Bachelor's Thesis

Towards Mobile Time-Dependent Route Planning

Harris Kaufmann

Date of submission: 15.03.2013

Advisors: G. Veit Batz
Jochen Speck
Supervisor: Prof. Dr. Peter Sanders

Institute of Theoretical Informatics, Algorithmics II
Department of Informatics
Karlsruhe Institute of Technology

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Ort, den Datum

Deutsche Zusammenfassung

Zeitabhängige Routenplanung erlaubt es Stoßzeiten und ähnliche periodisch wiederkehrende Effekte zu berücksichtigen. *Time-Dependent Contraction Hierarchies* sind eine hierarchische Darstellung von Straßennetzen, die es erlaubt optimale Wege effizient und unter Einbeziehung von zeitabhängigen Kantengewichten zu berechnen. Mobile Geräte sind vor allem dadurch eingeschränkt, dass sie nur wenig Hauptspeicher besitzen. Die Daten liegen im Flash Speicher des Gerätes in Form von Blöcken vor, die auch nur als Ganzes geladen werden können. Die Blöcke zu laden macht mit Abstand den größten Teil der Beantwortungszeit von Routenanfragen aus. In dieser Arbeit wird eine Darstellung von Time-Dependent Contraction Hierarchies in den Blöcken eines Flash Speichers vorgestellt, die es erlaubt, auf mobilen Geräten Routen zu beantworten und dabei möglichst wenig Blöcke zu laden.

Um das zu erreichen werden die Daten zunächst in eine Darstellung überführt, die so wenig Platz wie möglich benötigt. Dies beinhaltet auch eine verlustbehaftete Kompression der Reisezeitinformation. Durch diese Ungenauigkeiten kann es passieren, dass eine andere Route als die optimale berechnet wird. Trotzdem kann man mit der hier beschriebenen Methode noch von *fast exakten* Ergebnissen reden, denn die *Verzögerung*, die ein Benutzer gegenüber der optimalen Route erfährt, wenn er die vorgeschlagene Route benutzt ist durchschnittliche kleiner als 0.001%. Im schlimmsten Fall ist die Verzögerung ungefähr 0.1%, was für einen Benutzer letztlich die selbe Qualität wie ein exaktes Ergebnis hat, denn selbst im fließenden Verkehr ist der Einfluss des aktuellen Verkehrsgeschehens auf die Ankunftszeit größer.

Außerdem wird die Lokalität der Daten bezüglich beliebigen Routenanfragen erhöht, so dass ein Block, der in den Hauptspeicher geladen wird, außer den aktuell benötigten Daten auch möglichst solche enthält, die mit großer Wahrscheinlichkeit zu einem späteren Zeitpunkt ebenfalls benötigt werden. Dazu wird der Graph mit einem Algorithmus durchlaufen, der allen Knoten in eine *Reihenfolge* bringt. Genau in dieser Reihenfolge werden die Knoten und deren Kantendaten in die Blöcke geschrieben. Dadurch haben Knoten, die in der Reihenfolge nahe beieinander sind, eine große Chance im gleichen Block abgelegt zu werden. Mit dieser Technik wird erreicht, dass Routenanfragen im deutschen Straßennetz mit einer Time-Dependent Contraction Hierarchy mit durchschnittlich 102 Blockzugriffen beantwortet werden können. Unter der Annahme, dass jeder Blockzugriff 1,3 ms benötigt, ergibt sich eine Antwortzeit von 133 ms, was als Reaktion fast ohne Verzögerung empfunden wird. Für alltägliche Routen, die oft kürzer als eine Durchschnittliche Route im Gesamtnetzwerk sind, kann das Ergebnis noch schneller berechnet werden.

Abstract

Road networks with time-dependent edge weights can describe rush hours and similar regular changes in traffic over a day. Optimal routes can be calculated efficiently using time-dependent Contraction Hierarchies. On mobile devices it is usually not possible and also not reasonable to hold the entire data in the limited main memory. Therefore the main performance bottleneck is the data access on the flash memory, which can only be read blockwise. For fast route query calculations, the number of accessed blocks has to be low. To achieve this, we first reduce the amount of data using a lossy compression. This introduces inaccuracies in the calculations hence not always the optimal route is chosen. Nevertheless using this method the result is still nearly exact because the average introduced delay compared to the optimal route is less than 0.001%. In the worst case a delay of about 0.1% occurs which still is too small to be noticed. Second, the data is rearranged in a way that increases the locality. This leads to a lower number of required block loads. For a road network of Germany we can answer random route queries with an average of 102 block loads. Assuming that a block access takes 1.3 ms, this can be done in 133 ms. Users perceive this as a nearly instant reaction. Everyday route calculations tend to be even smaller and can be calculated faster.

Acknowledgements

I first and foremost would like to thank my advisors for helping me out with good ideas and improving my writing by a large extent.

Also I want to thank my family for helping me with english phrases and for the bold effort to understand the topic.

Last but not least a big thanks goes to the coffee bean. Nothing would have been possible without it.

Contents

1 Introduction

In recent years there has been a big rise in the popularity of smartphones and similar mobile devices. These devices are perfectly suited for route planning since they usually feature a reasonably fast processor, enough storage and a GPS receiver. Some of the existing route planning implementations take a server based approach to route planning, where the calculations are done on a server and then transferred to the device. This approach can calculate exact routes quickly using various algorithms, but requires a reliable data connection.

Route calculations done directly on the mobile device have to deal with limited main memory and slow access to the flash memory. In contrast to a server it is not reasonable and usually not even possible to hold all data in the main memory. Therefore the data is loaded from the flash memory when it is needed. This is by far the most time consuming part of a route calculation. On the flash memory, data is stored in blocks which can only be read as a whole. Therefore, the locality of data in these blocks has to be optimized. Each block that is loaded to the main memory should contain a high percentage of relevant data that is needed at a later point. For this data no other block has to be loaded, which reduces the overall number of block load operations and therefore the route calculation time.

There has been extensive work on the topic of route planning using various techniques (see Delling et al. [4] for an overview). Usually a model is used where roads have constant travel times. Many different approaches have been used to answer route queries in this model quickly. Also an efficient implementation for mobile devices has been developed [16]. In contrast time dependent edges allow travel times that are not constant but change with time. This is a much more realistic model and can take into account rush-hour and similar regular changes. Time-dependent travel time data is represented by a function that yields the travel time for a specific time of the day [15].

This has been efficiently solved by Time-Dependent Contraction Hierarchies (TCHs) [1]. TCHs are based on Contraction Hierarchies (CHs) which are hierarchical representations of road networks intended to speed up route queries by reducing the search space. The optimal routes can be found by only searching upward in the hierarchy beginning with the start and destination node. TCHs have been described for a server based environment. In this thesis we present data structures and algorithms that are the basis of a fast and nearly exact implementation of TCHs on mobile devices.

1.1 Our contributions

On the basis of TCHs we present an efficient method for storing the graph of the road network in a way that minimizes the number of required block load operations and accordingly speeds up the route planning process. On a road network of Germany we achieve an average number of 102 block loads per query. This is estimated to take 133 ms, and consequently will be perceived as nearly instant reaction. Generally 100 ms is the limit below which a user does not notice the calculating time. Also the margin of error is very low. For 99% of queries the algorithm yields the optimal route. For all other queries the maximum delay in total travel time is 0.1%. To put this into perspective, the worst case will induce a 3.6 second delay for every hour of travel. It is safe to say that this cannot be noticed by the user and is dominated by other effects. Therefore we call this method *nearly exact*.

Compressing the road network data. We compress the graph that represents the road network to take up as little space as possible. This ultimately increases the amount of data that can be stored in one block. To achieve this, we first create a representation of nodes, edges and functions that is adapted to the way in which data is accessed and does not contain any redundant information. Afterwards we perform a lossy compression that reduces the size of functions and minimizes the number of bits that are used to store values (Section 3).

Arrangement of the nodes. We developed algorithms that arrange all nodes in the TCH. In this arrangement the data is ordered in a way that accounts for the pattern in which nodes are processed during the query algorithm. Nodes that are likely to be processed in the same query are close in the arrangement (Section 4). The nodes are written into blocks in the order of the arrangement. No guarantees can be made which nodes share a block but nodes that are close in the arrangement will have a greater chance to end up in the same block. These are also nodes that are most likely relevant to the same query (Section 5).

Experimental evaluation. We implemented these techniques and performed experiments to support our claims. We give recommendations for reasonable configurations that perform well on real world road networks (Section 6).

1.2 Related Work

Different approaches to route planning have been developed to improve the Dijkstra's original algorithm [7]. Goal-directed methods prefer edges that shorten the distance to the destination node. Hierarchical methods, in contrast, exploit the hierarchical nature of road networks. For a general overview we refer to Delling et al. [4].

Time-dependent route planning was first described by Dreyfus [8] as a generalization of Dijkstra's algorithm. Introductions to this topic are provided by Orda and Rom [15] and Dean [3]. Demiryurek et al. [6] compared the total travel time of routes that were calculated using time-dependent edge weights to routes calculated using constant edge weights values. An general overview is provided by Delling and Wagner [5].

Contraction Hierarchies are based on highway-node routing [17] and were first described by Geisberger et al. [10]. They were generalized for time-dependent route planning by Batz et al. [1].

For constant edge weight Contraction Hierarchies a representation on mobile devices was evaluated by Vetter et al. [16].

2 Fundamentals

Road networks are described by a directed graph $G = (V, E)$, where nodes represent junctions and weighted edges represent road segments. The weight of an edge specifies the travel time for this segment. For mobile applications we are interested in providing the earliest arrival time (*EA-time*) and the corresponding path, which is the earliest arrival path (*EA-path*) for a specified start and destination node (*one-to-one route query*).

Dijkstra's algorithm [7] can provide that on a graph with constant edge weights. For each node a *label* and a *predecessor* is saved. The label represents the travel time from the start node to the current node and is initialized to ∞ for all nodes except the start node which has the label 0. The predecessor information is empty in the beginning for all nodes. The algorithm always *settles* the node with the smallest label, that has not been settled yet. For this a *priority queue (PQ)* is used where the label is used as *key*. A node is settled by *relaxing* all edges, which means that the labels of the target nodes are updated if they are greater than the sum of the label of the node that is settled and the edge weight. Every time a label is updated the predecessor is set to the node that is settled. A node that is settled contains the EA-time for a one-to-one route query from the start node to it. The EA-path can be deduced by following the predecessor information starting at the end. When the destination node is settled the algorithm can be stopped. The number of nodes that have been settled before termination is called *Dijkstra rank* of this query.

The search space can be improved by a *bidirectional Dijkstra* search. This performs a standard Dijkstra search beginning with the start node (*forward search*) and another one beginning with the destination node (*backward search*). For these searches the next node in each case is settled alternately. When a node is settled in both directions the search can be terminated. In the set of nodes where the label is not ∞ in both directions the minimal travel time can be found as the sum of labels of both directions [13].

2.1 Constant Weight Contraction Hierarchies

A *Contraction Hierarchy (CH)* [10] is an improved representation of the graph G that takes advantage of the hierarchical nature of road networks. Important nodes like highway junctions are part of many fastest routes and are placed higher in the hierarchy than less important nodes. In this graph route queries can be evaluated in a way that the search space extends only in the direction of important nodes. This limits the number of nodes that have to be settled before the EA-path is found.

