once, and also when storing the result, just the much smaller overlay graph needs to be written out to disk.

This splitting is done, and in the next subsections, we explain how. Section 6.2 details how the base network is represented, Section 6.3 explains the overlay graph, Section 6.4 how these are brought together.

The main problem addressed in this chapter is how to take advantage of the immutability of the largest amount of data while keeping the current performance characteristics and increasing memory efficiency.

[5] In the graph datastructure, there is a record for each node in each level, the "levelnode". A node that is present in 3 overlays would have 4 levelnodes: one in the basegraph, and one for each overlay graph it is present in

6.2 Representing the ground network

Our goal is to have one in-memory representation of the ground network, which can be shared among different instances of vehicle profiles and calculation profiles. Therefore, we can only store information that is equal among those instances, while it is still sufficient to generate all the information needed. See section 4.1 for a description of the items available in the ORTEC map format, especially table 1. This information is indeed not specific to any overlay-graph: travel times are not present in these files, only distances and road types (so that travel times can be computed with the help of a vehicle profile.Note that we do not store any node id mapping, so the information of `org_nr.net` does not need to be taken into account.

As can be seen in that table, copying this directly into memory would require 13 bytes per edge, or actually 16 bytes due to alignment constraints (if edge information is bundled into an object and stored in one big array). We aim to reduce the memory usage, and will show it is possible to reduce the usage to 8 bytes per edge without losing any information.

6.2.1 Implementation

The ground network is not hierarchical, so a simple adjacency edgelist is sufficient. This is implemented in the SimpleStaticGraph class.

```
class StaticSimpleGraph
{
  (...)
private:
  std::vector<EdgeID>   m_firstEdgeID;
  std::vector<SSGEdge>  m_edges;
  std::vector<uint32>   m_overrideLengths;
}
```

Listing 1: StaticSimpleGraph

This data structure takes 4 bytes per node plus 8 bytes per halfedge (a half edge is half of an (undirected) edge). The per-node overhead consists simply of an index in the halfedge vector where the outgoing halfedges for that node start.

Every halfedge consists of:

→ **27 bits** for the target (up to 134 million nodes)

→ **5 bits** for the road type (up to 32 road types, 18 currently used)

→ **9 bits** for the access bitmask (up to 9 vehicle types, all used)

→ **2 bits** for the direction flags

→ **16 bits** for the distance in meters (up to 65km)

→ **1 bit** as an overflow flag for the distance

For a total of 60 bits (8 bytes including 4 wasted bits). This compact representation is achieved using the `CompactStruct` class, see section 7.3.

In code:

```
1  class SSGEdge
2  {
3    (...)
4  private:
5    enum {flag_forward, flag_backward, flag_lengthoverride};
6
7    CompactStruct<32, 27, 5, 0> m_targetAndType;
8    // the bitwise negation of access is stored because the access
         flags default
9    // to be set, and we do not want to overflow
10   CompactStruct<16, 9, 0, 3> m_negAccessAndFlags;
11   uint16 m_dist;
12 }
```

Listing 2: SSGEdge

Although 65km seems sufficient for any single edge, this is not always true. Edges representing ferry connections can be much longer. These are luckily exceptions, and are therefore treated as such. If the distance override flag is set, `m_dist` does not represent a distance, but an index in an exception vector, where 32-bit integers are available for those few edges that are more than 65km long. Because of this, there is representation limit: no more than 65,000 distinct lengths longer than 65km can be represented in a single graph. This limitation does not pose problems on the actual maps in use.

6.2.2 Getting traveltime

Calculating The time needed to cross an edge is not explicitly given in the base graph. Given the distance in meters, and the speed as an integer in km/h, we can derive the time needed easily. Every road type has a speed associated with it in the Vehicle Profile, and every edge has a road type. Hence, given different Vehicle profiles, we get different times, therefore storing the time in the base graph while sharing it among multiple vehicle profiles is impossible.

Unit We have chosen to maintain the time in deciseconds: the error per edge is then at most 50ms (half of a decisecond), and for a path of 1000 edges would

accumulate to at most 50 seconds (probably less). It is possible to travel around the world nearly 3 times in $2^{32}-1$ deciseconds at a speed of 1 km/h, so we assume that we can store the quantity of time safely in an 32-bit integer at all times.

Rounding of traveltime Storing and using the time as an arbitrary precision rational number is not feasible, and because speeds might be such that no single unit of time allows a fixed-point unrounded representation, some rounding will be involved. We want to use the time per edge as a fixed per-edge integer value of some unit, it is intentionally not possible to retrieve the unrounded time. Given a vehicle profile, the (integer) time in deciseconds for a StaticSimpleGraph edge is static and deterministic, as if it were stored in a field. The reason behind this is that otherwise rounding errors can propagate inconsistently – and the repercussions on the correctness of HNR are not yet known. For example, we really want to have $(d_0 + d_1) + d_2 = d_0 + (d_1 + d_2)$.

