Many-to-many Customizable Route Planning with Time-dependent Driving Restrictions

Pim Agterberg (ICA-3470814)
Supervisors: Erik Jan van Leeuwen, Marjan van den Akker and Gernot Veit Batz

Department of Information and Computing Sciences, Utrecht University

December 18, 2017

Abstract

We present a variant of Customizable Route Planning (CRP) that computes all shortest paths between source nodes $s \in S$ and target nodes $t \in T$. Customizable Route Planning is a flexible routing algorithm that uses partitions to quickly calculate routes. It supports arbitrary metrics and can introduce a new metric fast. We also explored the feasibility of CRP in a time-dependent setting, where we take into account driving restrictions, i.e. roads are inaccessible during the night. By using a clever representation of time-dependent travel costs, we show that CRP can give exact results, taking into account driving restrictions, using little RAM and computing time.

1 Introduction

The *Vehicle Routing Problem* (VRP) asks "What is the optimal set of routes for a fleet of vehicles to deliver to a given set of customers?". This is a generalisation of the *Travelling Salesman Problem*. Often, the context is that of delivering goods located at a central depot to customers who have placed orders for such goods. Transportation is usually a significant cost, thus savings in this area are meaningful. To solve such a problem, a shortest route between each pair of customers is needed [1].

Finding the shortest route between a set of interesting locations is called the *many-to-many shortest route problem*. We formalize this problem as follows: given a set S of sources and a set T of targets, find the distances $dist(s, t)$ for all $s \in S, t \in T$. Here $dist(s, t)$ is the length of a shortest path from s to t in a (directed) graph $G = (V, A)$ with a set V of vertices and a set A of arcs.

A trivial solution to the many-to-many shortest route problem is to do $|S| \cdot |T|$ one-to-one shortest path queries. However, this is too slow in practice, so a faster approach is needed. A prominent solution to this issue is storing a subset of the one-to-all results, for all nodes in S and T, called a search space. For each combination (s, t), the intersection of these search spaces is inspected. The shortest path within this intersection is the requested shortest path $dist(s, t)$. A running time of $O((|S| + |T|) \cdot QT)$, where QT is the time required for a single one-to-all query, can thus be achieved for certain algorithms, namely those based on bidirectional searches [2].

It remains to choose a fast algorithm for one-to-one queries. Road networks are huge graphs, with millions of nodes and arcs. This means that even near-linear algorithms, like Dijkstra's algorithm, are not fast enough as an underlying shortest path algorithm [3]. Speed-up techniques have been devised that divide the work into two phases: a slower preprocessing phase that produces auxiliary data, and a query phase that uses this data to find the solution in a matter of milliseconds. See the section on related work for a full overview.

Two of the developed techniques stand out as the most frequently used in practice: *Contraction Hierarchies* (CH) and *Customizable Route Planning* (CRP). *Contraction Hierarchies* exploit the inherent hierarchical structure of road networks by adding shortcut edges. A modified Dijkstra algorithm can then find a shortest path in a fraction of a millisecond, visiting only few nodes [4,5]. *Customizable Route Planning* is specifically designed to fit the needs of real world systems. To achieve this, the preprocessing step of CRP is split into two parts. In the first metric-independent step, a multi-level partition is created, as are the topology of an overlay graph and empty distance matrices. In the second phase the actual costs are entered into the distance matrices, by processing the cells in a bottom-up fashion [6]. A more detailed explanation of CRP will be given later (See section 3.1). Both of these algorithms use bidirectional searches and can thus be adapted for many-to-many shortest paths.

While many-to-many versions of some shortest path algorithms have been researched, the many-to-many version of CRP is a surprising omission [2,5,7]. The concepts used for the other algorithms might be applicable to CRP. Creating a many-to-many version of CRP is the first key goal of this research proposal.

Up until now, we assumed static cost arcs. In practice, costs are far from static. Some arcs might be inaccessible at times. Think of closed-off city centres, highways closed during the night, or ferries. Another cause for a non-static cost is congestion. Some research on time-dependent routing has been done [8,9]. We aim to extend this research by answering many-to-many queries in a time-dependent setting.

There are several problems that arise when trying to combine many-to-many queries with time-dependency. First of all, the added layer of complexity naturally leads to a longer running time. While a slightly longer running time is no problem, we need to stay within acceptable bounds. The second concern is memory usage. Time-dependent arcs and paths take a lot more space than a simple static number. When we try to store complete search spaces for all points in S and T, on a continental sized road network, we thus use vast amounts of memory. While this memory requirement might pose no problem for modern servers with hundreds of Gigabytes of RAM, this is impractical for every-day use by end users.

When handling time-dependent costs, *Customizable Route Planning* (CRP) is the better choice over *Contraction Hierarchies* (CH). While CHs have slightly faster query times in general, we believe CRP will scale better with time-dependency. The power of CHs comes from contracting nodes and adding a small number of shortcuts to maintain correctness. With time-dependency added, this number of shortcuts increases enormously, since routes no longer strictly dominate each other. CRP does not suffer from this increase in complexity, since the number of stored shortcuts remains the same when adding time-dependency.

We want to show how CRP can be used in a many-to-many set-up and test the feasibility of a time-dependent many-to-many implementation. This leads to the following research question:

How can Customizable Route Planning be used in a many-to-many set-up? How does it perform with respect to space requirements, set-up time, and look-up time?

Outline The rest of this thesis is organized as follows. Section 2 gives an overview of related work. Our approach is outlined in Section 3. We provide experimental evaluation in Section 4 and finish with concluding remarks in Section 5.

2 Related Work

The classical solution to routing problems is *Dijkstra's algorithm* [3], which can solve the one-to-one shortest path problem in practically linear time. Beginning with the starting node s the algorithm scans outgoing edges and updates tentative costs. In each iteration the node with next lowest costs is selected and scanned. When the target node t is finally chosen, the solution is found. To improve on this standard, research was conducted into several directions. Below is a brief overview. For a full summary please refer to Bast et al. [10].

2.1 Speed-up Techniques for one-to-one Shortest Paths

2.1.1 Goal-directed Techniques

Goal-directed techniques aim to guide the search towards the target t. The most well-known goal-directed technique is *A** [11]. *A** uses a heuristic, which is a lower bound on the distance to t. It uses a modified *Dijkstra's algorithm* where the priority of a node is the tentative cost + the heuristic distance. This causes the algorithm to converge to the solution faster.

Faster convergence can be achieved using the ALT (*A**, *landmarks*, and *triangle inequality*) algorithm [12]. In a preprocessing stage, a number of nodes are selected as *landmarks*. The algorithm stores the distance between them and all vertices in the graph. These distances together with the *triangle inequality* are used to create a heuristic to be used with *A**. The quality of this heuristic depend heavily on the selected *landmarks*.

Another goal-directed technique is the concept of *Arc Flags* [13]. The graph is partitioned into cells during a preprocessing stage. For each arc, information is stored that shows if the arc is part of a shortest route to each cell. During querying this information is used to skip arcs not leading into the cell of t. The recently developed PHAST algorithm [14] computes these *Arc Flags* significantly faster.

2.1.2 Separator Based Techniques

Separator-based techniques exploit the observation that road networks are close to planar, implying that small separators exist in the graph. Graph separators can be split into two groups: *vertex separators* and *arc separators*.

A *vertex separator* is a subset $S \subset V$ of vertices that decomposes the graph into several cells when removed. Using this separator, an overlay graph over S can be constructed. Shortcut arcs are added to the overlay such that distances between any pair of vertices in S are preserved. The overlay graph can then be used to accelerate the query algorithm. Careful selection of the vertices results in significant speed-ups [15]. This approach can be applied recursively, resulting in multiple levels of overlays [16].

On the other hand, *arc separators* split the graph into cells by cutting arcs. For each created cell, shortcuts are calculated from and to all boundary vertices. The first version of this approach is the *Hierarchical MulTi method* [17]. After the preprocessing stage, *Dijkstra's algorithm* is applied on the graph induced by the cut arcs and the shortcuts.

Customizable Route Planning is built on the same principle and was devised a decade later [6]. For a full explanation, see section 3.1. CRP has been successfully implemented for numerous scenarios. Fast one-to-one queries with turn costs and fast recalculation of the metric were achieved by Delling et al. [6]. Additionally, one-to-one queries in the time-dependent case were solved with small error ($< 3\%$) by Baum et al. [18]. Delling et al. showed that using GPUs during the preprocessing leads to faster running times [19]. Finally, a method allowing the user to use a linear combination of metrics was devised by Funke et al. [20].

2.1.3 Hierarchical Techniques

In reality, sufficiently long routes eventually converge to a small network of important roads, such as highways. Hierarchical techniques utilise this concept.

Without loss of optimality, a node can be *contracted* by removing it from the graph and adding shortcuts between adjacent nodes. Short cuts are only added if the contracted node is part of a unique shortest path. After ordering all the nodes in order of some notion of importance, these nodes are *contracted* in order, adding the shortcuts in the process. The resulting structure is called a *Contraction Hierarchy* (CH) [4,5]. To find a shortest path in a CH, a bi-directional search is performed. A bidirectional Dijkstra search is performed from both source s and target t, only visiting nodes of higher rank. Geisberger et al. proved that, by combining the results of these two searches, a shortest path is found [4].

Exact Time-dependent CHs have been researched by Geisberger et al. [7]. Recently Strasser et al. presented a simple, but powerful heuristic for time-dependent routing on CHs [21]. Querying for a linear combination of two metrics was shown with CHs by Geisberger et al. [22]. Lastly, *Contraction Hierarchies* with customizable costs have been researched [5,7].