Definitions. A contraction hierarchy H is created from the graph G. For all nodes a total order $\prec$ is defined, where $u \prec v$ means that v is more important than u. Then nodes are *contracted*, which is a procedure that inserts *shortcut edges* into the graph. We will use the following notation:

$$H = (V, E_H) \ , \quad E_H = E \cup S$$

with S being the set of inserted shortcuts. For H the *upward subgraph* $H_\uparrow$ and *downward subgraph* $H_\downarrow$ are defined

$$\begin{aligned} H_\uparrow &= (V, E_\uparrow) \ , \quad E_\uparrow = \{(u,v) \in E_H | u \prec v\} \ , \\ H_\downarrow &= (V, E_\downarrow) \ , \quad E_\downarrow = \{(u,v) \in E_H | v \prec u\} \ . \end{aligned}$$

Edges in $E_\uparrow$ are called upward edges, while those in $E_\downarrow$ are called downward edges. They have the following properties

$$\begin{aligned} E_\uparrow \cap E_\downarrow &= \emptyset \ , \\ E_\uparrow \cup E_\downarrow &= E \cup S \ . \end{aligned}$$

Please note that $H_\uparrow$ and $H_\downarrow$ are *directed acyclic graphs (DAGs)*. The graph that is created from $H_\downarrow$ by reverting the direction of all edges will be called $H_\downarrow^\top$. Using this yet another graph H_* is defined

$$\begin{aligned} H_* &= H_\uparrow \cup H_\downarrow^\top = (V, E_*) \ , \\ E_* &= E_\uparrow \cup E_\downarrow^\top \ . \end{aligned}$$

H_* also is a DAG where all edges point upward in the hierarchy. In H_* we define the following relationships. The node with the highest order will be called *root node*. It is at the same time the only node that has no outgoing edges. This is a result of the contraction process described below. Nodes that have no incoming edges will be called *leaves*. For an edge $(u,v) \in E_*$ we call v the *upward neighbor* of u and u the *downward neighbor* of v.

Basic Properties of CHs. A Contraction Hierarchies has two specific properties that allow queries to be run efficiently on its graph. The first property is that the EA-times for a one-to-one route query from s to t are equal in G and H. The second property is that all EA-paths can be found on an *up-down-path*, which means

$$\begin{aligned}
&\forall \text{ EA-paths } \langle s \to \ldots \to t\rangle \subseteq G \;: \\
&\exists \text{ an EA-path } \langle s \to \ldots \to x \to \ldots \to t\rangle \subseteq H \;: \\
&\langle s \to \ldots \to x\rangle \subseteq H_\uparrow \wedge \langle x \to \ldots \to t\rangle \subseteq H_\downarrow \;.
\end{aligned}$$

Ordering The Nodes. The first step to create a CH is to use a heuristic to determine the total order of nodes $\prec$. This order has a big influence on the quality of the CH. A good order reduces the number of nodes that have to be settled for a query. As stated by Geisberger et al. [10] finding an optimal order is a difficult problem but a simple heuristic already performs well. In any case the order does not affect the basic properties of the CH.

Contracting The Nodes. Using the total order a CH is created by *contracting* nodes in their order of importance. Usually the node order is not known completely beforehand. This is not a problem because only the next node has to be known so the order can be built at the same time as the hierarchy. Contracting a node v means that for all paths $\langle u \to v \to w\rangle \in E$ which are the unique shortest path between u and w, a shortcut edge (u, w) is inserted. For this shortcut the *middle node* v is stored with the edge. The next contractions are done on the remaining *overlay graph* $G' = (V', E')$ where $V' = V \setminus \{v\}$ and E' the induced edges together with the added shortcuts. In this graph shortest paths are preserved. Please note that shortcuts also can be contracted to a new shortcut.

One-To-One Route Query. The algorithm that solves one-to-one route queries has to find the earliest arrival up-down-path. For this a bidirectional Dijkstra search is used. The forward search beginning with the starting node is performed on $H_\uparrow$ while the backward search beginning with the destination node is performed on $H_\downarrow^\top$. In this way all up-down-paths are searched. Because CHs tend to be flat and sparse, the search space is reduced greatly compared to a Dijkstra search on G (also see Section 6.2).

The result of a query is an EA-Path that may contain both edges of the original graph and shortcuts. Shortcuts can be expanded by reading the node that is stored with the shortcut, finding the two edges associated with these nodes and expanding the edges recursively if they are shortcuts themselves.

Stall-On-Demand. The structure of the Contraction Hierarchies guarantees that the shortest path between two points is always found. During the Dijkstra search in one direction however the edges which are reverse to the search direction are not relaxed. This means that optimal paths in $H_\uparrow$ and $H_\downarrow$ respectively are followed. These are not necessarily also optimal paths in H. Non-optimal path in H that are nevertheless followed trigger unnecessary data access operations.

This can be improved using the *stall-on-demand* technique first demonstrated for highway node routing [17]. Let $d(v)$ be the label of a node v and $c(v, w)$ the edge weight of the edge (v, w). Before a node v is settled for the forward search we first check if there is an edge $(v, w) \in H_\downarrow^\top$ so that $d(w) + c(v, w) < d(v)$. In this case we already know that v is on a non-optimal path and the search can be stopped (*stalled*) for v. Also for nodes in the neighborhood of v the stalling can be propagated if the path via w is shorter than the current label.

2.2 Time-Dependent Contraction Hierarchies

Time-dependent Contraction Hierarchies (TCHs) are demonstrated by Batz et al. [1] and are a solution for the more general and realistic model where edge weights are not represented by constant travel times but *travel time functions (TTFs)* $f : \mathbb{R} \to \mathbb{R}_{\geq 0}$ that specify the travel time of this edge depending on the departure time. Just like in Constant Weight Contraction Hierarchies the creation of TCHs consists of an ordering step followed by contraction of the nodes. Similarly queries are evaluated using a bidirectional Dijkstra-like algorithm. This may seem trivial but using TTFs instead of constant travel times increases the complexity significantly.

Properties of TTFs. TTFs have to fulfill the *FIFO-property* which means that the slope between two arbitrary points of the TTF must not be smaller than -1. This reflects the fact that when using optimal routes on a road network you can not arrive earlier at the destination if you started at a later departure time. In this thesis TTFs are periodic with a period $\mathcal{P}$ of 24h. This is not obligatory, since a time window can also be used.

Also TTFs are here piecewise linear functions. These are represented by a list of points sorted by their x value. For an edge (u, v) with the TTF f as edge weight we write $u \to_f v$. The travel time of this edge at the departure time τ is specified by $f(\tau)$. This is *evaluated* using the binary search algorithm. We find the two surrounding points of τ and use linear interpolation of the y values. Because the TTFs are periodic they can also be evaluated for x values before the first point or after the last point.

In addition we define $\min f$ and $\max f$ as the minimum respectively maximum of the TTF. The *complexity* $|f|$ describes the number of points in the TTF.

TTF Shortcuts. Similar to constant edge weight CHs shortcut edges are added to the graph when contracting a node v. For unique shortest paths $\langle u \to_f v \to_g w \rangle$ a shortcut $u \to_h w$ is inserted. For constant edge weights the travel time of the shortcut is the sum of the travel times of both contracted edges. For TTFs we have to define the *link* operation. When calculating a combined TTF we have to take the travel time of the first edge into account. Thus the linked shortcut edge $u \to_h w$ has the TTF $h(\tau) = g(f(\tau) + \tau) + f(\tau)$. The complexity of the shortcut is assessed as $|h| \in O(|f| + |g|)$.

During contraction a shortcut $u \to_h w$ can be created, while there is another shortcut or original edge $u \to_i w$ in the overlay graph. When using TTFs h can have lower travel times for some departure times while i is better for other departure times. These edges are merged to a *minimum edge* which contains the best parts of each edge. Therefore a set of middle nodes and their corresponding intervals of departure times where it is the best shortcut has to be saved. The complexity of the minimal edge of h and i is in $O(|h| + |i|)$.

It is apparent that a shortcut usually has a greater complexity than its source edges. This shortcut can itself be a source edge for another shortcut. Repeating this process multiple times hence can produce very complex shortcut edges.

Query. Also the query algorithm has to be adapted to these new conditions. TTFs can only be evaluated if a departure time is known. This is the case for the forward search but not for the backward search. Batz et al. [1] presented various algorithms which solve this problem. Here we use the *Inexact TCH* algorithm that is shown in the next section.

2.3 Inexact TCH Algorithm

The *Inexact TCH* algorithm provides the EA-time and EA-path of one-to-one route queries on TCHs. This algorithm also works well if the accuracy of time values is reduced. Three steps are required to evaluate a query. The first two steps create a small search space, which contains the EA-path. In the third step a Dijkstra search that uses TTFs is performed. Because TTFs can not be evaluated backwards this step is not bidirectional.

2.3.1 Step 1: Bidirectional Interval-Dijkstra Seach

The Interval-Dijkstra search uses intervals as labels for the nodes. This is represented by a min and max value that describe the minimum and maximum travel time to the node. As predecessor information a set of predecessors is saved. When relaxing edges these intervals have to be compared against each other to update the value of the target. Let $[q[v], r[v]]$ be the label of a node v where $q[v]$ is the min and $r[v]$ is the max value. When relaxing an edge $u \rightarrow_f v$ we distinguish the following cases. If $r[u] + maxf < q[v]$ we can determine that u provides the better route and the interval of v has to be updated and is set to $[q[u]+minf, r[u]+maxf]$. Also u is set as the only predecessor of v. If $q[u]+minf > r[v]$ nothing has to be updated. In all other cases the intervals are merged. The new interval of v is $[min\{q[u] + minf, q[v]\}, min\{r[u] + maxf, r[v]\}]$. u is then added to the list of predecessors and all existing predecessors remain.

During the Dijkstra search the min value of the interval is used as the key for the PQ. For the Interval-Dijkstra we cannot settle a node after its edges have been relaxed. It can happen, that the node is reinserted in the PQ at a later point. Instead we call this node *scanned*.

The search in this step is performed bidirectionally. The start node is initialized with the interval $[\tau, \tau]$, where τ is the departure time. Then an Interval-Dijstra is performed on $H_\uparrow$. Similarly the backward search is performed on $H_\downarrow^\top$ with the start interval $[0, 0]$. All other nodes are initialized with the label $[\infty, \infty]$. The search cannot be aborted if a common node has been found by both search directions. Instead the node becomes a candidate node, which could be part of the EA path. The sum of the max values of a node in both directions is an upper limit which is used to limit the search space. For each candidate that is found the *upper limit* is adapted. All paths where the min value exceeds the upper limit can be discarded. Also stall-on-demand can be applied to this search.

Using the predecessor information we get a subgraph that contains all paths in one direction that possibly are part of an EA-path. We call these subgraphs *cones*. The combination of the cone of the forward and backward search is called *corridor*. The EA-Path can always be found in the corridor. This is also true if min and max values of TTFs are inaccurate. The only requirement is that min and max values are *conservative*, which means that the min value is smaller than the lowest function value and the max value is bigger than the highest function value. This ensures that the created corridor will always contain the optimal EA path. The disadvantage is that with declining accuracy of min and max values the size of the search space will grow.

2.3.2 Step 2: Thinning The Corridor

The result of the first step are two cones that create a corridor between the start and destination node which always contains the EA-path. Also we obtain a set of candidate nodes. The corridor can now be used as reduced search space on which a Dijkstra search using the TTF data is performed. But we can do better. There are still a lot of nodes that do not belong to an up-down-path that contains a candidate node. A Dijkstra search would follow all the dead ends that were aborted in the first part because they exceeded the upper limit or were stalled.

We can remove these by doing a Breadth-first search (BFS) using only the edges to predecessors in the cone of the forward search starting with the candidate nodes. This will traverse the search space backwards and only include edges that are on a path between the start node and a candidate node. This step is not necessary for the cone of the backward search because a forward search on nodes of this cone has to start with the candidate nodes. This searches in opposite direction of the backward Interval-Dijkstra and therefore only includes edges that are on a path between the candidate nodes and the destination node.

2.3.3 Step 3: TTF Dijkstra

The thinned corridor is a very limited search space, that we will use for a forward Dijkstra search using the TTF data (*TTF-Dijkstra*), which was first described by Dreyfus [8]. This is essentially the same algorithm as the simple Dijkstra search except that the label of the start node is initialized with the departure time. Then every time an edge is relaxed the label of the current node is used as the time for which the TTF is evaluated. This step is separated in two parts. The first part will search on the thinned cone of the forward Interval-Dijkstra until the queue is empty (*Upward TTF-Dijkstra*). These are again the starting points of a TTF-Dijkstra search on the cone of the backward Interval-Dijkstra (*Downward TTF-Dijkstra*).

Using these three steps to evaluate routes also has a beneficial impact for mobile devices. During the Interval-Dijkstra no TTF data has to be accessed as the min and max values can be stored with the regular edge data. Only in this last step TTFs are evaluated. This runs on the severely reduced search corridor and therefore saves data access operations. Also no node data has to be accessed during the last step. All node information can be held in the main memory in a space-saving way using hash lists.