Calculating time from distance is only done per edge, never combined. In order to make this frequent operation as fast as possible, without compromising accuracy, just one integer multiplication and division is done, followed by a rounded division by two which we will assume the compiler renders into two simple bit operations[6].

```
1  static uint32 decisecondsFromSpeed(uint32 meter, uint32_t speed)
2  {
3    /*
4    t        = d / v
5    t[ds]    = d[m] / v[m/ds]
6    v[m/ds]  = v[km/h] * 1000 / (3600*10)
7    =>
8    t[ds]    =     d[m] * (3600*10/1000) / v[km/h]
9    t[ds]    = 2 * d[m] * (3600*10/1000) / v[km/h] / 2
10   */
11   return roundintdiv<2>( 2 * meter * (3600*10/1000) / speed );
12 }
13 // do a perfect rounded integer division, x.5 and higher will be
       rounded up
14 template<uint32 divider>
15 static inline uint32 roundintdiv(uint32 num)
16 {
17   STATIC_ASSERT(DividerMustBeEven, divider % 2 == 0);
18   return (num + divider/2) / divider;
19 }
```

Listing 3: Calculating traveltime from distance and speed

With these building blocks, we can store the base graph economically.

[6]this is true for the compiler used, Microsoft's Visual C++

6.3 Overlay graphs

The overlay graphs also need to be represented in memory. In the original code, the constructing and querying interface assumes that all outgoing edges are grouped by source node, it uses ids to iterate from the first to the last of such edge. In addition, for HNR, it is assumed that the edges are also grouped by level, by providing an even more fine-grained interface to a specific subset of edges.

These properties are exported through the data structure interface: when using the `UpdateableGraph`, you can iterate efficiently over all outgoing edges.

The data structure which implements this (UpdateableGraph) has these properties, but at a significant penalty. In order to maintain grouping, space is reserved in advance, and when space runs out, the whole group is moved to a newly allocated bit of the edge array. This, unfortunately, does not fit well with the desire to have compact overlay graphs. More importantly it is complicated to maintain the data structure contracts, this is not at all enforced by the interface. This provides very little protection against programming errors, it takes a very deep understanding of all these assumptions to not introduce code that breaks the data structure contracts that are either implicit or explicit.

As an example, during construction and deserialisation, no gaps are maintained between edges of level zero nodes, but when manipulating the data structure, the user needs to ensure himself that it doesn't assume these gaps to be present, and also not to introduce them with the wrong nodes.

But the main reason that the UpdateableGraph data structure is not directly suitable for a split overlay graph, is that edges are grouped by node, and only then by level. Splitting out level 0 requires to leave this paradigm.

Because of these drawbacks we chose a different approach: A linked list implementation, with its fixed overhead of 4 bytes per edge, has a more predictable, and as experiments will show, more compact representation. We implement the graph with an adjacency linked list. Every node has a pointer to the first edge leaving there, and every edge has a pointer to the next edge leaving from that node.

The possible penalty here is that due to loss of locality, this code will perform worse due to higher number of processor cache misses. On the other hand, modifying operations are much cheaper now. Edges never need to be relocated, and adding and removing an edge involves a constant number of link operations. It is also a possibility to restore locality in linear time, by reordering the edges. If no more modifications are required, the next pointer can even be reduced to one bit (to still distinguish between "there is more" and "end of list"). In the event of memory pressure, the linked list representation has the benefit of being

more compact, and hence having a greater chance to fit in the available RAM.

In ComTec, the representation is always a linked list, so including this 4 byte overhead. But locality is restored on operational graphs, because there is always a serialisation and deserialisation step between construction of the overlay graph, and the actual use. The serialisation routine writes the edges by following the links in the linked list, and hence at deserialisation time, edges belonging to the same source node are next to each other. No code was written that explicitly uses this knowledge, as this is very unlikely to make any difference.

In the LeveledGraph, time is stored explicitly, because there is no homogeneous road type anymore on the combined shortcut edges, hence no speed with which to derive the time from the distance.

6.4 Putting the bits together

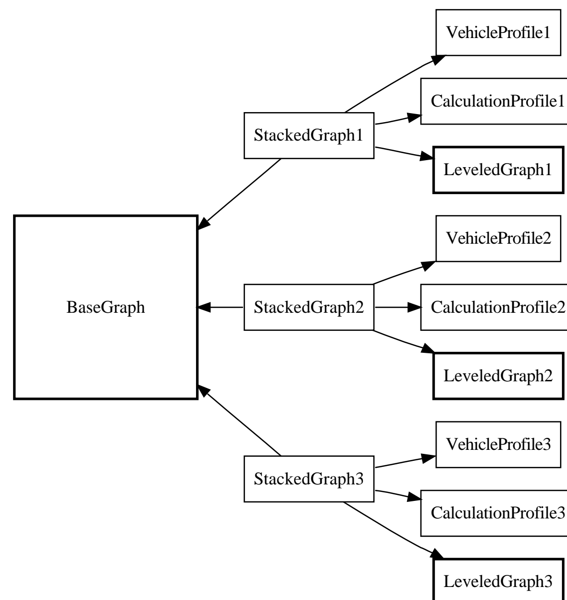


Figure 5: Instance example with three overlay graphs sharing one base graph. Arrows represent pointers to other object instances. Only StackedGraph objects are directly used by algorithms.

HNR algorithms, like the construction algorithm and its subalgorithms like stalling and graph reduction, and point to point/many to many queries, require a graph to operate on. With two distinct data structures, the StaticSimpleGraph (section 6.2) and the LeveledGraph (section 6.3) implementing respectively the basegraph and the overlay graph, these two need to be brought together and presented as one.