Highway Hierarchies [23] and *Highway Node Routing* [24] are predecessors of CH, based on the same principles. Besides being easier to understand, CHs are also faster [10].

2.1.4 Reduced Hop Techniques

By precomputing distances between pairs of vertices, shortcuts can be added to the graph. Using these shortcuts, a search on the graph can find its destination in fewer hops, and thus in less time. Single hop paths can be achieved by precomputing the distances between all pairs of nodes. For bigger networks, this is not only costly time wise, but also storage wise.

Alternatively, a two-hop approach can be considered: *Hub Labeling* [25,26]. Each node stores a number of *labels*, which contain the shortest distance from itself to certain nodes. These labels should obey the cover property: for any pair of vertices (v_0, v_1), at least one node on a shortest $v_0 - v_1$ path must be in the labels of both v_0 and v_1. We can now find a shortest path between two nodes by checking the intersecting *labels* of s and t.

Similarly, *Transit Node Routing* (TNR) [27,28] uses a distance table on a subset of nodes: the transit nodes. In a preprocessing stage, all pairwise distances between transit nodes are calculated. For each node in the original graph, a set of *access nodes* is calculated. A transit node is an *access node* if there is a shortest path through this transit node. Besides the set of transit nodes which are access nodes, the distance to these nodes is also stored. The shortest path query now hops from source s to target t through the *access nodes* of s and t.

2.1.5 Combinations

Some of the techniques mentioned can be combined to achieve even better results.

SHARC [29] combines shortcuts with *Arc Flags*. During preprocessing a partition is made and then the shortcuts and *Arc Flags* are calculated in turn. Unimportant nodes are *contracted*, with the restriction that shortcuts never span multiple cells of the partition. Next, *Arc Flags* are computed such that, for each cell C, a shortcut is only used if the target node is not in C.

Another combination with *Arc Flags*, is called CHASE [30]. It combines CHs with *Arc Flags*. A regular CH is constructed and *Arc Flags* are computed for the top k nodes. The PHAST algorithm [14] is fast enough to enable k to be the whole graph.

CH can also be combined with ALT: *Core-ALT* [30]. First an overlay graph is computed for the *core graph*: the top k nodes. Then *Landmarks* are computed for the core nodes only.

Lastly, the combination of TNR and *Arc Flags* results in TNR+AF [30]. *Arc Flags* are added as usual, after which a shortest path query only considers *access nodes* with correctly set *Arc Flags*.

2.2 Speed-up Techniques for many-to-many Shortest Paths

Many-to-many algorithms have only been published for *Contraction Hierarchies* and *Highway Hierarchies*. The original concept of overlapping search spaces is published by Knopp et al. [2]. Overlapping search spaces also works with CHs, because CHs naturally use a bi-directional Dijkstra's algorithm, creating a shortest path that moves only "upward" in the hierarchy from both s and t. Geisberger et al. applied this concept to a time-dependent CH, but only showed results for Germany, which is a relatively small map, with a low percentage of time-dependent edges [7].

3 Approach

We start this section with an extensive explanation of Customizable Route Planning. Next, we show our approach to extend CRP to a many-to-many setting, with some improvements to keep running times acceptable. Lastly, we show how CRP can be used to calculate routes over graphs containing arcs with restricted driving times. We show multiple solutions to problems arising from this time-dependent version of CRP.

3.1 Customizable Route Planning

CRP uses a *partition-based overlay graph.* A *partition* of V is a set of cells $C = \{C_0, \dots, C_k\}$, $C_i \subseteq V$, with each $v \in V$ contained in exactly one cell C_i. A multilevel partition of V is a family of *partitions* $\{\mathcal{C}^0, \dots, \mathcal{C}^L\}$, where ℓ donates the *level* of *partition* $\mathcal{C}^\ell$. CRP uses nested multilevel *partitions.* This means that for each $\ell \leq L$ and each cell $C_i^\ell \in \mathcal{C}^\ell$, there exists a cell $C_j^{\ell+1} \in \mathcal{C}^{\ell+1}$ with $C_i^\ell \subseteq C_j^{\ell+1}$. Note that C_i^ℓ is a *subcell* of $C_j^{\ell+1}$ and conversely $C_j^{\ell+1}$ is a *supercell* of C_i^ℓ. Finally, we introduce the notion of *boundary* vertices and arcs. A *boundary* vertex on level ℓ is a vertex with at least one neighbour in another level-ℓ cell. A *boundary* arc on level ℓ is an arc connecting vertices in different level-ℓ cells.

The first preprocessing step of CRP is the creation of a multilevel partition of V. The aim is to partition in such a way that the number of arcs between cells is minimized. A lower number of arcs will reduce the amount of work that needs to be done during the second preprocessing stage. We use an open source program, KaHiP [31], to partition our graph, more on this in Section 4.

The second preprocessing step is called the customization step. During this customization step, CRP builds an *overlay graph* H. The core of this *overlay graph* is a copy of the union of all *boundary* vertices and arcs. If a *boundary* vertex has multiple *boundary* arcs, we make a copy of the vertex for each arc. Each of these copies is connected to exactly one of the *boundary* arcs. Note that a *boundary* arc at level ℓ is also a *boundary* arc on all levels below. For each cell C a *clique* is then built: For every pair (v, w) of *boundary* vertices in C an arc (v, w) is created, its cost is the same as the shortest path between v and w restricted to C (See Figure 1). These arc costs can be calculated by running Dijkstra from each *boundary* vertex, restricted to the cell. For calculating *Cliques* on higher levels, the previously constructed shortcuts can be used.

Using more partition levels (up to a certain point) leads to a faster customization, since each level can operate on smaller graphs. Adding more levels does however increase space consumption. Additionally, the increased overhead can make queries slower. To avoid this downside, Delling et al. used a concept called *phantom levels*, which are additional levels only used during customization [6]. Metric-dependent space is unaffected by the *phantom levels.* Do note however, that we still need to create the metric-independent overlay graph structures for the additional levels. We use one phantom level in our implementation, see Section 4 for full details.

After the two preprocessing stages, queries use H as follows: Similar to a regular Dijkstra search, a distance label $d(x)$ is stored for each entry x. x can be a vertex in V or a *boundary* vertex in H. Initially only s is put into a priority queue with value 0. Next, in each iteration of the algorithm, the minimum-distance entity e is taken from the priority queue. If e is a vertex in V, a scan is performed as in regular Dijkstra. If e is in H however, neighbours to consider are either reached by a *boundary* arc or through a *clique.* Note that the level on which to check neighbours depends on e, s and t. To determine the appropriate level, choose

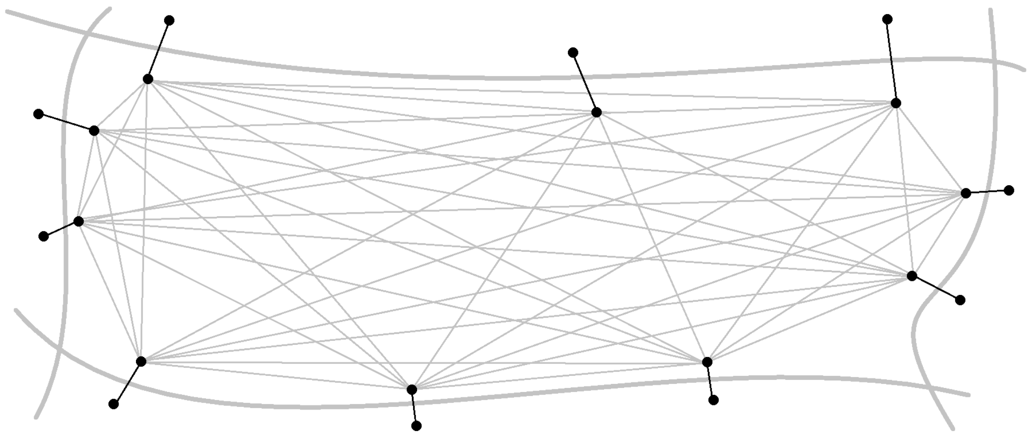

Figure 1: A cell with generated *clique*. The black vertices and arcs are *boundary* vertices and arcs.

the highest level cell containing e, but neither s or t. Remark that a level transition can occur by traversing a *boundary* arc. Finally, the neighbours are added to the priority queue with appropriate distance labels. When $e = t$ during a later iteration, a shortest path is found. Traversal of the cliques reduces the search space significantly, leading to fast running times. See Figure 2 for an example.

We can reduce the number of neighbourhood scans needed, by doing multiple steps at once. After relaxing a *clique* arc (u, v), we can immediately relax the connecting *boundary* (v, x) arc (only one by construction). The triangle inequality combined with our usage of full cliques guarantees we can safely execute the second relaxation without loss of optimality. v can only be updated from x, since all other neighbouring vertices are in the clique and the path through those vertices will thus be longer. We do not need to add v to the priority queue, saving us heap operations. Note that the only paths that are not considered using this double relaxation are the suboptimal ones that go directly from one *boundary* vertex to another via a third.

Implementation We implemented CRP in C/C++, most data is stored in a std::vector to increase performance. Our directed base graph is stored using two vectors that together act as an adjacency list. One vector V_1 stores all arcs (u, v) ordered by u. An arc object only stores destination v and cost c. The second vector V_2 stores for each vertex, the index of the arc vector from which its corresponding outgoing arcs are stored. All outgoing arcs of vertex i are now stored in $V_1[V_2[i]]$ up to $V_1[V_2[i + 1] - 1]$. Besides the adjacency matrix, we also store a vector containing cell id for each node.