The final result of this step is the EA-Path some of whose edges may be shortcut edges. These have to be expanded to represent road edges. Doing this efficiently on mobile devices is not part of this thesis and may be researched in the future.

3 Compression of the TCH

In this section we present methods used to compress the data of the TCH. This means that more data can be fitted in a block which in the end leads to less required block load operations. While some of these techniques involve a loss of accuracy, we are still able to provide nearly exact results for route queries. As shown by experiments in Section 6.5 a big reduction of data is possible while introducing a maximum error that is too small to be noticed.

3.1 Representation of the TCH

Using the TCH H_* (see Section 2.1) a representation is created that can be written to the flash memory. In this graph all edges point upwards in the hierarchy, but edges of $E_\uparrow$ are marked as upward edges while edges of $E_\downarrow^\top$ are marked as downward edges. For each node all outgoing edges of $E_\uparrow \cup E_\downarrow^\top$ are stored together. This makes sense considering the fact that every time a node is scanned during the *Interval-Dijkstra* all outgoing edges are accessed. During the forward search all upward edges are relaxed while downward edges are checked for their time to stall non-optimal paths. For the backward search this works vice versa.

The simplest representation of a graph is an adjacency array, which consists of two arrays. The first array contains an entry for each node, which contains the ID of the first edge adjacent to this node. The index in this array is the identifier for the node. Edges are stored in the second array, where every edge has an entry. All edges adjacent to a node are stored continuously so the edges of a node n can be found by reading the edges between the first edge of node n and the first edge of node $n + 1$. This approach does not suit our purpose here, because finding a node and its edges requires accesses to both arrays. This might trigger two block loads which is the quantity we want to reduce. Instead we use an adapted graph representation that has a better access pattern.

In our *Compressed TCH* representation a node is identified by the bit-specific address of the node data. We avoid storing a table of nodes altogether. This makes iterative access to the nodes impossible which is not needed to evaluate route queries. Instead we only need to find the target node of an edge which is identified by its address. For this TCH the only node data is information about outgoing edges. Therefore the identifier of a node is the address of the first bit of its first edge.

The bit-representation of edges has variable size after they are compressed, which makes array-like access impossible. Instead all edges have an additional field that indicates if the following edge belongs to the same node. With this information the edges of a node can only be accessed iteratively. This is sufficient because as stated earlier all of its edges are looked at when a node is scanned.

The point data of TTFs accounts for a substantial part of the total graph data (see Section 6.2.2), but TTFs are only accessed during the *Upward-* and *Downward-TTF-Dijkstra*, where the search space is limited by a corridor. To avoid loading point data of TTFs during the *Interval-Dijkstra*, we store it separately from the node and edge data. The representation of the edge only contains the upper and lower bounds as well as an identifier of the TTF, which again is the bit-specific address of the point data.

When implementing this graph representation we need to ensure that node identifiers are unique. This is the case as long as the edge data takes up a non-zero amount of bits. Then the next node starts at a different address and therefore has a different identifier. However the root node has no outgoing edges in E_* and therefore the following node will be placed at the same address. A node that is represented without data is called *empty node*. Duplicated identifiers can be avoided by giving empty nodes identifiers that are above the address of the last node. The limit above which node identifiers are not treated as addresses but as identifier of an empty node has to be stored with the data. Usually the root node will be the only empty node. This is not a correct CH because these nodes can only be reached from nodes which are lower in the hierarchy. Nevertheless allowing multiple empty nodes allows a much more fail-safe implementation. During our tests we encountered an incorrect but still usable TCH that was created by clipping a small part from an TCH. With the exception of some nodes on the border this worked fine.

3.2 Compressed Data Types

In this section we present the basic data types that are used to encode the edge and TTF data in the compressed TCH. For these data types the *bit-width*, which is the number of bits used, is adjustable and should be chosen as low as possible. For time values a reduction of the bit-width leads to a loss of accuracy. In this case we have to balance these two factors against each other, which is examined in Section 6.5.

Normalization. To get the best possible accuracy numbers should be *normalized.* This means determining an upper and lower bound (u and l) for the number v. v is normalized to the interval $[l, u]$ by calculating v', where

$$v' = \frac{v-l}{u-l} \ .$$

v' always is in the interval $[0, 1]$.

N-Bit Integer. Integers with a bit-width of n can represent 2^n different values. For example a 1-bit integer is used to store boolean values.

Addresses. Addresses are used as identifiers of nodes and TTFs. Because the data is stored disregarding the byte boundaries a node or TTF can begin and end at any bit position. This means that addresses have to be specific down to the bit, which will extend the needed bits by 3 additional bits. To determine the optimal bit-width we calculate the highest address A that will occur. Then the optimal bit-width n is

$$n = \lceil \log_2(A) \rceil .$$

Please note that the data is stored in blocks. On current hardware their size is always a power of 2. Therefore the address can be interpreted as two parts. The least significant $o+3$ bits of an address indicate the offset within the block, where 2^o is the block size in bytes. The remaining $b = n - (o + 3)$ bits contain the number of the block that has to be loaded to make this data accessible.

Fixed Point Numbers. Fixed point numbers represent values in the interval [0, 1]. No matter where the value is on the interval, the absolute accuracy remains always constant.

To find the representation of a fixed point number t using n bits we transform it to a n-bit integer value x. It can be rounded to the next higher, the next lower or the nearest possible value. Depending on the rounding mode these integers x are stored:

$$\begin{aligned} x &= \lceil t \cdot (2^n - 1) \rceil \ , \\ x &= \lfloor t \cdot (2^n - 1) \rfloor \ , \\ x &= \lfloor t \cdot (2^n - 1) + 1/2 \rfloor \ . \end{aligned}$$

While rounding to the nearest value the maximal absolute error will be $\frac{1}{2} \cdot \frac{1}{2^n - 1}$.

Floating Point Numbers. Floating point numbers have the characteristic that smaller values will have a better absolute accuracy than larger values. They provide a way to keep the relative error in a certain range. The numbers that can be stored also have to be in the interval [0, 1].

The number of available bits n is divided into a number of bits for the mantissa m and the exponent e where $n = m + e$. The exponent is always negative and does not contain a sign bit. It specifies the range of the interval that is encoded. An exponent c specifies the range $[2^{-c}, 2^{-(c-1)}]$. The only exception is $c = 0$ where the specified range is $[0, 2^{-(2^e - 1)}]$. To transform a value t to its floating point representation, we first determine the exponent c that specifies the range that contains t. Then we calculate t' which is t normalized to this range. In the end we store t' as m-bit fixed point number and c as e-bit integer. A rounding mode has to be specified when storing the mantissa as fixed point number.

While the relative error is dependent on m for values where the exponent c is not 0 no statement can be made about values where c is 0. Depending on the smallest non-zero value and e the relative error can be up to 100%. This prevents easy estimates about the relative error.

3.3 Bit-Representation of Edges

In this section we present a bit representation that is used to store an edge $e : u \to_f v$. The following information is essential to the bidirectional *Interval-Dijkstra* search and therefore has to be part of the compressed edge data:

- An indication whether the next edge also belongs to u
- The *direction*, which indicates whether $e \in E_\uparrow$ or $e \in E_\downarrow^\top$
- The address of v
- $\min f$ and $\max f$
- The address of f

This information has to be present in each edge but not always has to be stated explicitly. If possible we try to leave out information and save the associated bits. Other than full edges these types of edges can occur:

Constant Edge. Not all roads have high and low traffic times that influence the travel times. If the TTF f is a constant function we can replace $\min f$, $\max f$ and f itself by a single constant travel time value.

Bidirectional Edge. Constant edges tend to have the same travel time in both directions. All Edges that occur in $E_\uparrow$ and $E_\downarrow^\top$ identically do not have to be stored twice. For this we expand the direction field by a flag for bidirectional edges

Internal edges: An edge is called *internal* if it points to a node that resides in the same block. Instead of storing the whole address as an identifier it is sufficient to store only the offset inside the block. This can save a considerable amount of bits.

The bit-representation that is used to represent an edge has the following layout:

n	$c = 0$	i	*dir*	*ptarget*	*min*	*max*	*pfunction*

n	$c = 1$	i	*dir*	*ptarget*	*val*

[1bit, bool]	n	Indicates whether there is an edge immediately behind the current edge that belongs to the same node.
[1bit, bool]	c	Indicates whether the edge has a constant or variable travel time. Constant edges do not need a TTF.
[1bit, bool]	i	Indicates whether the target node begins in the same blocks as the current edge. Internal edges need less bits for *ptarget*.
[2bit]	*dir*	Direction of the edge. Can be upward, downward or bidirectional.
[address]	*ptarget*	Address of the target node.
[floating point]	*val*	Travel time of constant edges.
[floating point]	*min, max*	Minimum and maximum values of the TTF.
[address]	*pfunction*	Address of the associated TTF.

For *ptarget* and *pfunction* the optimal size is automatically calculated using the highest occurring address. This is not trivial for *ptarget* and is described in Section 5.2 at length. The bit-widths of these address fields can be further optimized when we take into account that edge and node data is separated from TTF data. For nodes we only need to determine the address relative to the first node block, while TTF addresses are relative to the first TTF block.

During the creation of the Compressed TCH the maximum travel time T that exists is determined. The travel times *val*, *min* and *max* are then normalized to the interval $[0, T]$. We chose floating point numbers to represent these values. This is motivated by the fact that travel time values need to have a low relative error. For long routes and edges a larger absolute error is not noticeable but short edges need to be evaluated very precisely.

While *val* should be rounded to the nearest possible value it is important to round *min* and *max conservatively*. This means that *min* is rounded down and therefore always smaller or equal to the lowest TTF value. In the same way *max* is rounded up. This ensures that the optimal EA-Path is always part of the the corridor created by the *Interval-Dijkstra*. Inaccurate *min* and *max* are not directly related to the accuracy of the result, but will increase the size of the interval used in the search and yield a larger search space. This will be discussed in Section 6.5.

3.4 Compression of TTFs

We are using TTFs that are given as piece-wise linear functions. This means they are represented by a list of points (x, y) where the $x \in [0, \mathcal{P}]$ is the time of the day while y is the travel time. These TTFs are compressed in two steps both of which include a loss of accuracy. The first step is a reduction of the number of points using the Imai-Iri algorithm (see below). After that we translate the TTF to a bit-representation where the data types have a reduced bit-width.

Please note that this step has to take place before the edges are composed. During the compression an error can be introduced to the minimum or maximum of the TTF. In order to keep conservative *min* and *max* values of the edges, these have to be calculated afterwards.

3.4.1 Function Approximation

The Imai-Iri algorithm [12] is an effective way to reduce the amount of data. This is done by reducing the number of points. Here we use the implementation of Neubauer [14], which takes a *relative tolerance* ϵ and the TTF as an input. Then a new TTF with the minimal number of points is calculated where the relative error is at most ϵ compared to the original TTF. The new TTF is called *Approximated TTF*.

3.4.2 Bit-Representation of TTFs

TTFs are represented by the number of points and a list of points. Therefore we use the following simple bit structure.

npoints	x_1	y_1	x_2	y_2	...

[int]	*npoints*	Number of points in the TTF.
[fixed point]	x_i	The x position of the i-th point.
[fixed point]	y_i	The y value of the i-th point.

The optimal bit-width of *npoints* can easily be determined. Let *maxpoints* be the maximal occurring number of points in any approximated TTF. The minimal number of bits that is required can be calculated as $\lceil \log_2(maxpoints) \rceil$.

The x values are normalized to the interval $[0, \mathcal{P}]$ and then stored as fixed point number. Therefore all times of the day can be described equally accurate. For best accuracy x is rounded to the nearest possible value and there is a chance that multiple points receive the same fixed point x value. They will be merged into one point with the average of all y values.

The y value is normalized to the conservative *min* and *max* value of the edge and then stored as a fixed point number rounded to the nearest possible value. This is a good way to reduce the amount of data, because for all TTFs that are evaluated the corresponding edge has been visited during the *Interval-Dijkstra*. Therefore *min* and *max* values of the edge are already known and we only have to specify the development inside these bounds. To be consistent, it is crucial that the compression steps are executed in the correct order. At first we create an approximated TTF using the Imai-Iri algorithm. Then the minimum and maximum of the approximated TTF is determined and conservatively converted to fixed point numbers. In the end these inaccurate but conservative *min* and *max* values are used to normalize all y. The normalization introduces an indirect dependency of the accuracy of queries on the accuracy of the *min* and *max* values. But for reasonably chosen bit widths of *min* and *max* this effect is negligible (see experiments in Section 6.5).