We introduce a new class, a StackedGraph, which will serve as a mediator, and provide an interface exposing a complete leveled graph (including base network). The StackedGraph has a pointer to both a StaticSimpleGraph and a LeveledGraph, but also to a VehicleProfile and a CalculationProfile. Without these two profiles, the base graph has only incomplete information on the graph: the layout and distances are known, but the time isn't (requires VehicleProfile), nor accessibility (requires vehicle profile), nor the cost (requires CalculationProfile).

6.4.1 Getting cost

Cost is not stored explicitly in either the base graph or the LeveledGraph. For any (sub)path, we would like to have the same cost whether we add up cost from two different edges, or retrieve the cost from a shortcut edge in an overlay graph.

More formally: given a path $A \succ B \succ C$, $d_{\text{cost}}(A,C) = d_{\text{cost}}(A,B) + d_{\text{cost}}(B,C)$, for all $A,B,C \in V$.

In order for this property to hold, no further rounding can be done. A cost function of the form $d_{\text{cost}}(A,B) = \alpha \cdot d_{\text{dist}}(A,B) + \beta \cdot d_{\text{time}}(A,B)$ where α, β, e_d and e_t are all integer, would satisfy this.

For ORTEC, the cost function is indeed defined as such. An integer f between 0 and 100 inclusive defines on a scale the influence of distance versus time in the cost function. $f = 0$ means zero influence by traveltime, so a pure shortest path cost function, and $f = 100$ means exclusive influence by traveltime, so a pure fastest path cost function. In order to make the two distinct units comparable when $0 < f < 100$, one unit of distance is normalised to be the distance travelled during one second at 60km/h, i.e., 60/3600th km. In effect, this means that the distance in meters is multiplied $\frac{3600}{60} \cdot \frac{1}{1000} = \frac{60}{1000}$, and the time is taken in seconds.

But recall that α and β must be integer to maintain the property above. In a first attempt, let's try the following, taking milliseconds as unit (recall that t is in deciseconds, hence the multiplication by 100):

$$\alpha = \left(\frac{100 - f}{100}\right) \cdot 60$$
$$\beta = \left(\frac{f}{100}\right) \cdot 100$$

When f is not divisible by 5, we get a fractional α, so we need to multiply both α and β with 5:

$$\begin{aligned}\alpha &= \left(\frac{100-f}{100}\right)\cdot 60\cdot 5 &&= 300-3f\\ \beta &= \left(\frac{f}{100}\right)\cdot 100\cdot 5 &&= 5f\end{aligned}$$

Now we have $0 \leq \alpha \leq 300$ and $0 \leq \beta \leq 500$. Suppose that $f = 100$, then time can be at most $2^{32}/500 \approx 8,600,000$ ds ≈ 10 days when the cost is represented as a 32-bit integer. This seems a lot, but long haul ferries can easily be 1000 km, and when a user sets the average speed of ferries to 4 km/h, we will overflow.

Solution Since the absolute value of the cost function is not important, but only the relative ordering matters, we can brute-force search all pairs of $(\alpha, \beta) \in [0, 25] \times [0, 15]$, and pick the pair that most closely represents the desired time factor f. This is done once when initialising the CalculationProfile, and takes a negligible $25 \cdot 15 = 375$ iterations. The time multiplication factor β is maximised to 15: travel time can go up to 331 days without overflowing an unsigned 32-bit integer. At 1km/h, one can travel 7950km in that time, or 15900km at 2km/h, the default ferry speed.

The maximum error is 5.

```
1  class CalculationProfile
2  {
3    uint32 m_distmultiplier;
4    uint32 m_timemultiplier;
5
6    void setDistTimeMultipliers(uint32 i_timeFactor)
7    {
8      double bestDiff;
9      // Border cases omitted
10     for (uint32 distM=1; distM<=25; distM++) {
11       for (uint32 timeM=1; timeM<=15; timeM++) {
12         // 1 meter equals a cost of 60
13         double distCost = 1.0/60.0;
14         // 1 decisecond equals a cost of 100
15         double timeCost = 1.0/100.0;
16         // fraction of time taken into account
17         double timeFrac = timeCost*timeM / (timeCost*timeM +
               distCost * distM);
18         double timeFactor = timeFrac * 100;
19         double diff = abs(timeFactor - i_timeFactor);
20         if (diff < bestDiff) {
21           bestDiff = diff;
22           m_distmultiplier = distM;
23           m_timemultiplier = timeM;
24         }
25       }
26     }
```

```
27      assert(bestDiff <= 5);
28    }
29  };
```

Listing 4: Calculating distance and time multiplication factors

6.4.2 Code listing

```
1  class StackedGraph
2  {
3    // (...)
4  private:
5    const StaticSimpleGraph* m_ssg;
6    LeveledGraph* m_og;
7    const VehicleProfile* m_vp;
8    const CalculationProfile* m_cp;
9    NodeID m_firstLevel1Node;
10 };
```

Listing 5: StackedGraph

7 Lower memory usage (peaks)

One of the leading goals of HNR and its associated algorithms is requiring an acceptable amount of memory. The research was motivated by practical considerations, and as such, memory usage was typically well within the limits of the used test system, 8GB.