The *overlay graph* is implemented using both vectors and maps for the various structures. We followed the example of Delling et al. [6]. A vector of overlay edge objects is used. Each containing information about their matching original edge, the neighbouring vertices and on which levels it exists. For each level we also store the id of the corresponding cell object. For each layer, we store a vector of these cell objects. A cell object stores which overlay edges connect to it, as well as the index from which this cells shortcuts are stored. The shortcuts for all cells are stored in one big vector. By using the index from a cell object, we

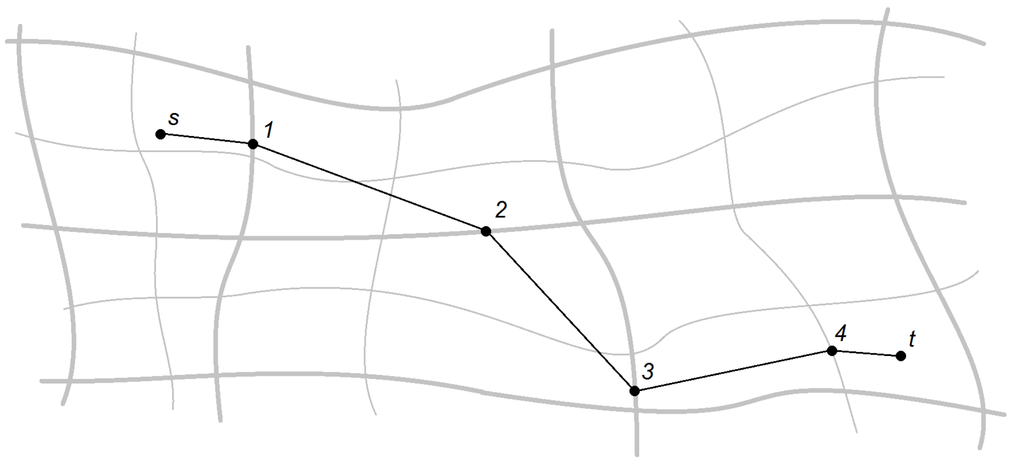

Figure 2: Shortest path found by CRP query. Parts s-1 and 4-t are found by a regular scan on the original graph. Other parts are calculated using the overlay graph H. Note that parts 1-2 and 2-3 are calculated on a higher level than part 3-4.

can find the correct shortcut entry. Additionally, we use a map to map from original vertex to its connecting overlay edge objects. This map is used to switch to the overlay graph when possible. Since we only ever add one of the two vertexes connected to an overlay arc to the priority queue, we instead store the index of the overlay edge object. When the overlay edge object is later popped and evaluated, we can use the stored cell id of the correct level to find the cell for which to relax all connecting clique arcs.

The priority queue used for CRP queries, is a binary heap that uses vectors to reduce heap overhead. We only store search node ids with their costs in the heap. These search node ids correspond to search node objects in a vector. A searchnode stores a boolean value indicating if it belongs to an overlay node or an original node. It also stores node id, whether it is still in the priority queue, and the tentative shortest distance. Lastly, we need two additional vectors to get from a graph node to a corresponding search node, one for the original nodes and one for the overlay nodes. For local Dijkstra queries, we do not need to mix two types of nodes and can thus use a regular binary heap as datastructure.

3.2 Many-to-many Queries

Besides calculating one-to-one queries, we can use CRP to find the shortest paths between a set of interesting locations. As discussed in the introduction, naively calculating $|S| \cdot |T|$ one-to-one queries is too slow for practical use. Instead we store search spaces and use the stored information to emulate the behaviour of $|S| \cdot |T|$ pairwise searches. We now describe this idea in detail.

We note that a shortest path found with CRP goes up in partition levels from s and then later on goes down towards t. This means that for each shortest path, there is a boundary vertex that is reachable from both s and t only going upwards in partition levels. This is the key property we exploit. The forward search spaces we store for $s \in S$ are one-to-all queries originating from S, searching only upwards in partition levels. Backward search spaces stored for $t \in T$ are analogous with one key difference: Edges are traversed in the other direction.

To find a shortest path from s to t, we combine the stored search spaces of the forward search from s and the backward search of t. Storing search spaces can be done using a vector of (v, d) pairs, where v is the vertex id and d the corresponding distance. For a vertex v that is stored in both search spaces, the distance of the shortest s, t-path through v is $dist(s, v) + dist(v, t)$. The shortest of these paths is the actual shortest path from s to t. Geisberger et al. proved that this is indeed an optimal path for up-down-paths in CHs [4]; the same concept applies for CRP. If we sort all search spaces on vertex ids, we can easily check the intersection of two search spaces using a linear scan over the sorted vectors.

Top level exploration. For larger graphs, most of the calculations are done in the upper levels, where clique-sizes are bigger (See Table 5). The top level is fully explored in both the forward and backward searches. We used this observation to do a simple, but very effective improvement. Namely stopping backward searches when they reach the top level. This change shaves off almost half the calculation time for sufficiently large graphs as can be seen in Table 1.

$\|S\| = \|T\|$	100	1 000	10 000
Regular many-to-many	217	5 125	427 171
Backward searches stopped at top level	170	2 607	206 672

Table 1: Overview of the impact of stopping backward searches when they reach the top level for many-to-many queries. Reported are the query times in ms. All queries are run on a graph of Western Europe.

Parallelization. The search spaces can be calculated independently, this means that the process can easily be parallelized. We used OpenMP to parallelize our software. For sufficiently large $|S|$ (> 20), we needed 3.4 times as little time creating search spaces by splitting the work over our 4-core machine. Checking the intersection of two search spaces can similarly be parallelized. This holds true even when using the bucket-like structure explained next. The theoretical improvement should be 4, but the overhead of initializing the other threads and the limited cpu-cache size diminish the improvement. Data-dependencies can also potentially slow down parallelized programs, but the parts we parallelize do not read data written by other threads, so this should not be an issue in our case. Parallelized computing is still possible when using the bucket-like structure explained next.

Bucket structure. Efficiently combining the information gained during the forward and backward searches can greatly improve performance. Instead of naively storing all data and intersecting $|S| \cdot |T|$ combinations of search spaces, we use a bucket-like structure to speed up the process [2]. We start by doing all the backward searches. We associate a bucket $b(v)$ with each node v of the graph. Each bucket stores pairs (t, d), representing paths from v to $t \in T$ with length d encountered during the backward searches. Next, we maintain a two-dimensional array A of tentative distances for each s, t pair, each entry is initialized to ∞. Lastly we do all the forward searches. Whenever a node v is settled, $b(v)$ is scanned. For every pair (t, d) stored in $b(v)$, we update $A[s, t]$ to $min(A[s, t], d(s, v) + d)$. $A[s, t]$ now contains the cost of a s, t shortest path. The use of this efficient bucket structure decreases calculation time for $|S| = |T| > 12$, even halving the calculation time when $|S| = |T| = 500$.

For higher set sizes this gain continues to increase, e.g. for $|S| = |T| = 10000$, calculations are 7.5 times faster.

When creating the buckets, we need to sort the vertex-, distance pairs (v, d) found during all the backward searches by vertex id. Following the idea of Knopp et al. we used a variation of counting sort [2]. This works perfectly fine for the overlay graph, but the base graph has little data entries (we only get data for the cells containing a $t \in T$) over a relatively large range (the size of the graph compared to the size of the overlay graph). We found using quick sort on the vector of pairs, combined with hash tables specifying index and size of the buckets for each node v, to give better results for the base graph. The hash tables are used during forward searches to quickly find the bucket corresponding to the current node v. When using counting sort, this can be done with a simple vector. Table 2 shows that using a combination of the two sorting methods gives good result for smaller set sizes, while keeping the power of counting sort for big set sizes.

Sorting method		$\|S\| = \|T\|$			
		100	1 000	10 000	20 000
overlay graph	base graph	time (ms)	time (ms)	time (ms)	time (ms)
counting sort	counting sort	393	1 409	27 302	85 899
counting sort	quick sort	143	1 349	26 913	83 002
hash table	quick sort	193	2 193	90 558	312 417

Table 2: Overview of different sorting methods used when creating the bucket-like structure used in many-to-many queries. Queries are run on a graph of Western Europe.

Pruning clique arcs The shortcuts stored in the clique arcs are calculated restricted to their cells. This means that some of these arcs are never part of a shortest path, i.e. the shortest path could go through another cell. These arcs can safely be removed from the overlay structure without sacrificing optimality. We can add a preprocessing step that runs Dijkstra locally for each clique arc, in the hopes of finding a shorter path through another cell. We tested this approach and were able to prune away 24% of the stored clique arcs. The average number of edge relaxations dropped from 287 678 to 152 689, the number of priority queue entries stayed the same however (all nodes are still reachable). It turned out that calculation time of the priority queue heap operations far outweigh the time needed for the simple check done when relaxing an edge. No significant improvement could be measured in running time for either one-to-one or many-to-many queries using the pruned cliques. We did reduce the RAM usage a little, but most RAM is consumed by the base graph, not the overlay graph. We revisited pruning after adding driving restrictions (See below) since saving some space is more important then. We were only able to prune 2% of the edges, saving hardly any memory and leading to even less impact on running time than in the static case. The idea of pruning clique arcs was then discarded all together.

3.3 Driving restrictions

Many countries impose driving restrictions for transport trucks during the weekend or night. Such restrictions greatly influence shortest route calculations: if a truck is unable to enter a country during the weekend, it will have to drive around the country. We aim to extend our routing algorithm to correctly handle legislation that imposes driving restrictions. Table 3 shows the driving restrictions for countries in western Europe.