4 Node Arrangement

In the last section we discussed techniques to reduce the space needed to store edges and TTFs. This allows more edges and more TTFs to be fitted into one block. In this section we present a method that increases the locality of the data. Each block should contain as much data as possible that is relevant to the current query.

To achieve this we create an *arrangement* of all nodes where those nodes which are most likely relevant to the same queries are close to each other. Then blocks are filled with the edge data of nodes in the order of their appearance in the arrangement. Nodes that are close to each other therefore have a high chance to end up in the same block. TTFs are written using the same arrangement. For each node in the arrangement all edges will be looked at and if they contain a non-constant TTF it will be written to the current function-block. This is reasonable because they are accessed in a similar pattern and the *TTF-Dijkstra* search profits from the same techniques as the *Interval-Dijkstra* search (see Section 6.4).

Using this method, no guarantees can be made which nodes will be in the same block. Even small changes to the bit width of values or other parameters will influence the positioning of data. This will cause limited fluctuations of the average number of block loads. The reason for this are nodes that are almost always relevant to the same queries may be separated by block boundaries which adds an extra block load. This is especially significant for important nodes that are part of many queries. Nonetheless the presented method is a good way to reduce the average number of needed block load operations.

4.1 Objectives for a Good Arrangement

Finding a good arrangement of nodes is not trivial. In a hypothetical optimal arrangement all nodes that are scanned before and after a node v during a Dijkstra search are also close to v in the arrangement. While this is somewhat predictable at the beginning of a Dijkstra search, there are a huge number of possible search spaces and successions of scanned nodes at a later moment. All possible predecessors and successors in a Dijkstra search would have to be close to v in the arrangement and beyond that for each of these nodes there are again additional nodes that have to be close. Obviously we cannot satisfy all of these demands at the same time.

Here we present objectives that are elements of a node arranging algorithm. These objectives cannot always be achieved at the same time but have to be balanced against each other. The aim is to reduce the average number of block load operations. Short routes have a very small search space and will always be calculated fast. On the other hand for very long routes a longer calculating time might be acceptable for the user. But the biggest impact on the user experience comes from the average number of block loads.

4.1.1 Objective 1: Preserve Structural Distances

This is the most obvious objective to increase the locality of the data. During a Dijkstra search edges to upward neighbors are relaxed. At a later point these upward neighbors are scanned themselves and their edges are relaxed. If these nodes are close in the arrangement we increase the chance that no additional block load is required for the upward neighbors. Depending on the direction of the search, edges of $E_\uparrow$ or $E_\downarrow^\top$ are relaxed. Therefore all upward neighbors in H_* have to be close.

The outcome of this is that we have to preserve the structural distances of the TCH in the arrangement. This cannot be applied perfectly, because we map a two dimensional graph onto a one dimensional arrangement. For a node v we want all nodes that have incoming edges $(u, v) \in H_*$ and all nodes that are the destination of outgoing edges $(v, w) \in H_*$ to be close in the arrangement. All nodes that have been determined to be close to v have incoming and outgoing edges themselves from or to nodes that again need to be close to their neighbor nodes. In the end we get all nodes of the connected component that need to be close to each other in the arrangement.

4.1.2 Objective 2: Preserve Temporal Distances

The Dijkstra algorithm always scans the active node with the shortest current travel time from the starting node. Hence the search space will increase faster in directions where edges have a small weight. A good arrangement utilizes this factor by putting nodes connected by short edges closer to each other than nodes connected by long edges. Of course since the travel time of edges is described by TTFs there exists no definite edge weight. Analogous to the *Interval-Dijkstra* (see Section 2.3) we use the minimum of the TTF to compare the weight of edges against each other.

4.1.3 Objective 3: Avoid Fragmentation of Important Paths

In the arrangement every node can only be placed once. This means that if we notice during the creation of the arrangement that a node u needs to be close to a node v that is already in the arrangement we are forced to separate these two nodes. These nodes almost inevitably end up in different blocks. Queries that follow paths that include u and v most likely need an additional block load. This is called *fragmentation* of a path. It can happen multiple times and in the worst case each node is placed in an individual block.

Fragmentation will occur in paths whose nodes are inserted into the arrangement at a later point. Therefore the most important paths that have the biggest impact on the average number of block load operations are added to the arrangement first. There are two aspects that constitute an important path. On one hand paths that are part of the upward search of the biggest number of start nodes influence the block loads for the biggest number of queries. On the other hand long paths have the potential to get fragmented exceptionally badly. Because the TCH is a DAG, there is a certain degree of correlation between these two aspects.

4.2 Definitions

Minimal Depth. The depth of a node v is defined as the length of a path from v to the root node r in H_*. There usually are multiple paths between those nodes. We define the minimal depth as follows:

$$P = \{\langle v \to \ldots \to r\rangle \in H_*\},$$
$$mindepth = \min(|p|) \ , \quad p \in P.$$

The minimal depth for all nodes is calculated by initializing each node with ∞. Then a BFS on $H_*^\top$ is started at the root node which has the minimal depth 0. Upon visiting a node v coming from the node u the minimal depth is updated.

$$\text{mindepth}(v) = \min(\text{mindepth}(v), \text{mindepth}(u) + 1).$$

Mass. The mass of a node v is defined as the number of nodes that are in the subgraph with v as its root. This subgraph contains all nodes u for which there exists a path $\langle u \to \ldots \to v\rangle$ in H_*. Calculating the mass of all nodes has a running time worse than $O(n)$. To do this in a passable time for large TCHs the calculation can be parallelized.

Layer. Layers are a way of separating the graph into horizontal slices with similar properties. The layer number of a node v is the maximal length of a path from any leaf node to v in H_*. The layers are built iteratively where all leaf nodes form the first layer. These nodes are then removed from the graph. The next layer contains all nodes that are now leafs in the reduced graph. This process is repeated until the last layer only contains the root node.

4.3 Arranging The Nodes

In this section we present algorithms that have proven to create a good arrangement of nodes. The common principle is to traverse the graph using different algorithms starting with the root node or the leaf nodes (see Section 2.1). The nodes are then added to the arrangement in order of their traversal. If we traverse the TCH in downward direction, edges of $E_*^\top$ are used to traverse the graph. Otherwise E_* is used. No distinction is made between upward and downward edges.

The algorithms used are modifications of a standard depth-first-search (DFS). These are adapted to comply with the objectives presented in the previous section. When traversing with a DFS like algorithm the order in which edges are visited has a considerable impact on the quality of the arrangement. Nodes that are reachable through the first processed edge will be substantially closer to the starting node than nodes reachable through other edges. We call the order in which edges are visited *edge preference.*

Similar to Section 3.1 we also adapted this for TCHs with a few incorrect nodes. These nodes do not have an upward neighbor and might not be found while traversing the graph. To avoid this side issue all nodes are checked at the end and if there are nodes left that have not been visited they are added at the end of the arrangement.

4.3.1 Depth-First-Search (DFS)

For this method a standard recursive DFS is executed on $H_*^\top$ starting with the root node. The nodes are added to the arrangement when leaving a node. In our experience using a low mass as edge preference yields the best results. This means edges that point to nodes with the lowest mass are traversed first.

As shown in Section 6.4 the DFS creates a good arrangement for nodes in lower layers. For nodes in higher layers the arrangement performs significantly worse than others that were created using a different algorithm, which overall makes this the worst choice of all presented algorithms.

4.3.2 Inverse Depth-First-Search (IDFS)

This algorithm traverses H_* with a DFS beginning with a leave. Contrary to the DFS traversion method nodes will be added to the arrangement when entering the node. This DFS run will find most nodes of the graph and traverse in a cone up in the hierarchy. There are however nodes on lower layers can not be reached by the DFS. To include these nodes in the arrangement the process is repeated with other leaves as a starting point until all nodes are visited.

The order in which leaves serve as starting point of the DFS runs is essential for the quality of the arrangement. In our experience a good method is to start with the leaf with the greatest minimal depth. After that the leaves are processed in order of decreasing minimal depth. The reason for this is probably that the longest paths are traversed in the first DFS run. Therefore they cannot get fragmented by later runs.

Equal to the DFS traversion method in 4.3.1, the best edge preference we found is to traverse edges that point to nodes with the lowest mass first. This is surprising, because the two algorithms traverse the TCH in different directions.

This yields a very bad arrangement for nodes on lower layers, while nodes on higher layers are arranged well, which is the exact opposite to the DFS traversion method (see Section 6.4).

4.3.3 Combined Depth-First-Search (CDFS)

A simple idea for improvement is to combine the strengths of both of the previous algorithms. As stated the DFS creates a good arrangement for nodes lower layers while the IDFS is preferred for nodes in higher layers. We first determine the border b in such a way that nodes in a layer greater or equal to b perform better when arranged using the IDFS compared to the DFS algorithm. Accordingly nodes in a layer smaller than b perform better when arranged using the DFS algorithm.

Algorithm 1 illustrates how the arrangement is created. First nodes on higher layers are arranged using the IDFS, then the remaining nodes are arranged using the DFS algorithm. It is important that the IDFS still starts at the leaf nodes. The graph is traversed in the familiar fashion but only nodes that are part of a layer above or equal b get added to the arrangement. Similarly the DFS has to start with the root node of the graph and traverse the whole graph but only nodes on a layer below b are written out. If this is not respected the quality of the arrangement is significantly worse in our experience.

4.3.4 Both-Way Depth-First-Search (BDFS)

This method (see algorithm 2) performs a DFS search that follows all edges on $E_* \cup E_*^\top = E_\uparrow \cup E_\uparrow^\top \cup E_\downarrow \cup E_\downarrow^\top$. The root node serves as start node. Nodes are always added to the

Algorithm 1. Create arrangement of a TCH using the CDFS algorithm. A border layer b is passed to the function and all nodes in layers above or equal b are arranged using the IDFS algorithm. All other nodes are arranged using the DFS algorithm.

```
function $traverseCDFS(b : \text{Layer}) : \text{Arrangement}$
    $r : Node$   // root node
    $L : Nodes$   // leaf nodes
    $A : Arrangement$   // node sequence
    $layer[n] : Layer$   // returns the layer of a node $n$
    $visited[u] := false$ for all $u \in V$   // visited nodes
    procedure $recursionIDFS(n : Node)$
        if $layer[n] \geq b$ then
            $A.add(n)$
        $visited[n] := true$
        $UN := \{p \in V | (v, p) \in E_*\}$ ordered by increasing mass of $p$
        forall the $w : Node$ in UN do
            if not $visited[w]$ then
                $recursionIDFS(w)$
    procedure $recursionDFS(n : Node)$
        $DN := \{p \in V | (n, p) \in E_*^\top\}$ ordered by increasing mass of $p$
        forall the $w : Node$ in DN do
            if not $visited[w]$ then
                $recursionDFS(w)$
        if $layer[n] < b$ then
            $A.add(n)$
        $visited[n] := true$
    sort $L$ by decreasing minimal depth
    forall the $l : Node$ in $L$ do
        $recursionIDFS(l)$
    $visited[u] := false$ for all $u \in V$   // reset visited flag
    $recursionDFS(r)$
    return $A$
```

arrangement when entering a node. For each node first edges in H_* are traversed in order of increasing minimum of the TTF. Then edges in $H_*^\top$ are traversed in order of decreasing mass of the target.

Our experiments show that this algorithm creates a better arrangement than the others. We assume the reason for this is that it has properties which increase the probability that all presented objectives (see Section 4.1) are met.

Structural and temporal distances are carried over to the arrangement when traversing edges of H_*. This step finds all reachable nodes above the current node in the hierarchy and adds them to the arrangement. This tends to be a manageable amount of nodes because because usually only one edge of $H_*^\top$ is followed before we again traverse edges of H_*. These nodes are structurally close in the TCH and thanks to this step as close as possible in the arrangement. Temporal distances are taken into account via the edge preference.