However, for several reasons the memory available to HNR in the context of ORTEC's products is less. The most important of those is that the dominant versions of the used operating system, is still 32-bit based. Also, Microsoft Windows 32-bit has a per-process limit of 2GB, compared to 3GB with Linux[7]. Also, the ORTEC logistics suite consists of many components, each requiring memory, and memory not used directly by applications can be used by the operating system for caching. Either way, in practice there is much less memory available than than the 8GB that was available to the research group in Karlsruhe.

A complicating factor is that for ORTEC, we would like to work with both distance, time, and cost, not just cost as in the scenario used in Karlsruhe.

The main problem addressed by this chapter is memory efficiency, especially during pre-processing.

7.1 Data structures

All data in main memory is organised in so-called data structures. The key to reduce memory usage is consider the combined memory usage, and of those data structures that take up large chunks of memory, consider whether they can be improved.

In some cases, not all data present in the data structure must be stored there, and a reduction can be achieved by dropping certain parts. One example is storing data that can be easily reconstructed based on other data. In other cases, the data structure might benefit from a more compact representation.

In either case, a trade-off might be involved between processing time and data structure size. But that's not necessarily the case, less space does not necessarily worsen performance. For example, it may increase the chance of the whole data structure to fit in CPU buffers, the fastest of all memory types, and this can sometimes be more important than requiring fewer CPU instructions.

Analysis of original memory usage At first, we determined the contribution of the various data structures to the memory usage. This was done by querying the current process memory usage at specific points in time during the construction of all data structures, and sometimes, when that does not provide

[7] there is a way to raise this per-process limit on Windows to 3GB, but it's somewhat invasive and does not result in one continuous block of virtual memory space

an adequate view of memory usage, by looking at sizes of specific (parts of) data structures.

We ran into huge non-persistent memory usage peaks: at some point during construction, memory usage would spike, but immediately after drop to a more moderate level again. These spikes can be attributed to the usage of the C++ STL `std::vector` data structure for the list of edges. The *vector* data structure is essentially a resizeable array, with considerable memory usage spikes when the capacity is exceeded. Replacing the use of the STL `vector` with an own implementation of a block-wise `BigVector` data structure resolved these spikes. See section 7.4 on page 45 for more details on this new data structure, including why other data structures are not suitable for such applications.

Concrete improvements One pattern used in the HNR code is the pattern of storing integers and boolean values in precisely as many bytes as needed, using bitwise operations, instead of alignment to byte and CPU-word boundaries. This typically results in slower performance, as the CPU has to do more work to access and manipulate the associated data. But when millions of such items are involved, the memory saving may outweigh this reduction of performance.

Using this pattern more is codified with a particular class, to prevent problems with forgetting to care for it. In section 7.3 we elaborate on this.

More efficient edges vector In the original implementation of HNR, the edges in the UpdateableGraph are stored in one big vector. Every group of edges originating from a given source node, uses space equal to the next power of two. So, from 5 to 8 edges inclusive, 8 spots are reserved. From 9 to 16 edges, 16 spots are reserved. This implies that in any case, at most twice as much of memory is used as would otherwise be strictly needed.

7.2 Misc

In order to make parts of the code more readily reusable, and enable easier development of enhancements, several data structure idioms were isolated from the higher level data structures. One example is a compact representation of one or two integers along with some number of boolean flags: now dubbed CompactStruct, robustness of the code is improved: it is no longer possible to forget or mix up usage of this idiom, all methods of the class ensure that even when faced with out of range input data, other members of the compact struct are not affected. The resulting behaviour is much like those of native types regarding overflows.

7.3 Utility: CompactStruct

While dealing with huge maps, we're dealing with 10s of millions of edges. In order to store those edges efficiently in memory, it's sometimes wothwile to store integers more compact than the default integer types support. For example, store a 27-bit target node id (enough for 134 million nodes) together with a 5-bit road type (enough for 32 types) in one i386 word (32 bits).

In general, this class can be used to have a memory-efficient way to store one or two unsigned integers along with a number of boolean flags in a safe manner. Using template arguments you can decide on the total number of bytes allocated, the number of bits for each of the two (unsigned) integers, and the number of flags.

C (and hence C++) already provides a facility to allocate sub-byte members in a struct/class. This technique is called bitfields. However, there are quite some drawbacks on using those bitfields. One is that the exact memory layout is then undefined, and can change between compilers and architectures. Even when the compiler is fixed, like at ORTEC, this imposes a problem due to 32-bit and 64-bit differences: a serialised graph would differ based on whether it is created in the 32-bit compiled version or the 64-bit compiled version of the code. A more serious limitation is that all bitfields combined are padded to the nearest multiple of the architecture's natural word size: 32-bit or 64-bit. In the context of HNR, it is sometimes desireable to be able to store an integer with some flags in a 16-bit field. Therefore, this more portable solution was created.

During all write operations, including the constructor, all input is masked, such that if an overflow happens, the other members are not affected. An assertion error might be raised unless assertions have been switched off.

All template arguments are statically verified, so you cannot make a mistake like wanting to put too many bits in a byte, or use flags that you did not allocate.