	Night	Saturday	Sunday	
Austria	22:00 - 5:00	15:00 - 24:00	0:00 - 24:00	
Czech Republic			13:00 - 22:00	
France		22:00 - 24:00	0:00 - 24:00	
Germany			0:00 - 22:00	
Luxembourg		21:30 - 24:00	0:00 - 21:45	
Hungary		15:00 - 24:00	0:00 - 22:00	Not in July and August
Italy			8:00 - 22:00	
Slovakia			0:00 - 22:00	Only motorways
Slovenia			8:00 - 21:00	Only motorways
Switzerland	22:00 - 5:00	22:00 - 24:00	0:00 - 24:00	

Table 3: Truck driving restrictions for European countries. The given times indicate the hours during which the routes are blocked for truck traffic.

Travel times on graph arcs with driving restrictions can no longer be represented with a static cost. Instead of storing static arc values, *travel time functions* need to be stored. Every arc a has a *travel time function* $f_a : \mathbb{W} \rightarrow \mathbb{R}^+$ mapping departure time (within some period $\mathbb{W}$) to a positive travel time. In our case $\mathbb{W}$ represents a week with each time step representing 0.1 seconds ($\mathbb{W} = \{w \in \mathbb{N} | 0 \leq w < 6048000\}$). When only dealing with driving restrictions these functions have very specific properties. When departing a during a restricted interval, you have to wait for the restriction to lift. f_a will have a slope of -1 during this period. If you depart just before the restriction goes into effect, but can not fully traverse the arc, you need to wait the full restricted interval. Departing at any other point in time, leads to the same travel time as an unrestricted arc. See Figure 3 for an example.

A *Travel time function* is implemented by storing a vector of *segments*. Each *segment* contains a $w \in \mathbb{W}$, a cost $c \in \mathbb{R}^+$ and a boolean representing the slope of the segment leading up to the point (w, c), either 0 or -1. Vertical *segments* are stored implicitly whenever two *segments* with slope 0 are stored consecutive. Using this method the function of Figure 3 can be represented using just 3 *segments*. Additionally we store the minimum f^{min} and maximum f^{max} cost, which can later be used to avoid costly calculations.

When querying on a time-dependent CRP-structure, two key changes need to be taken into consideration. First, we can assume the time-dependent functions are piecewise linear with complexity $|b|$, where b is the number of breakpoints. Concatenating two of these functions, called *linking*, with complexity $|c|$ and $|d|$ is no longer a simple addition, but can result in a new function with higher complexity $|c + d|$. Finding the combined minimum of two functions, used when update tentative distance labels, denoted *merging*, can also result in complexity $|c + d|$. Secondly, paths no longer strictly dominate each other, which means that vertices might have to be re-added to the priority queue at a later stage (the priority queue sorts on function minimum). This happens with 38.5% of the vertices with tests on our Western Europe network.

We formalize *link* and *merge* operations as follows: Linking two *travel time functions* f and g is defined as $\text{link}(f, g) := f + g \circ (\text{id} + f)$. On the other hand, merging f and g is defined by $\text{merge}(f, g) := \min(f, g)$. $\circ$ is mathematical function composition i.e. the point wise application of one function to the result of another to produce a third function. Both

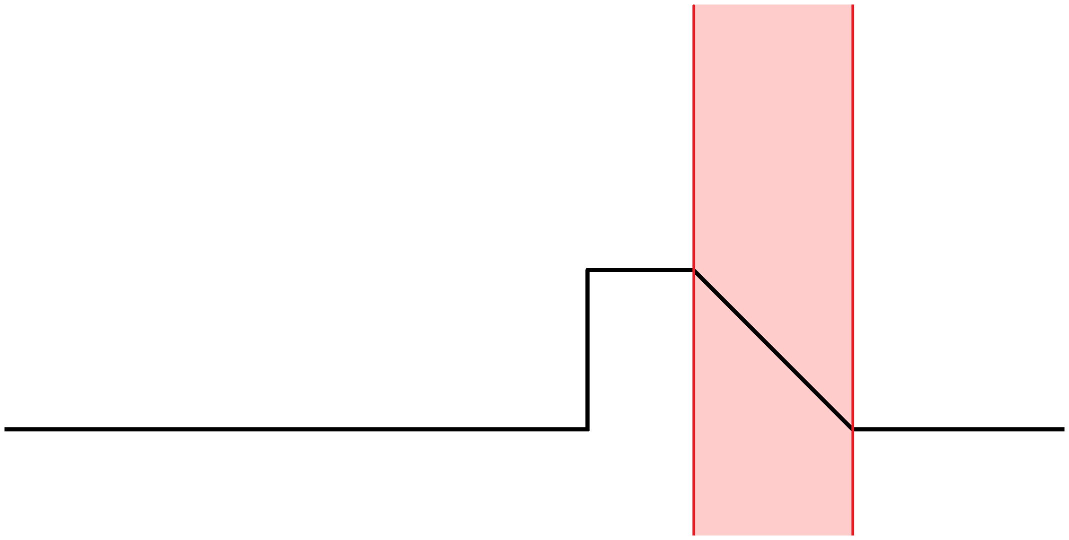

Figure 3: Travel time function of a graph arc. The horizontal axis represents a period $\mathbb{W}$ of possible departure times. The vertical axis represents the driving time needed to traverse the path for these departure times. The red area represents the period during which the path is blocked.

the *link* and *merge* operations are implemented using a linear sweep over the breakpoints in the functions. Using half-plane tests and line intersections, the new breakpoints and their corresponding *segments* can be calculated. During the sweep, the minimum h^{min} and maximum h^{max} of the resulting function h, can be calculated as well.

We can generate *travel time functions* for paths using these *link* and merge operations. When traversing a path of arcs $[a_1, \ldots, a_k]$, we can obtain the *travel time function* TTF by applying the *link* operation iteratively, i.e. $TTF = \text{link}(a_1, \text{link}(a_2, \text{link}(\ldots, a_k)))$. Given a set of $s-t$ paths P, the minimum $s-t$ profile MP over all paths in P is defined by $f_P(\tau) = \min_{p \in P} f_p(\tau)$ for $\tau \in \mathbb{W}$. This profile maps each departure time τ from s to a minimum travel time for the given paths. MP can be obtained by merging the respective paths.

We distinguish two types of one-to-one queries that can be executed using a time-dependent version of CRP. First, *Earliest Arrival Time* (EAT) queries ask for the minimum travel time (from s to t), when departing at a given time τ. Secondly, a *Travel Time Profile* (TTP) query asks for a full $s-t$ *travel time function*, i.e. it asks for the minimum travel time for each possible departure time τ.

EAT queries can be calculated using a time-dependent variant of *Dijkstra's algorithm* [3]. It still maintains a priority queue with tentative costs $d(\cdot)$. The costs in this case however, are the tentative earliest arrival times for the given departure time τ. When a vertex u is extracted from the priority queue, all outgoing arcs (u, v) are relaxed, updating the tentative cost of v, whenever $d(v) < d(u) + f_{(u,v)}(d(u))$, accordingly. As in a regular Dijkstra search, v is then added to the priority queue.

Solving TTP queries can be done similarly with some key changes. The priority queue is sorted on minimum travel time $f_v^{\min}$, with f_v the profile of vertex v. Initially $f_s = 0$ and $f_v = \infty$ for all other vertices. When u is extracted from the priority queue, travel time profiles are propagated instead of travel times. For each outgoing arc (u, v), $g := \text{link}(f_u, f_{(u,v)})$ is calculated, then we set $f_v := \text{merge}(f_v, g)$. If f_v changed, we update the priority queue accordingly. Note that u can be reinserted at a later stage, when part of its non-minimal profile is improved by one of its neighbours. Because vertices can be re-added to priority queue the stopping criteria becomes $f_u^{\min} < f_t^{\max}$.

Using the stored minimum and maximum travel costs mentioned earlier, we can skip some expensive calculations [18]. Before relaxing arc (u, v)), we check if $f_u^{min} + f_{(u,v)}^{min} > f_v^{max}$, i.e., the minimum of the linked profile exceeds the maximum of the label of v. If this is the case, we don't need to update the label. Similarly, merging can be omitted if $g^{min} > f_v^{max}$. We can save additional computations by creating special cases for linking constant functions. If both functions are constants, linking becomes a trivial addition. When exactly one of the functions is constant, linking can be done by simply shifting values for each *segment* of the non-constant function by the constant. Because linking is not commutative, we need to distinguish two cases, depending on which of the functions is constant: If $\text{link}(f, c)$, with c a constant, we increase the cost of each *segment* by c. Conversely, if $\text{link}(c, g)$, we also need to translate the whole function by $-c$ over the w axis. Only when neither function is constant, do we apply the expensive link operation.

Foschini et al. showed that the best theoretical upper bounds for both the number of breakpoints of the final $s-t$ profile, and thus the space consumption of the stored profiles, are superpolynomial [32]. In our test instances the highest number of breakpoints encountered was less than a thousand. Even though this is far from the stated upper bound, we still need to improve on the regular TTP query to make it feasible for usage on the large-scale graphs. Both the space consumption and the number of calculations needed will have to be reduced with varying techniques.

Symbolic representation To reduce memory consumption, we introduce the notion of a symbolic function representation. A symbol is assigned to each set of driving restrictions: one for each country in Table 3 and one for the unrestricted case. A travel time function that exactly fits one of the symbolic restrictions can now be stored by only storing the appropriate symbol and *freeflow* travel cost. The *freeflow* travel cost is the cost of the path when non of its arcs are restricted, i.e. when you can freely drive the path without waiting for closed of roads. When needed, we can convert the symbolic representation, using the restrictions set by Table 3 and the *freeflow cost.* Note that all the original graph arcs can be stored symbolic.