When traversing $H_*^\top$, edges that have a target with the greatest mass are followed first. Therefore paths that can be reached by the biggest number of nodes are the first to be inserted in the arrangement. For these paths the chance of fragmentation is minimized.

Algorithm 2. Create arrangement of a TCH using the BDFS algorithm. It is based on a standard DFS, but follows edges in $E_* \cup E_*^\top$. Also edge preferences are taken into account.

```
function $traverseBDFS()$ : Arrangement
    $r : Node$   // root node
    $A : Arrangement$   // node sequence
    $added[u] := false$ for all $u \in V$   // added nodes
    procedure $recursion(v : Node)$
        $A.add(v)$
        $added[v] := true$
        $UN := \{m \in V | (v, m) \in E_*\}$ ordered by increasing min value of edge $(v, m)$
        forall the $w : Node$ in UN do
            if not $added[w]$ then
                $recursion(w)$
        $DN := \{n \in V | (v, n) \in E_*^\top\}$ ordered by decreasing mass of $n$
        forall the $w : Node$ in DN do
            if not $added[w]$ then
                $recursion(w)$
    $recursion(r)$
    return $A$
```

5 Filling The Blocks

When an arrangement has been established, the nodes and TTFs can be written to blocks using the bit-representation presented in the Sections 3.3 and 3.4.2. While this might seem to be straightforward there are a few things to consider.

5.1 Continuous and Non-Continuous Storage

Blocks can be filled using *continuous storage* of data, where nodes respectively TTFs are written next to each other and the address of the next one is the bit after the last bit of the previous. In this case some nodes or TTFs will inevitably be distributed into two blocks. The result is that an access to this node or TTF will always trigger two block load operations.

This can be avoided by checking the size of what needs to be written, and if it is too big for the current block the data is moved to the next block. However, thereby some bits remain unused. This is called *non-continuous storage*. Nonetheless a fallback mechanism has to be implemented. A node or TTF might exceed the size of a block and therefore has to be written across block boundaries.

If a node is distributed into two blocks clear rules have to be established which block nodes and edges are attributed to. In our implementation this is the block that contains the first bit of the node and edge respectively. A node that is attributed to a block b therefore can contain some edges attributed to b and some attributed to $b+1$. Consistently, in this case an edge is internal iff the edge and its target are attributed to the same block.

5.2 Circular Dependencies

The variable size of the bit-representation of edges causes *circular dependencies*. These are non-trivial dependencies where a parameter that decides the bit-size of an edge is influenced by the position of edges, which again depends on the size of previous edges. For the bit-representation of edges two circular dependencies occur. Section 5.2.1 presents an algorithm to resolve these without introducing edges of non-optimal size.

Internal Edges In order to decide that an edge is internal we need to find out whether the target node is in the same block. This is a problem if the edge belongs to a node that is before the target node in the arrangement. In this case the position of the target node depends on the size of preceding edges, which again partially depend on the position of nodes that are not yet known.

Size of Address Pointers The number of bits that are used to describe an address pointer should be as low as possible. To achieve this we find out the highest address that needs to be stored and determine the number of needed bits (see Section 3.2). For TTFs this is an easy thing to do, because the size of TTFs and therefore the highest address is known beforehand.

For nodes once again a circular dependency comes up. The highest address that can occur depends on the size of the edges, which in turn depends (via the size of the target pointer) on the highest address. This cannot be calculated in advance because if the size of the addresses changes different edges are internal, which also has an influence on the highest address.

5.2.1 Solving the Circular Dependencies

We solve this circular dependency in two steps. At the end we know for each edge if it is internal and the addresses of all nodes. Also the optimal size of target-node pointers is determined during this process. Using this information we can write the compressed TCH representation to the flash memory.

Step 1: Assign Nodes to Blocks In this step we determine for all nodes in which block they will be placed (means: the first bit of the first edge, see Section 3.1). For this we simulate writing nodes and edges to the blocks but only memorize the number of occupied bits p of the current block. If one or more edges that have already been written in the simulation turn out to be internal we can reduce p by the amount of saved bits. This takes advantage of the fact that the size of every edge is reduced by *ired* bits if it is internal. This is the same amount for edges with TTFs and edges with constant travel time.

$$ired = ptarget - \log_2(blocksize)$$

Algorithm 3 shows how this is done using continuous storage. In addition to the bit-counter p there is *numberEdgesTo* which counts how many times a node n has been target of an edge in the current block. We process the nodes in order of their appearance on the arrangement. Every new non-empty node is assigned to the current block. Then *numberEdgesTo* is checked for the current node. If it is nonzero this means that there have been edges in the current block that point to this new node. These edges can be reduced to internal edges. Accordingly the position pointer is reduced by the size that is saved.

For every edge of the node we increase *numberEdgesTo* of the edge target by 1. Then we increase p by the size of the edge. For this we also check if the target node has been assigned to the current block. If this is the case we already know that the edge is internal. If p exceeds the block size a new block is added, *numberEdgesTo* is cleared and p is reduced by the block size.

Please note that there is a case where this algorithm does not yield the optimal result when using continuous storage. If a new block has been created a node can still be the target of many edges in the old block. If all of these were internal sometimes the node could still be placed in the old block. But we consider this case to be rare and without a significant influence on the results.

There is however still the circular dependency of the address pointer. This can only be solved using trial and error. We find out the bit-size s of the smallest possible edge. Usually this is an internal edge with constant travel time. We now guess the number of bits of the target pointer as

$$ptarget = \lceil \log_2(s \cdot |E_*|) \rceil .$$

The node assignment step is started using this bit-width. If the sum of occupied bits exceeds $2^{ptarget} - 1$, *ptarget* is increased by 1 and the node assignment is restarted.

Step 2: Finding Node Addresses Before we can write the nodes to the flash memory we have to determine the addresses of all nodes first. This can now easily be done because we know the size of all edges in advance. If the target node has been assigned to the current block the edge is internal. Calculating the addresses is necessary because the address serves as an identifier of the node. To write the target pointer of an edge we need to know the address of the target node even if it has not been written yet.

6 Experimental Evaluation

To evaluate the performance, we built a software framework which implements the Inexact TCH Algorithm (Section 2.3) and all techniques presented in this thesis to compress (Section 3) and arrange (Section 4) the TCH and place it into blocks (Section 5). Conditions of mobile devices are simulated by implementing block-wise access and a block cache.

Algorithm 3. Assign nodes to blocks using continuous storage. The algorithm takes the arrangement A as input and stores for every block all nodes that have been assigned to it.

```
procedure assignNodes()
    $p := 0$   // number of occupied bits in the current block
    if currentBlock.contains(e.target) then
        n
    umberEdgesTo[$u$] := 0 for all $u \in V$
    forall the $n$ : Node in $A$ do
        if $n.edges = \emptyset$ then
            emptyNodes.add($n$)
        else
            currentBlock.add($n$)
            $p$ := $p$ - numberEdgesTo[n] · ired
            forall the $e$ : Edge in $n$ do
                numberEdgesTo[e.target]++
                $p$ := $p$ + e.fullSize
                  // fullSize is the size of the edge with a full target address
                if currentBlock.contains(e.target) then
                    $p$ := $p$ - ired
                if $p \geq blocksize$ then
                    currentBlock := new Block
                    $p$ := $p$ - blocksize
                    numberEdgesTo[$u$] := 0 for all $u \in V$
```

All tests were executed on road networks of Germany and Western Europe which enable estimates about the scaling of the techniques in a real world environment. On these road networks we ran sets of randomly generated one-to-one route queries. The focus here is to evaluate the number of needed block load operations as well as the accuracy of EA-paths calculated with compressed data compared to the actual EA-path. Also a reference configuration is proposed that turned out to be a good trade-off between good accuracy and little average block load operations on these road networks.

6.1 Basics

6.1.1 Road Networks

Two road networks of Germany and Western Europe were used to run the experiments. The data was provided by PTV AG for scientific use. Edge weights and TTFs on the road network of Germany were collected from historical data and reflect the traffic between Tuesday and Thursday. For Western Europe the edge data is generated synthetically to represent a high amount of traffic. Using node ordering and contraction techniques described by Batz et al. [1] these road networks were converted to TCHs.

6.1.2 Queries

All tests are done using groups of queries that describe the search of an EA-Path between a source and destination node starting at a specific time of the day. A query is represented by a set (s, d, t_d, t_a) where s and d are the start and destination node, t_d is the time of departure and t_a the time of arrival. In the query sets s, d are picked uniformly at randomly out of V for the nodes and t_d out of the interval [0h, 24h] for the departure time. The arrival time t_a however is calculated on the original graph. It reflects the earliest arrival time using uncompressed edge weights and will be used as reference value for accuracy testing of the compressed TCH. Unless stated otherwise a group of 10,000 randomly selected queries is used. This is sufficient to get solid average values for accuracy and performance.

Queries are evaluated using the Inexact TCH algorithm (see Section 2.3). As a simplification we did not propagate the stall when using stall-on-demand.

6.1.3 Blocks

For mobile devices the main bottleneck is the access to the flash memory. Hence we will use a idealized model that represents a generic mobile device with flash memory. It is divided into blocks of size 2^k bytes, which only can be read as a whole. The access time is always constant no matter where the block resides. Also there is no advantage of reading blocks in a sequential order compared to random access. In this model we ignore the calculating time that is negligible compared to the time needed for flash memory access.

The size of blocks can vary depending on the technology used in the chips of the flash memory. We estimate 4 kiB blocks to be a plausible value that will be used for the experiments. With improving technology future block sizes will most likely be larger following the trend of the past years. The performance of different block sizes will also be evaluated in Section 6.7.

To estimate the performance on a real device we will use values that were determined for the Nokia N800 Internet Tablet by Vetter [16]. For random access and 4 kiB block sizes an access time of 1.3 ms was measured. During the evaluation of experiments in this section we assume

ϵ	min, max		constant		x-value	y-value
	m	e	m	e		
0.1%	8	4	8	4	11	7

Table 1: Standard configuration defined by the relative tolerance of the function approximation (ϵ) and the number of bits used for all values. Floating point values are split into a bit-width for the mantissa and for the exponent. The arrangement used for the standard configuration is BDFS.

that the total time necessary to answer a query is calculated by multiplying the number of blocks loaded with the access time of 1.3 ms.

Block Cache A block cache is implemented that holds loaded blocks in the main memory in case they are accessed at a later point of the query. If the cache is full, a replacement strategy controls which block is replaced. Unless stated otherwise a cache is assumed that is large enough so no block has to be replaced. Before each query is evaluated the cache is cleared.

Block Access Per Phase The Inexact TCH algorithm is divided in three phases (see Section 2.3). The bidirectional Interval-Dijkstra, the Upward-TTF-Dijkstra and the Downward-TTF-Dijkstra. For each phase block loads that are triggered during this phase are counted separately. Please note that Upward and Downward-TTF-Dijkstra search can share blocks. A block that was loaded during the Upward-Dijkstra phase may also contain data for the Downward-Dijkstra phase. This does not trigger a block load during the Downward-TTF-Dijkstra phase and no additional block load is attributed to this phase.

Block Access Per Layer If a node is accessed for which the block is not in the main memory the block load is accounted to the layer of the node. The basic assumption for this is that nodes on the same layer share similar properties, with respect to an arrangement. For different arrangements it is evaluated how many block loads are triggered by nodes of a specific layer. Of course, once a block is loaded into the main memory it can contain nodes of any layer which, when accessed, do not trigger a block load that is attributed to its layer.

6.1.4 Configuration Parameters

A *configuration* describes a set of parameters that control the representation of the TCH in the flash memory. This consists of the *compression parameters* which contain the relative tolerance (ϵ) of the function approximation and bit-widths for *min* and *max*, *constant*, x and y values. Also the arrangement algorithm, which can be one of DFS, IDFS, CDFS or BDFS, is part of the configuration.

The standard configuration has the compression parameters shown in Table 1 and uses the BDFS as an arrangement. It has proven to perform well in all our measurements. In Section 6.1.4 we show basic measurements of the standard configuration. Section 6.4 and Section 6.5 subsequently justify why we think that this is a good configuration.