Template arguments:

- `t_bits: the number of bits to allocate (must be a supported native integer type, typically 8, 16, 32 or 64)`
- `t_int1size: The number of bits (>= 1) for the first unsigned integer`
- `t_int2size: The number of bits for the second unsigned integer, 0 to disable`
- `t_nflags: The number of boolean flags to support`

```
1 CompactStruct<32, 26, 4, 2> targetAndType(1234567, 3);
2 printf("target is %d, type is %d, in %d bytes\n", targetAndType.
      getInt1(), targetAndType.getInt2(), sizeof targetAndType);
```

```
3 // target is 1234567, type is 3, in 4 bytes
4 targetAndType.setFlag<1>(true); // set the second flag
5 // targetAndType.setFlag<2>(false); // compile error
```

Listing 6: Example

```
1 template <size_t t_bits, size_t t_int1size, size_t t_int2size,
      size_t t_nflags>
2 class CompactStruct
3 {
4 private:
5   typedef typename IntegerTypes<t_bits>::unsigned_type fieldtype_t;
6
7   static const size_t s_int2shift = t_int1size + t_nflags;
8   static const size_t s_flagshift = t_int1size;
9
10  static const fieldtype_t s_int1mask = ((fieldtype_t)1<<t_int1size
        ) - 1;
11  static const fieldtype_t s_int2mask = ((fieldtype_t)1<<t_int2size
        ) - 1;
12
13  // layout: <empty><int2><flags><int1> (left to right, or msb to
        lsb)
14  fieldtype_t m_field;
15
16 public:
17  CompactStruct() : m_field(0) {}
18
19  CompactStruct(fieldtype_t i_int1) : m_field(i_int1 & s_int1mask)
        {}
20
21  CompactStruct(fieldtype_t i_int1, fieldtype_t i_int2)
22    : m_field(((i_int2 & s_int2mask) << s_int2shift) | (i_int1 &
          s_int1mask))
23  {}
24
25  // A destructor is always generated, so this is a good place to
        put static asserts
26  ~CompactStruct()
27  {
28    STATIC_ASSERT(Int1MustBeUsed, t_int1size > 0);
29    STATIC_ASSERT(DatatypeMustHaveEnoughBits,
30      t_int1size + t_int2size + t_nflags <= t_bits);
31  }
32
33  fieldtype_t getInt1() const
34  {
35    return m_field & s_int1mask;
36  }
37
```

```
38    void setInt1(fieldtype_t i_newvalue)
39    {
40      m_field &= ~s_int1mask;
41      m_field |= i_newvalue & s_int1mask;
42    }
43
44    fieldtype_t getInt2() const
45    {
46      STATIC_ASSERT(Int2MustBeAllocated, t_int2size > 0);
47      return (m_field >> s_int2shift) & s_int2mask;
48    }
49
50    void setInt2(fieldtype_t i_newvalue)
51    {
52      m_field &= (~s_int2mask) << s_int2shift;
53      m_field |= (i_newvalue & s_int2mask) << s_int2shift;
54    }
55
56    template<size_t flagnr>
57    bool getFlag() const
58    {
59      STATIC_ASSERT(FlagNrOutOfRange, flagnr < t_nflags);
60      return 0 != (m_field & ((fieldtype_t)1<<(s_flagshift + flagnr))
          );
61    }
62
63    template<size_t flagnr>
64    void setFlag(bool i_value)
65    {
66      STATIC_ASSERT(FlagNrOutOfRange, flagnr < t_nflags);
67      if (i_value)
68        m_field |=  (fieldtype_t)1<<(s_flagshift + flagnr);
69      else
70        m_field &= ~((fieldtype_t)1<<(s_flagshift + flagnr));
71    }
72  };
```

Listing 7: CompactStruct

7.4 Utility: BigVector

Due to the large amount of small objects of certain types, like edges and nodes, those objects are not allocated individually. This would cause a lot of allocater overhead, just as importantly, would incur at least 4 bytes (or 8, if compiled for 64-bit) of overhead per instance for the pointer referring to it. If the actual size of the data object is small (8 or 12 bytes are not unusual), memory demand would even double, and the same problem now needs to be solved for the involved pointers.

Fortunately, C++ allows the programmer to store objects in consecutive blocks of memory, and allow referencing the objects by index. Two standard forms of these are the C array, and the STL vector class.

This works well, as long as you know beforehand how many of those objects you want to instantiate: a continuous block must be allocated, and cannot generally be enlarged later on (there might already be other data just beyond the old end). With an array you're out of luck if you want to store more objects than originally planned, but the STL vector class has functionality to deal with this case. When the capacity is exceeded, a new, larger continuous block of memory is allocated, all the old data is copied to the new block, and the old block is deallocated. This causes a big memory usage spike and unnecessary moving around of data.

Originally, the list of edges is stored in a STL vector. During construction, it is not known yet how big the overlay graph will be. Due to the amount of edges, the spikes are significant.

As an alternative, we have created a 'BigVector' class that uses block-wise allocation of space, allocating new blocks as needed. The interface is the same as for the STL vector, with some methods left unimplemented because they are not needed. One example is insertion in other places than the end, it is of course possible to implement that, but would not be very efficient. There already is a data structure (rope) that can deal efficiently with half-way insertions and deletions.