An extension to these symbolic functions can be made by storing two additional values. One for the translation over the w axis and another for the additional cost induced by linking constants, denoted *ac*. Using this extension, linking two functions of the same type will result in a travel time profile that is symbolically storable more often.

Symbolic function representation after linking is only possible in the following two cases: Firstly, whenever a symbolic function links with a constant, we can simply add the constant to *ac* and, depending on the order of linking, possibly the translation. Secondly, the linking of two symbolic functions that share symbol and translation. The resulting function will have the same symbol and translation, the cost will be the sum of the original costs and *ac* will be the sum of the original *ac*s. In all other cases we need to transform the symbolic functions to their full representation and use these for a regular link. In practice symbolic links will often occur, since a single restriction type stretches over a full country (Experiments on our Western Europe map showed 58.4% of the search nodes were stored symbolically when creating backward search spaces for many-to-many queries). Switching from a full representation to a symbolic one reduced the memory requirements of the time-dependent graph by more than two-fold.

Low level engineering The memory footprint of stored functions is further reduced by using some low level optimizations. Our vector of travel time functions is in reality a vector of pointers, each pointing to an object with data. Our cpu prefers padded bitwords to reduce cache-misses, which means all pointers compiled on our system have even pointer addresses. We can utilize this fact by storing our symbolic functions in line: we use 63 of the 64 bits usually used by the pointer to store our symbolic function. The last bit has to be set to 1, to show it is not a real pointer. When reading from our vector of travel time function pointers, we now check the last bit. If it is odd, we interpret the pointer itself as a symbolic function. If it is even, we evaluate the pointer and get our regular function object from the address pointed to.

Additional memory savings We originally assumed memory requirements would be the greatest challenge to overcome when handling time-dependent CRP. We overestimated memory requirements for many-to-many queries, however. A single search space on a graph of Western Europe only consists of a little over 3000 nodes, many of them on lower levels, thus having less breakpoints. By using bitlevel engineering and a smart representation of functions, memory usage has become very acceptable (See Figure 8). We did have three additional ideas to reduce the memory usage of time-dependent CRP further. The ideas were to only partially store search spaces, to contract nodes into single entities or to store shared parts of search spaces only once. All would greatly affect calculation time in a negative way. This is a big sacriface to make for little extra gain, given the already low memory footprint. The rest of our time was thus used focussing on reducing query times, which meant that we discarded these ideas.

Clique flags With the principle of *clique flags*, we can decrease time-dependent query times [18]. The idea is to stall nodes that are unlikely to be part of any shortest path, de-stalling happens when this assumption is flawed. For each vertex of the overlay graph, we add a flag to its cost that is true if the vertex was only updated from nodes belonging to the same cell. Whenever the corresponding cost is replaced after relaxing a clique arc, this flag is set to true. If, after relaxing a boundary arc, the cost is (partially) improved, the flag is set to false. We don't need to relax outgoing arcs of flagged vertices, since this can not possibly improve costs of other vertices. If a vertex v is flagged, all vertices, that can be reached through clique arcs from v, will not improve after relaxing v. This is because of the triangle inequality and our usage of full cliques, a path going through an extra point will never be shorter. By construction each boundary vertex v has only one boundary arc (v, u). By construction, u is updated whenever v is updated from a clique arc. This means that u, the vertex on the other end of the boundary arc will already be optimal with respect to paths through v. All neighbouring vertices are either reached through clique arcs, or through the boundary arc, so no possibly improvement does indeed exist.

Function Approximation To reduce the running time of the time-dependent queries we looked into function approximation. We believed we had a working approximation algorithm that could approximate a function within a given margin of error. All of our travel time functions are piece-wise linear, where each part has a slope of 0, -1 or is a vertical line. The idea is to find a line, that is either vertical or has a slope of 0 or -1, such that as many consecutive points, starting from the first breakpoint, lie within given error margin of this line. If the first k points are within error bounds of line ℓ, we have found our first approximated segment. We reiterate this process starting from breakpoint k, until we fully approximated the function. Internally this is done by doing a linear scan over all breakpoints while keeping track of a line for each of the three possible directions. For every breakpoint we have to recalculate if there exists a line, for each direction, such that no bounds are broken. Once we get to a breakpoint k where all lines break given error bounds, we choose the feasible line from breakpoint $k-1$, with lowest error margin and restart from $k-1$. All chosen lines are merged into one function, which is our approximation. Since travel time functions are periodic, we also need to do some additional calculations to make sure the start and end of the period have the same value.

Tests using this algorithm yielded peculiar results. Replacing all the clique arc costs which had an approximation with less breakpoints, could be done relatively quickly (a matter of seconds). Querying on this inexact, theoretically easier graph representation should be quicker, but instead it was 2-10 times slower on our test cases. This either means there is a mistake in our algorithm, or queries behave strange with approximated results. A possible explanation could be that because of slight function deviations after approximation, the number of vertices that are re-added to the priority queue is increased. In either case, we did not have enough time to fully investigate what caused the strange results.

3.4 Many-to-many queries with driving restrictions

Many-to-many calculations can still be done with time-dependent costs, but will obviously take a lot longer to execute than for the static case. When storing a search space we need to store travel time profiles instead of static costs. All the techniques used to speed up the static case many-to-many queries can be used for the time-dependent version as well. This means we can apply parallelization and the bucket technique discussed in Section 3.2 to speed up the process. *Clique flags* and symbolic function representation further help to reduce memory usage and running time. Combining all these techniques, we can get exact many-to-many results for smaller set sizes ($|S| = |T| < 50$) on a continental sized graph with driving restrictions, in under a minute. See Section 4.

4 Experiments

All algorithms are implemented in C++ using Microsoft (R) C/C++ Optimizing Compiler Version 19.00.24215.1 (flag /O2). Experiments were conducted on a dual 4-core Intel i7-4819MQ @ 2.8 GHz, with 16 GiB of DDR3-1600 RAM, 6 MiB of L3 and 256 KiB of L2 cache. We ran partitioning and many-to-many queries in parallel (using all 8 threads) and customization and one-to-one queries sequentially.

Input data All test instances are based on the road network of Germany and Europe to reflect literature. Real road networks were available for varying detail levels, we chose the networks of Germany and Western Europe that resembles the networks used in literature as much as possible. We also test on the biggest network available to us, that of full Europe (74 million nodes). When considering time-dependent arcs, we use the road blocks as mentioned in Table 3.

Partitioning Partitioning was done using KaHiP, which is a family of partitioning programs, designed to handle big graphs. We used the STRONG KaFFPa variant [31]. We iteratively split the graphs into 16 cells, until the cells were no bigger than 25 nodes. 16 was chosen because 2 partition ids could then easily be encoded in a byte and it was one of the values showed to work well by Delling et al. [6]. This means we use a 5- or 6-level partition depending on graph size. The level containing the smallest cells is only used during customization as a *phantom level*, again following Dellings example. Computing the partition took 67 minutes for Germany, 127 minutes for Western Europe and a little over a day for Europe. This is slower than in related work, but it is not the focus of this work and sufficiently fast to test with.

We considered using a country border respecting partitioning to decrease calculation times for time-dependent queries on our Western Europe test instance. If cells only contain one type of driving restriction, linking and merging is easier. We created a country border respecting partition of Western Europe by reducing the weight of edges between nodes with the same restriction type (The partitioning algorithm minimizes the total weight of arcs between cells). Changing the weights forced KaHiP to cut the graph on country borders with different driving restriction types. This resulted in more cells with only one type of restriction. Unfortunately, after testing we had to conclude a regular partition yielded better results on all fronts. Memory usage was comparable, but query times increased to at least double by using the country border respecting partitioning. By forcing the partition to respect borders we increased the number of boundary arcs, in turn creating more clique arcs on the highest level (256 845 vs 772 963). The lower levels were unaffected, likely because no further borders were available to respect. Easier linking and merging did not outweigh the overhead of all these extra arcs on high levels.

Network	# Vertices	# Arcs	Time-dependent arcs
Germany	6 208 757	14 551 536	14 550 416 (100.0%)
Western Europe	16 050 802	32 025 621	10 886 365 (34.0%)
Europe	73 986 416	170 421 556	60 701 703 (35.6%)

Table 4: Network properties. The number of vertices and arcs of the graph are reported, as is the number of time-dependent arcs according to Table 3

Customization Table 5 shows details on customization for the different test instances. We report the single-threaded customization times, average clique sizes and the number of boundary and clique arcs for each partition-level. Note that the number of boundary vertices is exactly double the number of boundary arcs by construction. For the time-dependent case we additionally report the percentage of time-dependent clique-arcs. We can clearly see that the number of non-symbolic time-dependent clique-arcs increases at higher levels. For static cost networks, lower levels need more calculation time, since they have more cells and thus more shortcuts to calculate. For time-dependent networks however, this is not always true. The added complexity of linking and merging Travel Time Profiles at higher levels with more complex functions, explains this behaviour. Even though the Germany network has less nodes, the top level partition was of worse quality, i.e. the number of boundary arcs is higher resulting in more clique arcs that need calculating. For static cost networks, the customization times are fast enough to customize cost-functions in real-time.