6.1.5 Accuracy

Because of the lossy compression the *calculated arrival time* for the *calculated best path* on the compressed TCH will deviate from the EA-Time. Independent of this the calculated best path

	Germany	Europe
# Nodes	4.7 M	18.0 M
# Edges	10.8 M	42.2 M
% Time-Dependent Edges	22.1 %	23.0 %
Number of Layers	110	155

Table 2: Basic properties of the TCHs

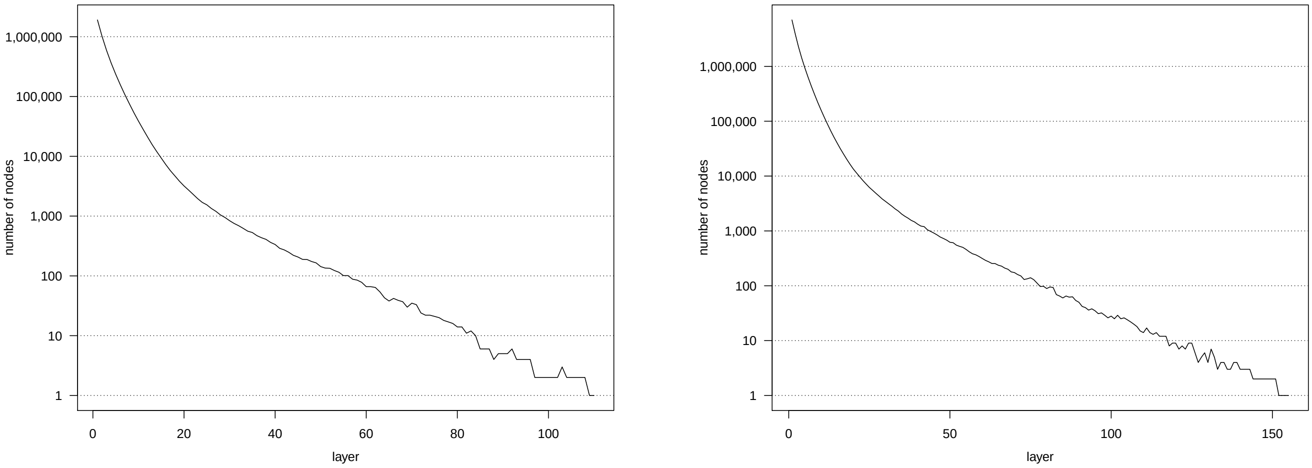

Figure 1: The number of nodes per layer on the German (left) and European (right) TCH.

can be different from the EA path on the uncompressed TCH. The relative delay an user will experience because of a non-optimal path is called *real delay*. To find the real delay the calculated best path on the compressed TCH is recorded and recalculated with the uncompressed edge data. Using the result and the EA-time stored with the query (t_a) we can calculate the relative real delay. A query that results in a calculated best path where the real delay is 0 is called *perfect query*.

Please note that the calculated arrival time can contain errors even if the optimal route is found. In contrast to the real delay the calculated arrival time may also be earlier than the optimal arrival time.

6.2 Road Networks

Table 2 shows basic properties of the TCHs. The number of edges consequently includes shortcut edges. On the other hand bidirectional edges (see Section 3.3) are counted only as one edge. The number of layers only shows a small increase although the number of nodes and edges of the European graph is several times the number of nodes of the German graph. This illustrates the hierarchical nature of TCHs that limits the search space of a query.

6.2.1 Graph-Structure by Layer

Layers (see Section 4.2) are a good way to compare different levels of the hierarchy. The TCH is separated in horizontal slices that are expected to have similar properties. Figure 1 shows the number of nodes on each layer. Lower layers contain the majority of the nodes. In fact the first two layers already contain more than 60% of the nodes. Figure 2 shows the distribution of edges with a non-constant TTF. While most of the edges have a constant travel time they are almost exclusively distributed on the lowest few layers. This fact is not surprising because edges

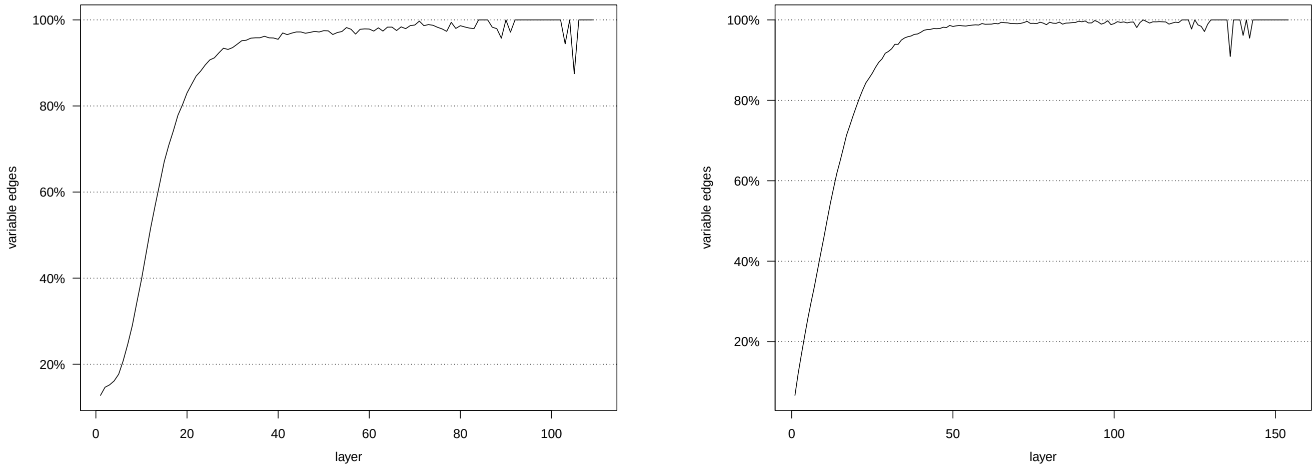

Figure 2: The percentage of time-dependent edges per layer on the German (left) and European (right) TCH.

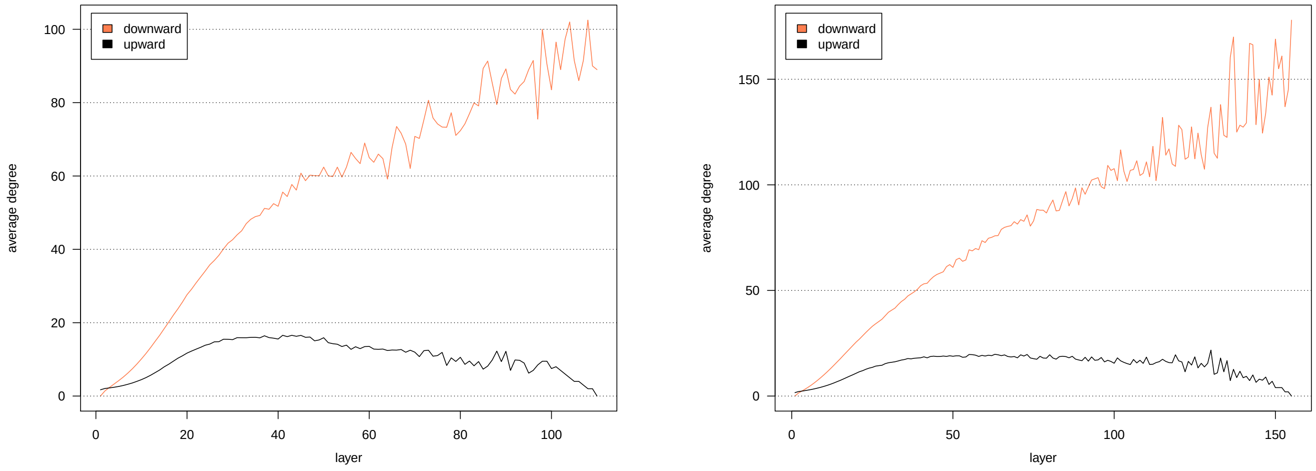

Figure 3: Average outgoing (upward) and incoming (downward) degree of nodes in H_* by layer on the German (left) and European (right) TCH.

	Germany	Europe
node blocks	14,990	61,670
TTF blocks	30,185	87,119

Table 3: The number of node and TTF blocks using the standard configuration.

	ϵ							
	0%		0.01%		0.1%		1%	
	avg	max	avg	max	avg	max	avg	max
Germany	102.6	11,350	34.8	336	22.5	117	14.4	58
Europe	54.2	8,952	24.6	194	15.9	72	11.2	29

Table 4: Number of points of TTFs using different relative tolerance values ϵ in the function approximation. Constant value edges were not taken into account when calculating the average.

on higher layers tend to be shortcuts that have been contracted multiple times (see Section 2.2) and therefore have a high probability of being time-dependent. The average degree of incoming and outgoing edges for different layers are evaluated in Figure 3. The number of incoming edges gets rather large at the top of the graph. This allows queries to reach the top of the TCH very quickly.

Based on these results we can distinguish at least two parts of the TCHs. The lowest few layers contain the majority of nodes whose edges are predominantly not time-dependent. Also the degree of incoming and outgoing edges is comparatively low.

6.2.2 TTFs

The TTFs are responsible for a substantial part of the data (see Table 3). In this section we examine the number of points that form a TTF and how it is influenced by function approximation with the Imai-Iri algorithm. For this purpose we take a look at 4 configurations that will later be used in accuracy testing. Apart from the uncompressed TCH, approximations with a maximal relative tolerance ϵ of 0.01%, 0.1% and 1% are used to minimize the number of points (see Section 2.2) in TTFs. For each case the average and maximal number is determined.

Table 4 shows the number of points in TTFs. The average and maximum number of points of approximated TCHs (see Section 3.4.1) with different relative tolerance values ϵ are compared to the original data (0%). As clearly visible there are TTFs in the original graph that have a huge number of points. These are shortcut edges that are the result of multiple contraction operation (see Section 2.2). In the worst case the number of points of a shortcut is the sum of the numbers of points of the edges from which it was created. These outliers prevent the uncompressed TTF data from being used as a reasonable basis for a compressed TCH that is used on mobile devices. Even the introduction of a very small tolerance like 0.01% solves this problem. Not only the average complexity of TTFs is reduced severely but also the TTFs with the biggest number of points are compressed to a fraction of their original size. Therefore this step is essential when creating a compressed representation.

It is noteworthy to observe that TTFs in the German TCH contain more points on an average as well as in the maximum than the European TCH. This remains the same when compressed, which is not expected because the German road network is much smaller. An explanation could be that the data for both graphs was created using different methods. For the TTFs created from real world historic data in the German TCH there is probably more variability that is

	total # of block loads		% per phase			real delay		% perfect
	avg	max	bidir	up	down	avg	max	queries
Germany	102.28	263	69.5	17.5	12.9	$1.1 \cdot 10^{-4}$	0.104	98.9
Europe	372.94	1818	34.6	41.8	23.6	$5.8 \cdot 10^{-4}$	0.081	94.0

Table 5: Measurements of the standard configuration. The average and maximum number of total block loads is show as well as the percentage that is triggered in each phase. Also the average and maximum real delay and the percentage of perfect queries is shown.

harder to compress than the artificially created TTFs of the European TCH.

More data per TTF leads to less TTFs per block, which increases the average number of block loads. This could lead to the conclusion that TTF access performs worse for the German TCH than for the European TCH. This is, however, not the case. We assume that there is a different effect with a greater influence on the number of block loads. As described by [1] (Table I.), the TTFs in the European original graph have a larger range between the minimum and maximum values. This creates a larger search space which means that during the last phase of the query algorithm considerably more TTFs have to be evaluated and therefore more data has to be loaded.

6.3 Standard Configuration

Measurements of the number of block loads and the real delay of the standard configuration (see Section 6.1.4) are shown in Table 5. On average it allows a random query to run using 102.28 block loads on the German TCH and using 372.94 block loads on the European TCH. Assuming an access time of 1.3 ms per block load an average query on the German TCH needs about 133 ms while an average query on the European TCH needs about 485 ms. A significant difference between the TCHs is that on average only 30.4% of the block loads are during the TTF-Dijkstra when using the German TCH. For the European TCH it is 65.4%. This is probably caused by the different nature of the TTF data described in Section 6.2.2.