Using BigVector in certain select places as a replacement for the STL vector, removess the need for copying data, and removes the memory usage spike. With this change, it was possible to preprocess substantially larger maps than before.

Some example methods from BigVector

```
1  template <typename t_element_t, size_t t_blockbytes =
       _BIGVECTOR_DEFAULT_BLOCKBYTES>
2  class BigVector
3  {
4  public:
5    typedef t_element_t value_type;
6
7    t_element_t& operator[](size_t i)
8    {
9      assert(i < m_size);
10     return (m_blocks[i/s_blocksize])[i % s_blocksize];
11   }
12
13   void resize(size_t i_newsize)
14   {
15     if (i_newsize > m_size)
16     {
```

```
17        const size_t newLastBlockNr = (i_newsize-1)/s_blocksize;
18        reserveBlocks(newLastBlockNr+1);
19        for (size_t i = m_size / s_blocksize; i < newLastBlockNr; ++i
              )
20        {
21          m_blocks[i].resize(s_blocksize);
22        }
23        m_blocks[newLastBlockNr].resize(i_newsize - newLastBlockNr*
              s_blocksize);
24        m_size = i_newsize;
25      }
26      else
27      {
28        throw "BigVector::resize: Decreasing size not supported";
29      }
30    }
31
32
33
34  private:
35    // blocksize is in number of elements, such that each block is
          t_blockbytes
36    static const size_t s_blocksize = t_blockbytes / sizeof(
          t_element_t);
37
38    typedef std::vector<t_element_t> block_t;
39
40    std::vector< block_t > m_blocks;
41    size_t m_size;
42
43    void reserveBlocks(size_t num)
44    {
45      for (size_t i = m_blocks.size(); i < num; ++i) {
46        m_blocks.push_back(block_t());
47        m_blocks[i].reserve(s_blocksize);
48      }
49      assert(m_blocks.size() >= num);
50    }
51  };
```

8 HNR and forbidden areas

8.1 The situation

Different vehicle profiles and calculation profiles have different road access characteristics. Trucks may not access certain shopping areas, because of environmental regulations, or the existence of areas reserved for pedestrians. With certain calculation profiles, ferries are disallowed, and thus islands become impossible to reach as a whole.

When access restrictions are truly restrictive, there is no problem. Possibly the overlay graphs are disconnected, and consist of multiple components. When routing within reachable areas, everything works fine, and when calculating a route between two mutually unreachable nodes, the correct result is returned: no connection possible.

However, implementing such access restriction very rigidly yields practical problems while scheduling. When a supermarket in the center of a city needs to be supplied, the transport company might enter the postal address of the supermarket location – but that's at a pedestrian zone, the loading bay at the back door is, however, reachable. It is undesirable to not have routing in this case. The assumption is that any address is ultimately reachable, but it might require location-specific knowledge of the truck driver for the "last mile". For composing a transport and distribution schedule, it is not very relevant how exactly the "last mile" is done: it will take just a few minutes, and the error margin on the whole travel is much higher anyway. What is important, is that the complete travel has some realistic time.

As a consequence, for the purpose of planning, it makes sense to implement an access restriction not as a hard constraint, but as a more significant component in the edge cost. The primary optimisation criterion would be "minimize the number of access violations", or "minimize the number of meters travelled over forbidden roads", and only the secondary criterion would be "minimize the travel cost". This would result in the algorithm avoiding forbidden roads, but taking them anyway if it is unavoidable.

8.2 Naive Implementation

The naive implementation is to extend the edge weight data structure with yet another field: the number of violations. This would work much like in chapter 5. There can be sizable areas of all inaccessible roads, therefore every edge in the overlay graph data structure would need to have some more bytes reserved, yielding a significant increase in memory usage (roughly 25%).

Worse, experiments reveal a dramatic drop in performance for both the pre-

processing and the actual querying. The stall-on-demand technique used in the HNR preprocessing phase and in the query phase now fail to work correctly, and in the neighbourhood of 'forbidden' edges, the search space explodes to include the complete graph. The introduction of the accessibility factor indeed introduces a severe violation of the otherwise roughly proportionate edge lengths. A search is only stopped once it is sure that no other connection would yield a shorter path, but if the only way to get to a neighbouring node is via a violation, the search algorithm has no choice but to explore the complete graph before really being able to conclude nothing better can be found: it is better to have a many thousands kilometer detour than to commit a violation. The levels of the nodes are not helping out: which nodes specifically are unreachable, is very dependent on the profiles chosen: in one profile, a road can be very important, in another it is a last resort if no other route exists.

We conclude that the naive implementation is not workable in practice.

8.3 A more workable strategy

When looking at access violations in practice, one notices that by far the most nodes are well reachable. Only islands, shopping centres, and many tiny dead ends are unreachable.

Let us calculate the strongly connected components of the road network graph. Tarjan's algorithm[Tar72] is well suited for this. We check whether the largest such component indeed covers the majority of the graph (as experiments show, typically more than 99.9% is in the largest strongly connected component). For all nodes that are not in this component, we force the level of that particular node to zero, even if during Highway Hierarchies preprocessing the level was determined otherwise. As a result, all nodes in all overlay graphs are now in the strongly connected component, and as a result, each overlay graph is fully connected without any access violations. Construction can happen without search space explosion, because we can now add a condition to the search: edges with a violation will not be relaxed at all: it would never yield a better result.