Network	TD		Lvl 1	Lvl 2	Lvl 3	Lvl 4	Lvl 5	Total
Germany	Both	Boundary arcs	2 408	15 263	85 944	549 089		652 704
	Both	Clique arcs	409 914	982 279	2 389 765	6 193 189		8 582 954
	Both	Avg clique size	151	60	21	8		9
	Yes	Ns. arcs (%)	97.8	97.4	92.9	83.6		88.0
	Yes	Time (s)	296	220	181	263		960
	No	Time (s)	0.8	0.9	1.8	2.9		6.5
Western Europe	Both	Boundary arcs	1 747	14 230	85 700	501 862	3 379 077	3 982 616
	Both	Clique arcs	256 845	1 000 072	2 532 223	5 903 018	16 274 649	22 177 667
	Both	Avg clique size	109	56	21	8	3	4
	Yes	Ns. arcs (%)	97.1	74.0	58.9	49.1	37.6	48.6
	Yes	Time (s)	72	37	51	102	437	699
	No	Time (s)	0.7	1.0	1.4	2.4	4.0	9.8
Europe	Both	Boundary arcs	3 028	22 204	152 840	984 145	7 199 230	8 361 447
	Both	Clique arcs	843 192	2 620 292	7 956 489	20 336 909	66 552 868	86 889 777
	Both	Avg clique size	189	87	37	15	7	7
	Yes	Ns. arcs (%)	87.9	67.9	50.1	38.5	28.8	34.2
	Yes	Time (s)	3272	1330	757	650	963	6973
	No	Time (s)	3.8	4.7	6.4	12.4	27.4	57.4

Table 5: Performance of customization. Reported are the number of boundary and clique arcs per level, as well as the average clique sizes. The running times per level and, in the case of time-dependent networks (TD = Yes), the number of non-symbolic (time-dependent) clique-arcs (ns. arcs) are also reported. The smaller networks use fewer levels of customization, which explains the empty entries.

One-to-one queries Details on one-to-one query performance can be seen in Table 6. For each map, we report the static case one-to-one query times, calculated over 100 000 queries. Earliest Arrival Time queries were calculated over 20 000 queries and Travel Time Profiles over 100 queries. In each of these queries, source, target and departure time (where applicable), were chosen uniformly at random. Travel time is used as cost-function and it is calculated by dividing the distance of an arc with the average travel speed on the arc, kindly provided by ORTEC. Static cost and EAT queries were tested for correctness against a (time-dependent) Dijkstra's algorithm. TTP queries were in turn tested against EAT-queries with random departure time. Time-independent queries have a competitive running time in the field (See the paragraph about related work below). No published work handles networks with time-restricted arcs, so the time-dependent queries are harder to evaluate[1]. EAT queries are very fast and thus very usable in practice. The difference between the EAT queries of Western Europe and Germany seems odd, but can be explained by noting that Germany has 100% restricted edges (vs. 34% for Western Europe) as well as a denser top level with more clique arcs. Calculating a full Travel Time Profile, takes several seconds. This is fast enough for single queries, but as a subroutine, this might be to slow too use, depending on the application.

[1]Work with time-dependent queries has been published, see e.g. [8, 9, 18, 21]. However, it is unfair to compare to their work, since they use a broader and more difficult definition of time-dependency (Modelling congestion instead of blocked roads) and often use approximations instead of exact results.

Network	Static (ms)	EAT (ms)	TTP (s)
Germany	2.33	8.39	0.82
Western Europe	2.22	5.94	5.30
Europe	6.02	20.47	54.62

Table 6: Performance of one-to-one queries. Query running times are reported for time-independent queries (Static), Earliest Arrival Time queries (EAT) and full Travel Time Profiles (TTP).

Exact time-dependent routing is known to be computationally expensive. If we did not apply all our space and query time reducing optimizations, our results would be far worse. Table 7 shows some measurements of the improvements on the Western Europe test instance. Query speeds improved across the board, many-to-many queries even improved 7-fold. Memory usage is cut in half using the symbolic representation. Our Europe instance used 15 of the 16 available GiB of RAM loaded into memory, without symbolic calculations this network would not fit in memory. When using all improvements, 95% of the calculation time is spend linking and merging travel time functions, 84% of these calculations are performed on the highest level of our overlay graph. This highest level has the biggest cliques and because of vertices that get re-added to the priority queue, an enormous amount of clique arc relaxations need to be done. To further improve running times of CRP with time-dependent driving restrictions, either the number of arc relaxations needs to be reduced, or the link and merge operations need to be further optimized.

	EAT		TTP		$\|S\| = \|T\| = 50$		$\|S\| = \|T\| = 100$	
Improvements	Time (ms)	Mem.	Time (ms)	Mem.	Time (s)	Mem.	Time (s)	Mem.
None	8.71	7.0	38.9	7.0	595	7.5	887	7.5
All	5.94	3.4	5.3	3.4	80.7	3.6	160	3.6

Table 7: Performance of various queries on the network of Western Europe. Results are shown for either using all time-dependent improvements discussed in this thesis or non. Reported are Earliest Arrival Time (EAT), Travel Time Profile (TTP) and many-to-many queries. For each, query time and memory usage are reported. Memory usage is in GiB.

Many-to-many queries Table 8 provides an overview of the many-to-many query performance. Reported are the parallelized running time and memory usage of many-to-many queries on our test instances for different source and target sets. The nodes in these sets are again chosen uniformly at random. Time-independent queries were run n times, such that $n \cdot |S| > 100000$. This means that a query with $|S| = |T| = 10$ was run 10000 times, while a query with $|S| = |T| = 10000$ was only run 10 times to reduce calculation times. This gives a nice balance between sufficiently large test sizes and acceptable running times. For time-dependent queries a similar approach was used, but such that $n \cdot |S| > 5000$. Time-dependent queries return $|S| \cdot |T|$ Travel Time Profiles. Given these results, a source $s \in S$, a target $t \in T$ and a departure time: travel-cost can simply be looked up in $O(1)$ time with the returned structure. By using symbolic representations we did not run into memory issues. Memory issues do arise however, when querying very large source and target sets.

(We ran out of memory for $|S| = |T| = 50000$ on the Western Europe instance with static arc costs). All many-to-many queries were tested against their respective one-to-one variants, since Dijkstra's algorithm would be too slow for many-to-many queries. Running times for static cost queries are very small and even large queries ($|S| = |T| = 10000$) can be executed in well under a minute. Queries, taking into account road-blocks, take a lot longer, growing over a minute for even small set sizes. Do note however: these results are exact.

		Static		Time-dependent	
Network	$\lvert S\rvert = \lvert T\rvert$	Time (ms)	Memory (GiB)	Time (s)	Memory (GiB)
Germany	10	21.05	0.8	3.59	1.5
	20	32.26	0.8	6.14	1.5
	30	43.51	0.8	9.34	1.5
	50	67.46	0.8	15.8	1.5
	100	126.8	0.8	34.0	1.5
	200	254.4	0.8	61.5	1.5
	300	388.3	0.8	91.8	1.5
	500	781.7	0.9	151	1.6
	1 000	1 321	0.9	307	1.7
	2 000	2 864	0.9	675	2.0
	3 000	4 688	0.9	1 067	2.2
	5 000	8 896	1.0	1 892	2.7
	10 000	23 822	1.5	$>$ 2 000	$\geq$ 2.7
Western Europe	10	46.56	2.6	24.9	3.6
	20	55.59	2.6	33.7	3.6
	30	65.58	2.6	49.4	3.6
	50	89.57	2.6	80.7	3.6
	100	147.0	2.6	160	3.6
	200	226.9	2.6	251	3.6
	300	402.2	2.6	471	3.7
	500	650.6	2.6	838	3.7
	1 000	1 338	2.7	1 813	3.7
	2 000	2 938	2.7	$>$ 2 000	$\geq$ 3.7
	3 000	4 957	2.7	$>$ 2 000	$\geq$ 3.7
	5 000	9 672	3.0	$>$ 2 000	$\geq$ 3.7
	10 000	26 938	3.4	$>$ 2 000	$\geq$ 3.7
Europe	10	104.3	9.5	31.6	14.9
	20	131.9	9.5	54.3	14.9
	30	160.8	9.5	77.8	14.9
	50	218.4	9.5	131	14.9
	100	355.7	9.5	251	14.9
	200	686.5	9.5	499	14.9
	300	987.3	9.5	751	14.9
	500	1 520	9.6	1 251	15.0
	1 000	3 820	9.6	$>$ 2 000	$\geq$ 15.0
	2 000	8 083	9.6	$>$ 2 000	$\geq$ 15.0
	3 000	12 333	9.7	$>$ 2 000	$\geq$ 15.0
	5 000	23 416	9.8	$>$ 2 000	$\geq$ 15.0
	10 000	51 050	10.2	$>$ 2 000	$\geq$ 15.0

Table 8: Performance of many-to-many queries. Query times and memory usage are given for both time-independent queries (Static) and time-dependent queries. All queries calculate a full cost matrix from a source and target set with size $|S| = |T|$. Time-dependent matrix-entries contain Travel Time Profiles.

Local Queries In real-world applications, most queries tend to be local. To evaluate CRP on queries of varying ranges, we have plotted one-to-one query times against the Dijkstra rank [23]. When Dijkstra's algorithm is run from s, the Dijkstra rank of a vertex v is r if v is the r-th vertex evaluated. This means that vertices with lower Dijkstra rank are more local than those with higher Dijkstra rank. Dijkstra rank is a better measure than for example the distance between two vertices since it accounts for number of arc-relaxations needed. It works well equally well on dense areas as on sparse areas of a map. We run 500 queries for each rank tested, with s chosen uniform at random on our Western Europe map. Static cost results can be found in Figure 4, while the comparison with Travel Time Profile queries can be found in Figure 5 (20 queries each rank).