Route queries on the German compressed TCH find the perfect route in 98.9 % while on the European compressed TCH 94 % are evaluated perfectly. This, however, does not make a significant difference because the average real delay as well as the maximum real delay is in the same order of magnitude.

6.3.1 Dijkstra rank

Random route queries are a good indicator of the quality of the data compression and storage but does not represent the ordinary use of route planning. Most real route queries are a lot shorter than the average random query and therefore can be computed faster. For long routes most users are willing to accept a longer route calculation time. The route distance is not a good indicator to test this. Long distances in sparsely populated areas can be calculated easily while even short distances in cities are a lot more complex to calculate. To solve this problem we used the Dijkstra rank as route complexity indicator. The Dijkstra rank is the number of settled nodes needed to reach the destination using a (unidirectional) TTF-Dijkstra search algorithm.

The second group of queries is ordered by their Dijkstra Rank. This describes the number of settled nodes of a TTF-Dijkstra search between the start and destination node. There are 100 random queries with a Dijkstra rank of 2^n where n is between 5 and 22 for the German TCH and between 5 and 24 for the European TCH. Using the queries ordered by rank statements can be made about the performance depending solely on the complexity of the query. This corresponds to the fact that countryside routes of big distances can be calculated easily because few roads with little connections exists. The same complexity is met for very short distances in a city (see Section 2.3).

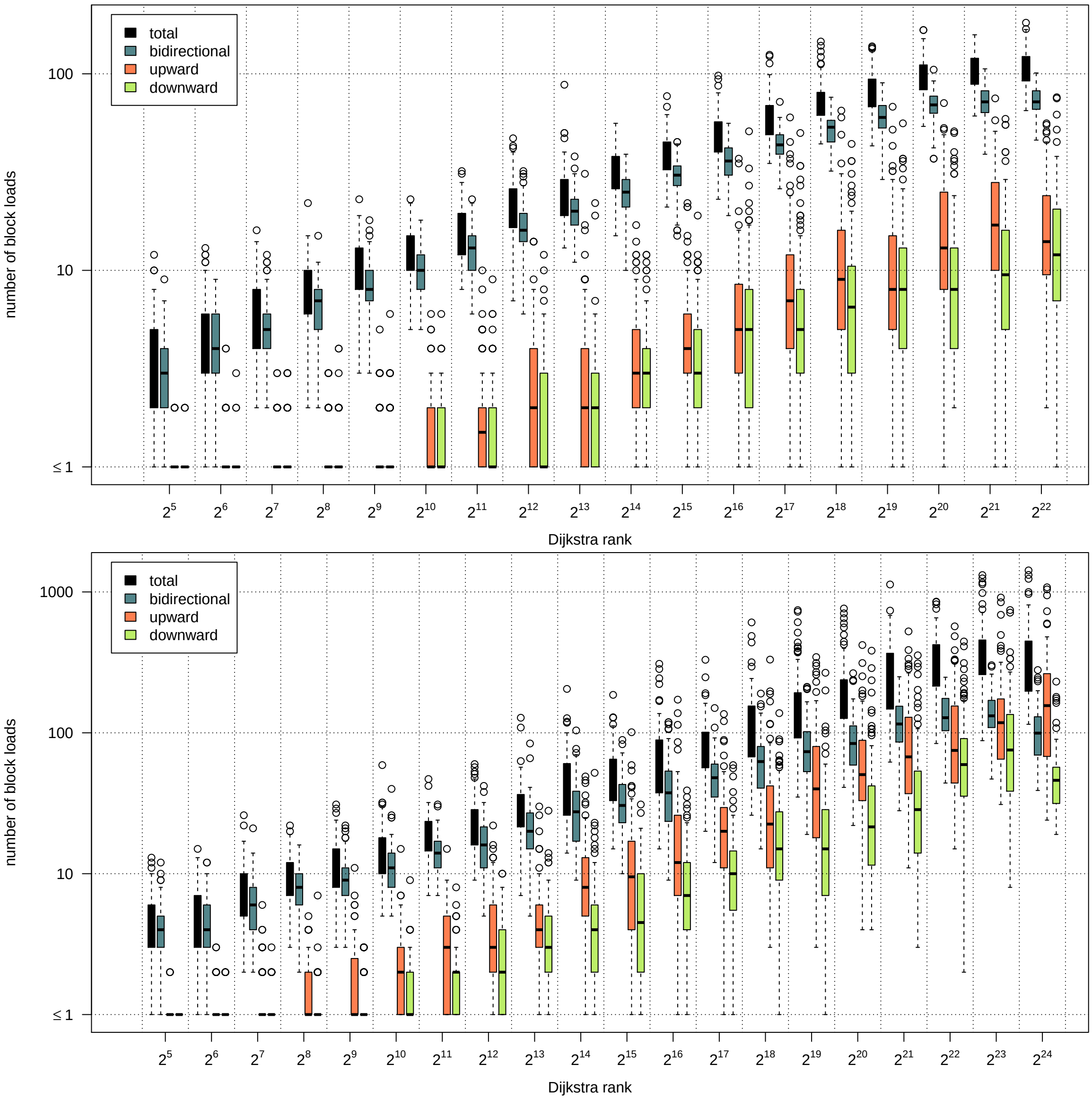

Figure 4: Performance of queries grouped by their Dijkstra rank.

Figure 4 shows the number of block loads over the Dijkstra rank. Also the phases of the query algorithm are plotted separately. From the results we can conclude that the algorithm scales as expected with increasing Dijkstra rank. However the TTF access on the European graph rises to huge amounts with high Dijkstra ranks. Also even for lower Dijkstra ranks there are outliers that require a huge amount of block loads. As already mentioned in Section 6.3 this has probably to do with the nature of the original TTF data. Nontheless this is not satisfying and could be subject of future work.

	Germany	Europe
non continuous and internal edges	102.3	372.9
continuous writing	102.2	372.8
no internal edges	104.2	377.3

Table 6: Different ways to write to blocks

6.3.2 Filling Configuration

Table 6 shows different methods to store edges in blocks. We try to place complete nodes and TTFs in blocks. If nodes and TTFs are written continuously the total amount of data is reduced while some nodes and TTFs overlap the border between two blocks. This data needs two block loads when accessed. The results show no clear advantage of non-continuous over continuous writing. Maybe the advantages (whole nodes in blocks) annihilate with the disadvantages (increased space need).

Not using internal edges increases the size of nodes and therefore the number of block loads. As the results show, internal edges reduce the number of block loads as expected, but only to a small extent. This is an interesting result, because internal edges make the implementation significantly more complex.

6.4 Performance of Different Node Arrangement

In Section 4.3 we describe different algorithms (DFS, IDFS, CDFS, BDFS) to create node arrangements. Randomly shuffled node arrangement is used as a reference to visualize the magnitude of improvement of the algorithms. To compare these we use the average number of block load operations of 10,000 random queries. A low number of average block load operations means that the loaded blocks contain a high amount of needed data. This has been defined as the goal of this thesis.

All arrangements use an algorithm to traverse the graph as well as an edge preference which decides which edge is followed first, starting with the current node. In case two edges have the same edge preference value there is no recommendation. To ensure that the order of the creation process of the TCH does not influence the results in these cases, all edges are randomly shuffled before ordering them by edge preference. Similarly the starting nodes of the IDFS are randomized before determining the order in which they serve as starting point.

The CDFS is constructed as combination of a DFS and an IDFS. All nodes below or equal to a specific layer are ordered using the DFS algorithm while those above are ordered using the IDFS algorithm. In the following tests layer number 14 is used as the border between the two techniques. The reason is that layers below or equal to the border trigger more block loads when arranged with IDFS compared to DFS. For higher layers this works vice versa. Surprisingly the border is identical for the European and German TCH. This can be seen in Figure 5.

Table 7 compares the performance of the node arrangements for the German and European TCH. We measured the average number of loaded blocks as well as the worst case with the maximum number of total block loads encountered. A more differentiated analysis is possible by looking at the number of block loads that are triggered during each phase of the query algorithm. During the bidirectional phase only node data is loaded. We measured the average number of block loads and the average number of nodes that are used per loaded block. During the upward and downward phase all relevant node data already resides in the main memory. The average number of block loads therefore describes block loads of TTF data. Also the average number of required TTFs per loaded block is examined.

	total		bidirectional		upward		downward	
	# loads		# loads	nodes	# loads	TTFs	# loads	TTFs
	avg	max	avg	per block	avg	per block	avg	per block
Germany								
random	622.76	1,235	571.45	1.02	29.78	2.01	21.53	1.91
DFS	130.62	316	97.07	6.01	19.28	3.11	14.27	2.88
IDFS	114.02	284	81.77	7.13	18.52	3.24	13.73	3.00
CDFS (14)	110.74	276	78.48	7.43	18.56	3.24	13.71	3.00
BDFS	**102.28**	**263**	**71.13**	**8.20**	**17.93**	**3.35**	**13.22**	**3.11**
Europe								
random	1,695.56	6,128	1,160.90	1.01	343.05	4.65	191.61	4.88
DFS	431.42	1,969	168.21	6.96	169.26	9.43	93.96	9.95
IDFS	395.63	1,868	144.31	8.12	161.00	9.91	90.32	10.36
CDFS (14)	391.83	1,887	140.32	8.35	161.17	9.90	90.34	10.35
BDFS	**372.94**	**1,818**	**128.96**	**9.08**	**155.81**	**10.24**	**88.17**	**10.61**

Table 7: Performance of different node arrangements during different phases of the query algorithm. The compression parameters of the standard configuration are used, therefore the results of the BDFS arrangement describe the standard configuration.

	Germany	Europe
DFS	75.7 %	77.3 %
IDFS	35.2 %	34.9 %
CDFS (14)	74.3 %	76.3 %
BDFS	70.6 %	65.8 %

Table 8: Percentage of internal edges in the whole graph for different arrangement algorithms.

Please note that for the random arrangement the average number of TTFs loaded per block is not close to 1. The reason is that in writing an arrangement to blocks all TTFs of a node are stored consecutively. When a node is settled during the TTF-Dijkstra search all TTFs in the corridor are evaluated. Therefore even with a random arrangement the number is considerably larger than 1.

To examine the performance of the node arrangements on different levels of the hierarchy we measure the average number of block loads per layer and query (see Section 6.1.3). Figure 5 shows the results. All node arrangements show a similar pattern. While nodes on the bottom layers have a small degree and the top layers contain very few nodes the middle section of the graph is the most difficult part (see Section 6.2.1). Most block loads are triggered on these layers, which can be explained by the large number of well connected nodes.

It can clearly be seen that for both graphs the IDFS is by far the worst arrangement for layers up to layer 14. For higher layers the DFS performes bad. CDFS therefore combines the best of both algorithms. The problem with this approach is that every time the search crosses the border between lower part that was arranged by the DFS and the upper part that was arranged by the IDFS, an extra block load is triggered. This can be seen by the spike in the line of the CDFS.

Table 8 shows the percentage of all edges that are internal (Section 3.3) depending on the arrangement. The low percentage of the IDFS catches the eye. We assume that this is related to the bad performance of the IDFS on lower layers. These lower layers also contain the vast majority of nodes, which explains the big difference.