For the query phase, recall that any HNR query is always bidirectional. It starts out with the source and target node, and the search expands until the two search spaces meet. Only once the search space is outside any unreachable part of the graph, it will find nodes that are in the overlay graph. Compared to the old situation, the search space is maximised by the largest component that's not in the largest strongly connected component.

The decision which nodes are to be forced to level zero is a vehicle profile dependent decision, so the set of nodes that are forced to level zero is stored

alongside the overlay graph.

8.4 Multiple components

The query in nodes that are not in the reachable component is always done on level zero, without help from the overlay graphs. But consider the situation where ferries are restricted, and one has a map of Europe including the United Kingdom and Ireland. The island of Great Britain has millions of nodes. Routing through these nodes without overlay graph would severely degrade performance: we effectively degraded to Dijkstra-like performance. To prevent that, a threshold is required on the size of subgroups of unreachable nodes, that are themselves reachable (even though via violations). For an island the size of Texel, with a few thousand nodes, this is not a problem.

We can also allow more than one component to be considered reachable, and considered for the overlay graph. However, because the components are not mutually reachable, the overlay graph too would be disconnected. If a query would happen between nodes in the different components, and both searches have reached the overlay graph, no result would be found (recall that the search would never descend to a lower level, so level zero will not be used anymore – and it was the only hope of finding a connection). But searches within components would work well.

One idea is to use this as a feature: if a query yields "unreachable", no road connection is available, and alternative transport possibilities should be employed in the schedule. However, this has several problems:

In the first place, it is hard to tune the parameters to get the correct number of components. Should Ireland be a component of itself? If the only tuning parameter available is component size, this is hard to achieve.

Secondly, and more importantly, the cut between components is not well defined. Close to a disallowed ferry, a search can take the ferry before ascending to higher levels in the overlay graph. Only further ashore the search space would typically be dominated by the nodes in the overlay graph before reaching the coastline.

Thirdly, as a practical matter, keeping track of unreachable connections takes adaptations to data structures and protocols in ComTec. There is currently no way to signal reachability, and if a matrix is request containing addresses of two components, the whole matrix result would currently be rejected.

As a consequence, the default settings at ORTEC are such that only one component is allowed, by setting the minimum component size to 50% of the nodes or more. This setting is conservative, because the twin setting specifying the maximum size of an unreachable area is the one that's designed to guard

against performance issues.

8.5 Consequences for the implementation

In the vehicle profile specific preprocessing, that is, while generating the overlay graphs, three new steps have been introduced:

- Tarjan's algorithm is ran over the graph, while all edges which are inaccessible are considered non-existent. The component(s) of sufficient size according to some parameter are tagged.
- All other nodes are tagged "hard to reach", and the level of all those nodes is set to zero
- As a precaution, the largest tree of such "hard to reach" nodes is searched, and if the size exceeds some threshold, construction is aborted.

In the rest of the implementation, two modifications were required: the first being that during construction, edges with violations are not relaxed, to ensure that all shortcut edges in the overlay graph have zero violations.

The larger modification is changing the edge weight property in the query algorithm. In the priority queue, and the labeling of nodes, the tentative and permanent labels must include the number of violations. It is for the correctness of the algorithm important that edge weights obey normal numeric properties, such as $(A+B)+C = A+(B+C)$, so the number of violations is stored as an integer with large domain (32-bit). Because the number of labels and entries in the priority queue is limited, this is not a problem.

The overlay graph data structure didn't need to change, and for the base graph the existing access bit for each edge is adequate to encode 1 or 0 violations (see chapter 6 on storing this bit).

9 Experiments and results

In the previous chapters, many different ways to improve the flexibility of HNR have been attempted. However, just as HNR itself is not, from a theoretical computer science point of view, provable more efficient than plain old Dijkstra's algorithm, the changes discussed in this thesis do not necessarily improve matters.

In this chapter, we show the results of numerous experiments performed to determine the actual effectes of the adaptations proposed in this thesis.

9.1 Methodology

In general, all experiments are conducted with three different maps: a small map of the three BeNeLux countries, `NBeNeLux725`; the largest map currently in production at ORTEC customers covering the South-Eastern USA, `NUSAregSE750`, and a map covering most of Europe, currently too large to use in practice due to prohibitive amounts of CPU required for routing calculations. All three maps are supplied by NavTeq[8].

Map name	Nodes	Edges	Avg. Degree
NEuroTest2007	33 726 990	40 992 944	2.43
NUSAregSE750	6 692 241	8 647 314	2.58
NBeNeLux725	1 598 251	2 011 833	2.52

(the number of edges counts the number of (directed or not) node-node connections, so would be 4 for the graph in Figure 1. Degree is the number of edges connected to each node, so every edge is counted twice)

Figure 6: Main features of tested maps

All tests are run on a machine with a single Intel Core 2 Duo processor at 2.67 GHz, and 8GB of RAM. The operating system is Microsoft Windows Vista 64-bit, and the code is compiled using Microsoft Visual C++ 2008. In all experiments, the Microsoft-specific "Secure SCL" feature has been turned off, because it forced range checking for STL vectors, which is something we only want to do during debugging.