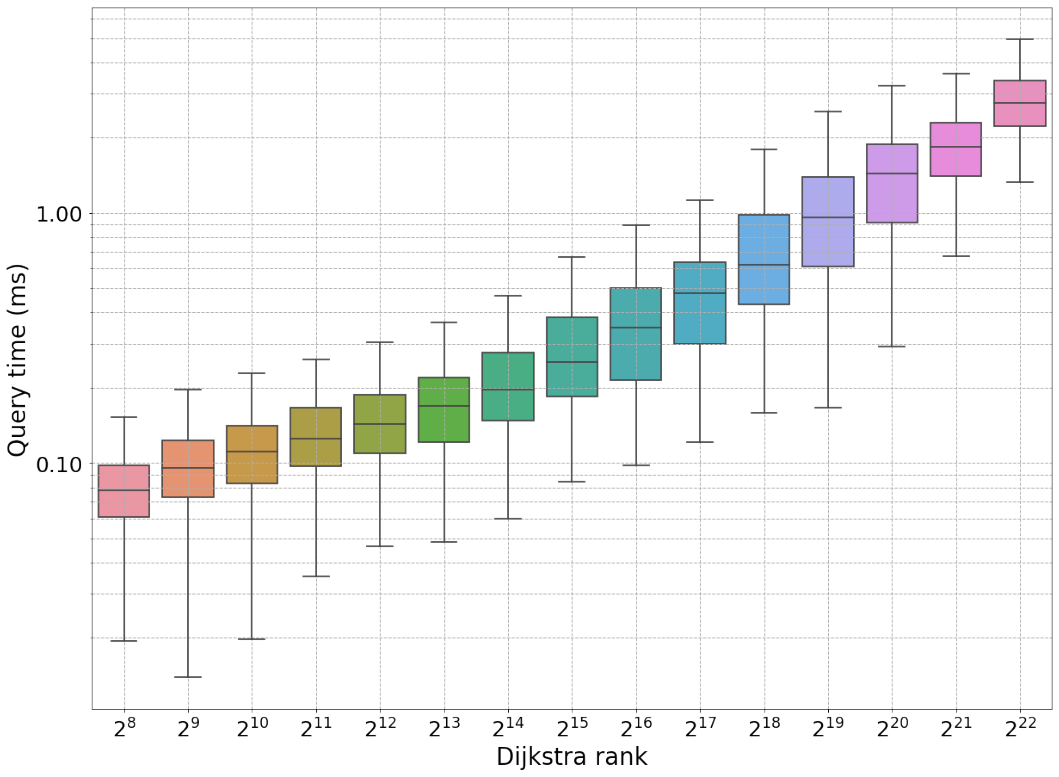

Figure 4: Performance of one-to-one static cost CRP for various query ranges. The queries are calculated on the Western Europe map. Note that the scale of the y-axis is logarithmic.

The plots show that local queries are much faster than global ones. Up to ranks of 2^{16}, static queries can be performed in less than 1 ms. Time-dependent queries show a wider spread of data points for each rank, this due to the fact that time-dependent query results can vary greatly depending on start vertex. Querying in countries without restrictions is way faster than querying in areas with travel restrictions. This observation combined with the relatively small test-set (20 queries each rank) also explains the odd values of Dijkstra ranks 2^{14} and 2^{19}. The upwards trend however, is clear: Local queries are a lot faster than global ones.

When calculating many-to-many queries, locality has no influence on running time. Many-

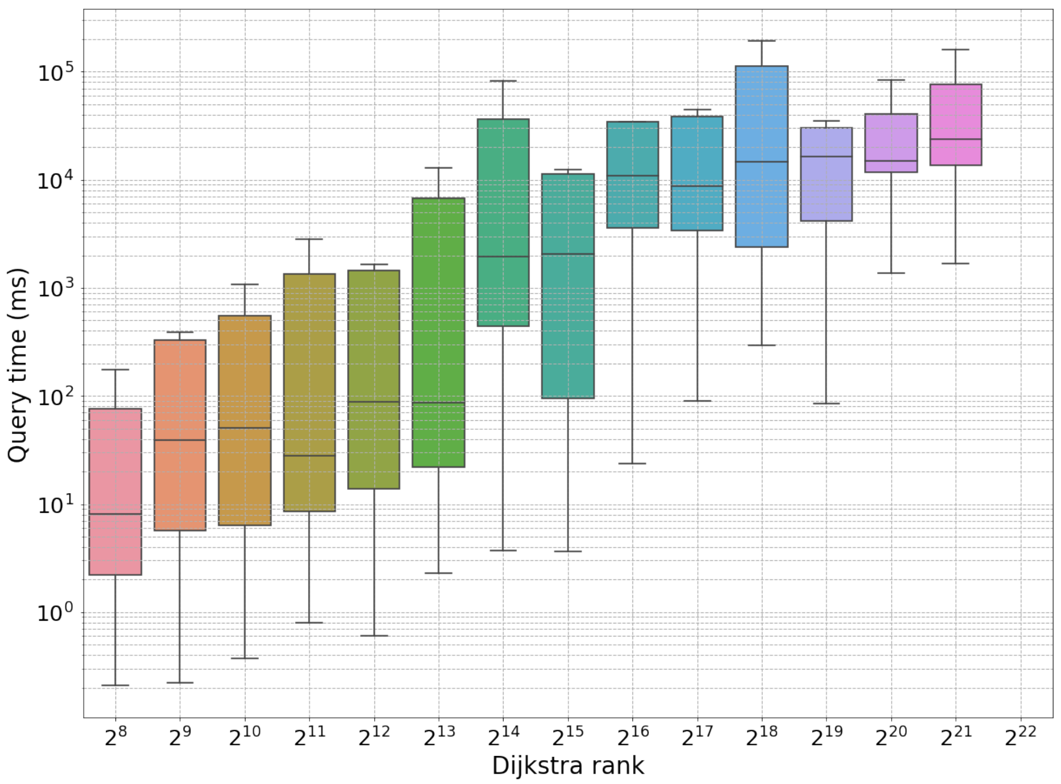

Figure 5: Performance of one-to-one Travel Time Profile queries for various query ranges. The queries are calculated on the Western Europe map. Note that the scale of the y-axis is logarithmic.

to-many queries use fully explored search spaces and these still have to be calculated when points are local, to ensure correctness. This was confirmed in practice with tests. An interesting consequence of this fact is that for small, local vertex sets S and T, it is faster to calculate $|S| \cdot |T|$ one-to-one queries, than calculating the result matrix using the search space intersection technique. For queries of Dijkstra rank 2^{14} for example, many-to-many queries up to $|S| = |T| = 16$ are faster when doing $|S| \cdot |T|$ one-to-one queries.

Performance against the field We compare our static cost one-to-one queries with the excellent work of Delling et al., who proved CRP to be a real contender in the field [6]. We also report running times for Dijkstra's algorithm [30] and those of Contraction Hierarchies by Geisberger et al. [4]. Both Delling and Geisberger used a network with 18 million vertices and 44 million arcs, most comparable to our Western Europe network. The findings are reported in Table 9. While not the focus of our work, we still managed to get fairly close to the results of Delling et al. with regards to the query time. Contraction Hierarchies are the obvious choice if query time is your only concern. CRP on the other hand has some benefits, the 2 phase preprocessing allows a quick change of cost function. Additionally, CRP can handle turn-restrictings without significant overhead (See Delling et al. [6]) and we believe it scales better with time-dependency, since no extra shortcuts will be added by extending to time-dependency. Our preprocessing times are far from Dellings work, but they were good enough to create the data structures needed to conduct our many-to-many work.

Method	Data from	Preprocessing time (s)	Query time (ms)
Bidirectional Dijkstra[a]	Bauer et al.	0	2 713.2
Contraction Hierarchies[b]	Geisberger et al.	480	0.15
Customizable Route Planning[c]	Delling et al.	706 + 0.37	1.85
Customizable Route Planning	This work	457 000 + 9.8	2.22

[a] AMD Opteron 270 @ 2.6 GHz, 16 GiB RAM, 2x1 MiB L2 cache, SUSE Linux 10.3
[b] AMD Opteron 270 @ 2.0 GHz, 8 GiB RAM, 2x1 MiB L2 cache, SUSE Linux 10.3
[c] Intel Xeon X5680 @ 3.33 GHz, 96 GiB RAM, 6x256 KiB L2 cache, Windows Server 2008

Table 9: Comparison of one-to-one query performance. Preprocessing time and query time are reported. If two numbers are mentioned for preprocessing time, they represent cost-independent preprocessing + cost-dependent preprocessing. The queries were done on a network of Western Europe with 16-18 million nodes, depending on method.

Comparing our time-dependent work is hard since no work has been published for time-dependent queries using only restricted driving times, as far as we are aware. The paper of Baum et al. is the closest we could find [18]. They used a time-dependent version of CRP to calculate shortest routes. Instead of using time-dependent driving restrictions, they used data of traffic patterns to simulate congestion. Several differences arise because of this. Travel time profiles of congestion don't have the same specific shape as profiles of driving restrictions. Slopes of different angles can exist, resulting in more complex travel time functions after linking or merging. We expect that the difference between $f^{\min}$ and $f^{\max}$ will be smaller with congestions than with driving restrictions, this is because restrictions can span over many hours, f.e. the night or weekend. As a consequence, vertices of congestion profiles will less often be re-inserted in the priority queue. The networks used in the paper of Baum et al. also have a limited set of time-restricted arcs (6.2% for their Western Europe instance) and their period only spanned a single day (compared to our week). To reduce running time, they use approximations, using an algorithm by Imai and Iri [33]. Travel Time Profile queries are not reported by Baum et al., but we can compare Earliest Arrival queries. They report EAT query times of 3.65 ms on Western Europe with an average error of 0.54 %, while we get exact results in 5.94 ms. The many differences in type of EAT query make the comparison hard to evaluate however. We can also compare the customization step. Baum et al. were unable to complete this step using exact calculations, because they ran out of memory. Allowing small errors they could complete the customization step in as little as 110.2 seconds, our exact customization took 699 seconds. Once more we want to emphasize that this comparison is unfair to either side for different aspects.