Unless otherwise noted, all experiments are performed 6 times, the first result is discarded (making sure the cache is full) and the other 5 experiments are averaged.

9.2 Time

Many to many queries, existing Dijkstra-based implementation versus HNR:

[8] `http://www.navteq.com/`

Dimension	From cache	Uncached	HNR
1×1	5ms	2s	1ms
100×100	30ms	430s	866ms
1000×1000	2.2s	1h23m	7.8s
5000×5000	177s	15h	46s

For smaller queries, nothing beats getting a response from a completely filled cache. However, if part of routes are not in the cache, HNR performs better.

For large results, with millions of source-target pairs to be returned, HNR is taking a lead over the cache. This can be explained by an improved implementation in the protocol layer, the actual retrievel time from the cache is very small. What we notice here is that even though HNR takes quite a while to calculate all 25 million results in the last testcase, this time is still an improvement over the old situation.

10 Conclusion

With the adaptations proposed and implemented in this thesis, Highway Node Routing has shown to be able to be used in production systems.. In terms of the 10-point list in the introduction, the code runs succesfully on Windows as a reliable service, and can be used via both a network service interface and via shared library calls. The ORTEC network format is supported and the memory usage is reduced to the point that bigger maps than what was previously possible at ORTEC can be used.

The flexibility of HNR is shown by the ability to have multiple overlay graphs in one process now. And finally, when there are a limited amount of blocked edges, we can now perform queries here in the dynamic fashion while simultaneously let the unmodified graph remain available for other queries in the same process.

10.1 Objective and central question

The goal of this thesis was to develop a service that can answer one-to-one and many-to-many queries across a map in a consistently fast manner, using Highway Node Routing.

This goal has been achieved. At the time of writing, the first release with Highway Node Routing as the default routing subsystem is in a pilot phase with a couple of ORTEC's customers.

With Highway Node Routing, a single-source, single-target query can consistently be answered within 10ms, including the remote procedure calling interface. This is an improvement of 99% over the case when one (or both) of the addresses were new in the old situation, and not a notable regression if both were already cached.

In the case of many-to-many requests, things are a bit more diffuse. For small requests, there is a no noticeable difference. For medium-sized requests, it is clear that calculating the routes takes a non-trivial amount of time, and a fully filled cache beats Highway Node Routing easily. At 11s for a 1000×1000 matrix, the performance is still acceptable. For really large requests, HNR beats the cached matrix implementation. This is not thanks to Highway Node Routing itself, but due to work on improving the transport mechanism.

Highway Node Routing as currently implemented at ORTEC is now a full replacement for the current routing subsystem, including most if not all existing functionality: ad-hoc queries, routing in the presence of forbidden edges, routing with a cost-function based on time and distance, and still returning both values.

The central question in this thesis will be: *How can Highway Node Routing be applied, adjusted and optimised for use by scheduling software?*

For this, several changes to the original implementation were required. Most notable are the changes required to get routing in the presence of forbidden edges to work like the older, Dijkstra-based implementation: those edges are avoided, but taken if required.

10.2 Subquestions and outline

To answer the central question the following subquestions are formulated:

Integration

- Adding interfaces required for ORTEC's ComTec framework, embedding as a service
- Reading the ORTEC map format

Implementation

- Multiple path ('length') dimensions during calculation
- Being more memory efficient, amongst other by sharing data among HNR overlay graph instances

Adaptations of the HNR algorithm

- Make HNR work sensibly with forbidden areas
- Prepare HNR for dynamic queries w.r.t. extra restricions

A Bibliography

[Dij59] E.W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1(269-270):6, 1959.

[dK] A. de Koning. Shortest path algorithms based on Component Hierarchies.

[DSSW06] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Highway hierarchies star. *9th DIMACS Implementation Challenge [1]*, 2006.

[Flo62] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, 1962.

[FT87] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[GKW06] A.V. Goldberg, H. Kaplan, and R.F. Werneck. Reach for A*: Efficient Point-to-Point Shortest Path Algorithms. In *Proceedings of the eighth Workshop on Algorithm Engineering and Experiments and the third Workshop on Analytic Algorithmics and Combinatorics*, page 129. Society for Industrial Mathematics, 2006.

[HNR68] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[Joh77] Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *J. ACM*, 24(1):1–13, 1977.

[Sch05] Dominik Schultes. Fast and exact shortest path queries using highway hierarchies. Master's thesis, Department of Computer Science, Universität des Saarlandes, 2005. `http://algo2.iti.uka.de/schultes/hwy/`.

[Sch08] Dominik Schultes. *Route Planning in Road Networks*. PhD thesis, Fakultät für Informatik, Universität Fridericiana zu Karlsruhe, 2008. `http://algo2.iti.uka.de/schultes/hwy/`.

[SS05] P. Sanders and D. Schultes. Highway Hierarchies Hasten Exact Shortest Path Queries. *LECTURE NOTES IN COMPUTER SCIENCE*, 3669:568, 2005.

[Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1:146–160, 1972.

[Tho99] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM (JACM)*, 46(3):394, 1999.