It is interesting to consider combining parts of the work of Baum et al. and our thesis. We believe that their implementation of time-dependent CRP can be easily extended to also calculate many-to-many queries. We do not know their Travel Time Profile query performance and can thus not predict the resulting running times, however. An interesting consideration for future work could be the usage of their approximation with our model of time-dependency.

The publications regarding many-to-many queries are close to non, the most recent work we could find was of Geisberger et al. [4], dating back to 2012. They used Contraction Hierarchies as underlying shortest path algorithm and only reported seconds required to calculate an $|S| \cdot |S|$ distance table. They used a network of Western Europe with 18 million nodes. We compare our results to their results, see Table 10. Included are also the results of Knopp et al. who originally developed the efficient many-to-many algorithm back in 2007 [2]. They used Highway Hierarchies. Our algorithm is outperformed by Contraction Hierarchies on all reported datasets. Note that the gap between CRP and CH is smaller for many-to-many queries than for one-to-one queries. The difference is also small enough to consider CRP for many-to-many queries, because of its added versatility (as mentioned in previous paragraph).

	$\lvert S\rvert$	100	500	1 000	5 000	10 000	20 000
Highway Hierarchies[a]	Knopp et al.	0.6	1.7	3.3	26.3	76.6	247.7
Contraction Hierarchies[b]	Geisberger et al.	0.4	0.5	0.6	3.3	10.2	36.6
Customizable Route Planning	This work	0.4	1.5	3.8	23.4	51.1	82.9

[a] AMD Opteron @ 2.6 GHz, 16 GiB RAM, 1MiB L2 cache, SUSE Linux 10.1
[b] AMD Opteron 270 @ 2.0 GHz, 8 GiB RAM, 2x1 MiB L2 cache, SUSE Linux 10.3

Table 10: Comparison of many-to-many query performance. Reported is the measure number of seconds needed to calculate an $|S| \cdot |S|$ distance matrix. The queries were done on a network of Western Europe with 16-18 million nodes, depending on method.

5 Conclusion

We have presented a many-to-many version of Customizable Route Planning. By careful engineering and applying many techniques of the latest research, we have shown that CRP performs well in a many-to-many setting. While Contraction Hierarchies offer a faster approach, Customizable Route planning can be used as a more robust alternative. CRP offers less sensitivity to small metric changes, because it can be recustomized quickly. Customizable Route Planning can perform many-to-many queries for big source and target sets, on continental sized road networks, in only a few seconds.

We also explored an exact time-dependent version of CRP that handles driving restrictions. We have shown that exact Earliest Arrival Time Queries and even Travel Time profiles can be calculated quickly. Even many-to-many queries can be handled if a waiting period of several seconds is acceptable. Our biggest contribution would have to be the symbolic representation of travel time functions, which allowed even the biggest available time-dependent networks to be loaded in main memory.

Future work can be done in several directions. To speed up time-dependent CRP, one can further investigate function approximation or use a more heuristic approach. Strasser et al. put forward another approach in which they created a subset of the graph on which to query. Their results are promising and can likely be applied to (many-to-many) time-dependent CRP [21]. Another angle for future work is to look into adhering to more legislation rules, e.g. including mandatory rests in the route planning. Finally, when working with a different variant of time-dependency, perhaps a variation of our symbolic representation could be used.

References

[1] G. Laporte, “The vehicle routing problem: An overview of exact and approximate algorithms”, *European journal of operational research*, vol. 59, no. 3, pp. 345–358, 1992.

[2] S. Knopp, P. Sanders, D. Schultes, F. Schulz, and D. Wagner, “Computing many-to-many shortest paths using highway hierarchies”, in *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pp. 36–45, SIAM, 2007.

[3] E. W. Dijkstra, “A note on two problems in connexion with graphs”, *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[4] R. Geisberger, P. Sanders, D. Schultes, and C. Vetter, “Exact routing in large road networks using contraction hierarchies”, *Transportation Science*, vol. 46, no. 3, pp. 388–404, 2012.

[5] J. Dibbelt, B. Strasser, and D. Wagner, “Customizable contraction hierarchies”, in *International Symposium on Experimental Algorithms*, pp. 271–282, Springer, 2014.

[6] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck, “Customizable route planning in road networks”, *Transportation Science*, 2015.

[7] R. Geisberger and P. Sanders, “Engineering time-dependent many-to-many shortest paths computation”, in *OASIcs-OpenAccess Series in Informatics*, vol. 14, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

[8] G. V. Batz, R. Geisberger, P. Sanders, and C. Vetter, “Minimum time-dependent travel times with contraction hierarchies”, *Journal of Experimental Algorithmics (JEA)*, vol. 18, pp. 1–4, 2013.

[9] B. C. Dean, “Shortest paths in fifo time-dependent networks: Theory and algorithms”, *Rapport technique, Massachusetts Institute of Technology*, 2004.

[10] H. Bast, D. Delling, A. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck, “Route planning in transportation networks”, in *Algorithm Engineering*, pp. 19–80, Springer, 2016.

[11] 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*, vol. 4, no. 2, pp. 100–107, 1968.

[12] A. V. Goldberg and C. Harrelson, “Computing the shortest path: A search meets graph theory”, in *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pp. 156–165, Society for Industrial and Applied Mathematics, 2005.

[13] M. Hilger, E. Köhler, R. H. Möhring, and H. Schilling, “Fast point-to-point shortest path computations with arc-flags”, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, vol. 74, pp. 41–72, 2009.

[14] D. Delling, A. V. Goldberg, A. Nowatzyk, and R. F. Werneck, “Phast: Hardware-accelerated shortest path trees”, *Journal of Parallel and Distributed Computing*, vol. 73, no. 7, pp. 940–952, 2013.

[15] F. Schulz, D. Wagner, and K. Weihe, “Dijkstra’s algorithm on-line: an empirical case study from public railroad transport”, *Journal of Experimental Algorithmics (JEA)*, vol. 5, p. 12, 2000.

[16] M. Holzer, F. Schulz, and D. Wagner, “Engineering multilevel overlay graphs for shortest-path queries”, *Journal of Experimental Algorithmics (JEA)*, vol. 13, p. 5, 2009.

[17] S. Jung and S. Pramanik, “An efficient path computation model for hierarchically structured topographical road maps”, *IEEE Transactions on Knowledge and Data Engineering*, vol. 14, no. 5, pp. 1029–1046, 2002.

[18] M. Baum, J. Dibbelt, T. Pajor, and D. Wagner, “Dynamic time-dependent route planning in road networks with user preferences”, in *International Symposium on Experimental Algorithms*, pp. 33–49, Springer, 2016.

[19] D. Delling, M. Kobitzsch, and R. F. Werneck, “Customizing driving directions with gpus”, in *European Conference on Parallel Processing*, pp. 728–739, Springer, 2014.

[20] S. Funke and S. Storandt, “Personalized route planning in road networks”, in *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, p. 45, ACM, 2015.

[21] B. Strasser, “Intriguingly simple and efficient time-dependent routing in road networks”, *arXiv preprint arXiv:1606.06636*, 2016.

[22] R. Geisberger, M. Kobitzsch, and P. Sanders, “Route planning with flexible objective functions”, in *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pp. 124–137, Society for Industrial and Applied Mathematics, 2010.

[23] P. Sanders and D. Schultes, “Engineering highway hierarchies”, in *European Symposium on Algorithms*, pp. 804–816, Springer, 2006.

[24] D. Schultes and P. Sanders, “Dynamic highway-node routing”, in *International Workshop on Experimental and Efficient Algorithms*, pp. 66–79, Springer, 2007.

[25] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck, “A hub-based labeling algorithm for shortest paths in road networks”, in *International Symposium on Experimental Algorithms*, pp. 230–241, Springer, 2011.

[26] D. Delling, A. V. Goldberg, and R. F. Werneck, “Hub label compression”, in *International Symposium on Experimental Algorithms*, pp. 18–29, Springer, 2013.

[27] J. Arz, D. Luxen, and P. Sanders, “Transit node routing reconsidered”, in *International Symposium on Experimental Algorithms*, pp. 55–66, Springer, 2013.

[28] H. Bast, S. Funke, and D. Matijevic, “Ultrafast shortest-path queries via transit nodes.”, in *The Shortest Path Problem*, pp. 175–192, 2006.

[29] R. Bauer and D. Delling, “Sharc: Fast and robust unidirectional routing”, *Journal of Experimental Algorithmics (JEA)*, vol. 14, p. 4, 2009.

[30] R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, and D. Wagner, “Combining hierarchical and goal-directed speed-up techniques for dijkstra’s algorithm”, *Journal of Experimental Algorithmics (JEA)*, vol. 15, pp. 2–3, 2010.

[31] P. Sanders and C. Schulz, “Think Locally, Act Globally: Highly Balanced Graph Partitioning”, in *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA’13)*, vol. 7933 of *LNCS*, pp. 164–175, Springer, 2013.

[32] L. Foschini, J. Hershberger, and S. Suri, “On the complexity of time-dependent shortest paths”, *Algorithmica*, vol. 68, no. 4, pp. 1075–1097, 2014.

[33] H. Imai and M. Iri, “An optimal algorithm for approximating a piecewise linear function”, *Journal of information processing*, vol. 9, no. 3, pp. 159–162, 1986.