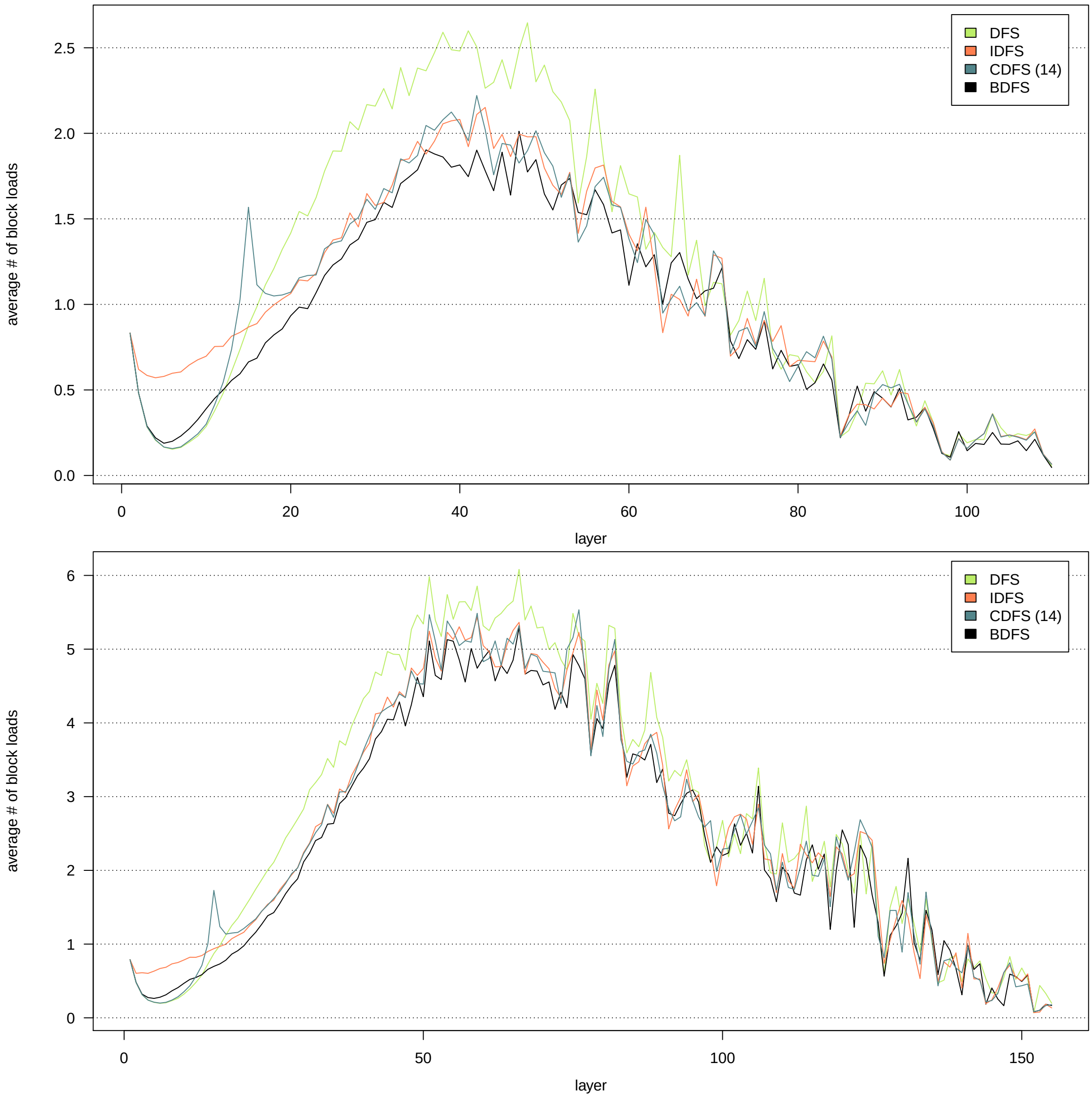

Figure 5: Average number of block loads that are caused by node data not being in the main memory faced with the layer of the node.

	Germany			Europe		
ϵ	avg [%]	max [%]	perfect [%]	avg [%]	max [%]	perfect [%]
0.01%	< 0.001	0.004	99.9	< 0.001	0.004	99.8
0.1%	< 0.001	0.056	99.0	< 0.001	0.076	98.0
standard configuration	**< 0.001**	**0.104**	**98.9**	**< 0.001**	**0.081**	**94.0**
1%	0.005	0.713	92.4	0.015	1.149	81.4

Table 9: Real delay of Imai-Iri compression without bitcompression compared to the standard configuration case with bitcompression.

With these measurements we draw the following conclusion: the BDFS always delivers the lowest number of block load operations and undercuts all other arrangement algorithms nearly on all layers. Out of the presented node arrangements it is the best one without any compromise.

6.5 Accuracy of Inexact Queries

Our goal is to provide nearly exact results for route queries. This is only achieved if the real delay is small enough not to be noticeable for the user. As a simple rule we defined that the maximum real delay should be around 0.1%. With a delay this small, other road users and environmental effects will be dominating effects. Also this only describes the real delay in the worst case. On average the delay will be smaller by magnitudes. The standard configuration (see Section 6.1.4) satisfies this requirement. The goal of this section is to justify the choice of this configuration.

The difficulty with determining good configuration parameters is that all parameters independently influence the accuracy and number of block loads. Theoretical thoughts are more complex than expected in this case. While an approximated TTF is bounded by the relative tolerance ϵ the same cannot be said about a path that contains multiple time dependent edges. The edges are linked non-linearly (see Section 2.2) and any inaccuracy also influences the time for which the following edge is evaluated. For a more detailed examination refer to Geisberger and Sanders [9]. Also very small time values can have a large relative error using the floating point format.

Instead our approach to determine the configuration parameters is to take an educated guess and then to vary the parameters one at a time to find the local optimum. We chose the relative tolerance of the Imai-Iri algorithm comparatively large to profit from the reduced number of points. We estimated that the reduction of points outweighs a reduction in bit-width that introduces the same inaccuracy in terms of data reduction. After that we chose the bit-widths in the compression parameters as small as possible so that the maximum real delay remained around 0.1% .

In the following we subsequently examine the influence of changes to different parameters of the configuration. The standard configuration is used as reference and values are changed one at a time.

6.5.1 TTF Approximation

Table 9 shows the behavior of the average and maximum real delay as well as the percentage of perfect routes calculated using the 10,000 random queries. For $\epsilon \in \{0.01\%, 0.1\%, 1\%\}$ we determine the real delay without compression of the bit-widths. Also for comparison purposes the results of the standard-configuration (0.1% with bitcompression) are shown.

In all cases the average real delay was very small, certainly not noticeable in real world traffic. Meanwhile single bad routes can make a bad impression on the user of the route planning program. As previously stated we want to achieve a maximal real delay of 0.1% which makes an ϵ value of 0.1% a suitable candidate.

6.5.2 Bit-Widths

The results of our measurements are shown in figures 6 (for Germany) and 7 (for Europe). The percentage of perfect queries, the average number of block loads and the maximum real delay are plotted over the total bit-width. For numbers in the floating point format the total bit-width has to be split between the mantissa and the exponent. As an example for a total of 12 bit there are 11 possibilities to split the bits so that $m+e = 12$. Some of these configurations lead to good results while most of them do not. To solve the problem we calculate all results in a reasonable range and choose the result with the highest percentage of perfect queries. The bit-width distribution that of the standard configuration is always favored over a configuration with the same number of bits and a slightly higher percentage of perfect queries. The standard configuration is marked by a vertical line in the graphs.

The interpretation of the results is mainly based on the results of the European TCH. For the German TCH the same trends are detected but the changes mostly happen in a very small range.

min and max values The *min* and *max* value of edges are used during the bidirectional Interval-Dijkstra to find a corridor which is then used as search space for the TTF-Dijkstra algorithm (see 2.3). There is only an indirect influence on the accuracy of the result. *Min* and *max* values with a large rounding error cause y values of TTFs to be normalized to an unnescessary big interval. That leads to slightly more inaccurate results.

As is obvious from the measurements there is no relevant influence of the accuracy of min and max values on the accuracy of route queries. At the same time there is an easy way to determine the best bit-width. Very inaccurate values lead to a bigger min to max range because of the conservative rounding. The search space that is calculated in the first phase of the query algorithm gets unnecessarily large, which leads to an increase in block loads. Increasing the number of bits leads to a smaller search space but at the same time increases the data per edge that has to be stored. Between these extremes an optimal bit-width can be found for which the number of block loads is minimal.

constant edge values Constant edges are described by a single floating point travel time value. This leads to very predictable characteristics. For constant values with smaller bit-width the results of queries are more inaccurate while the number of block loads is smaller. For increasing bit-width the accuracy as well as the number of block loads increases.

As expected, the accuracy of queries increases with the bit-width used for the constant values. It has to be noted, however, that there is no real increase in the average number of block loads. This can be explained by the fact that constant edges, though they account for the majority of all edges, are almost exclusively in the lowest layers. In these layers the upward degree is low so the search can climb up fast in the graph. Therefore constant edges have no relevant influence on the average number of block loads.

x values of TTFs The x values of TTF points are fixed point values that describe the time of the day for which the travel time is specified. A low bit-width here results in a low resolution

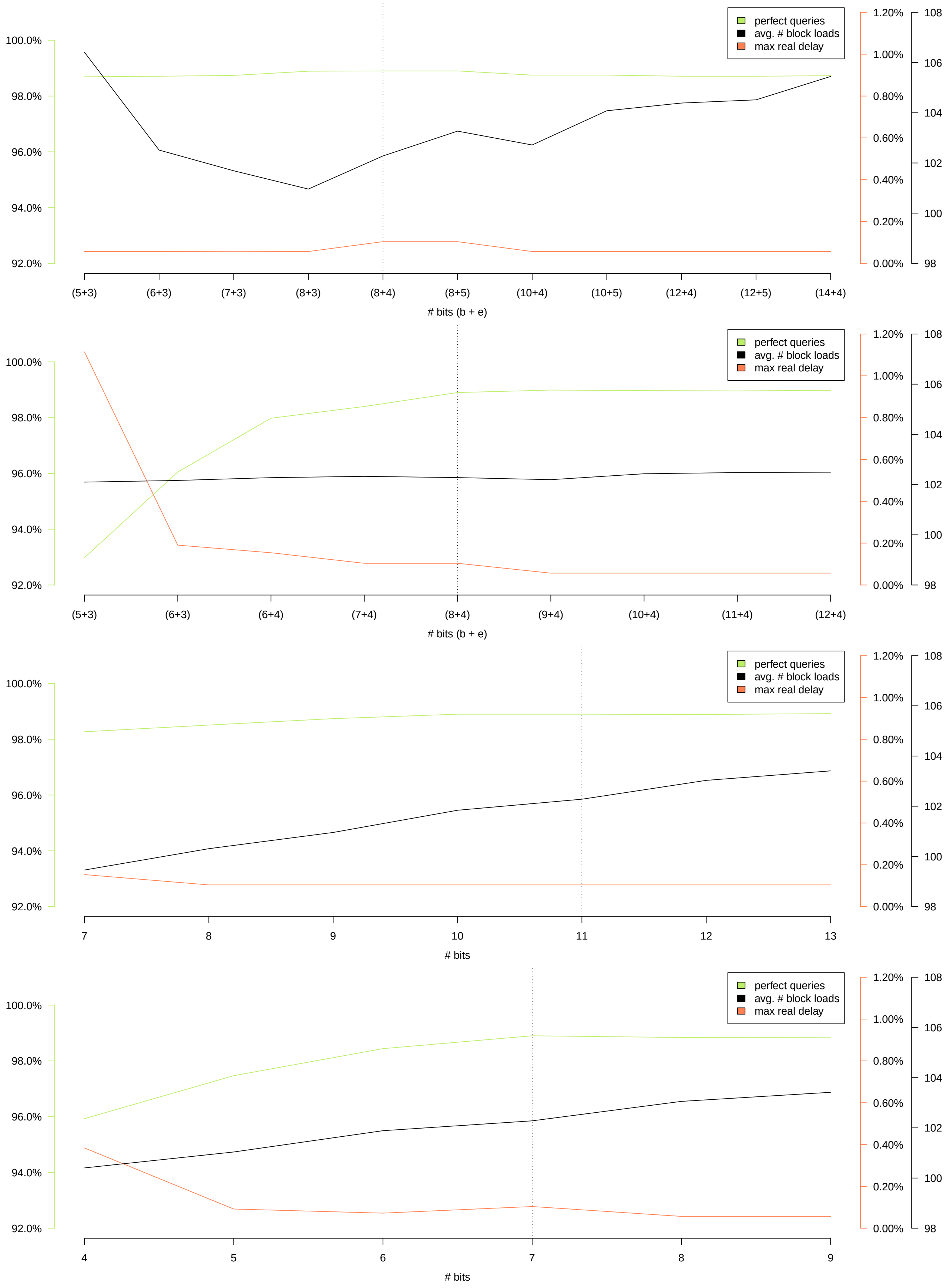

Figure 6: Germany: Influence of changes to the bit-width of a single parameter on the maximum real delay, percentage of perfect queries and average # of block loads. The graphs describe from top to bottom changes to: *min-max*, *const values*, x, y.

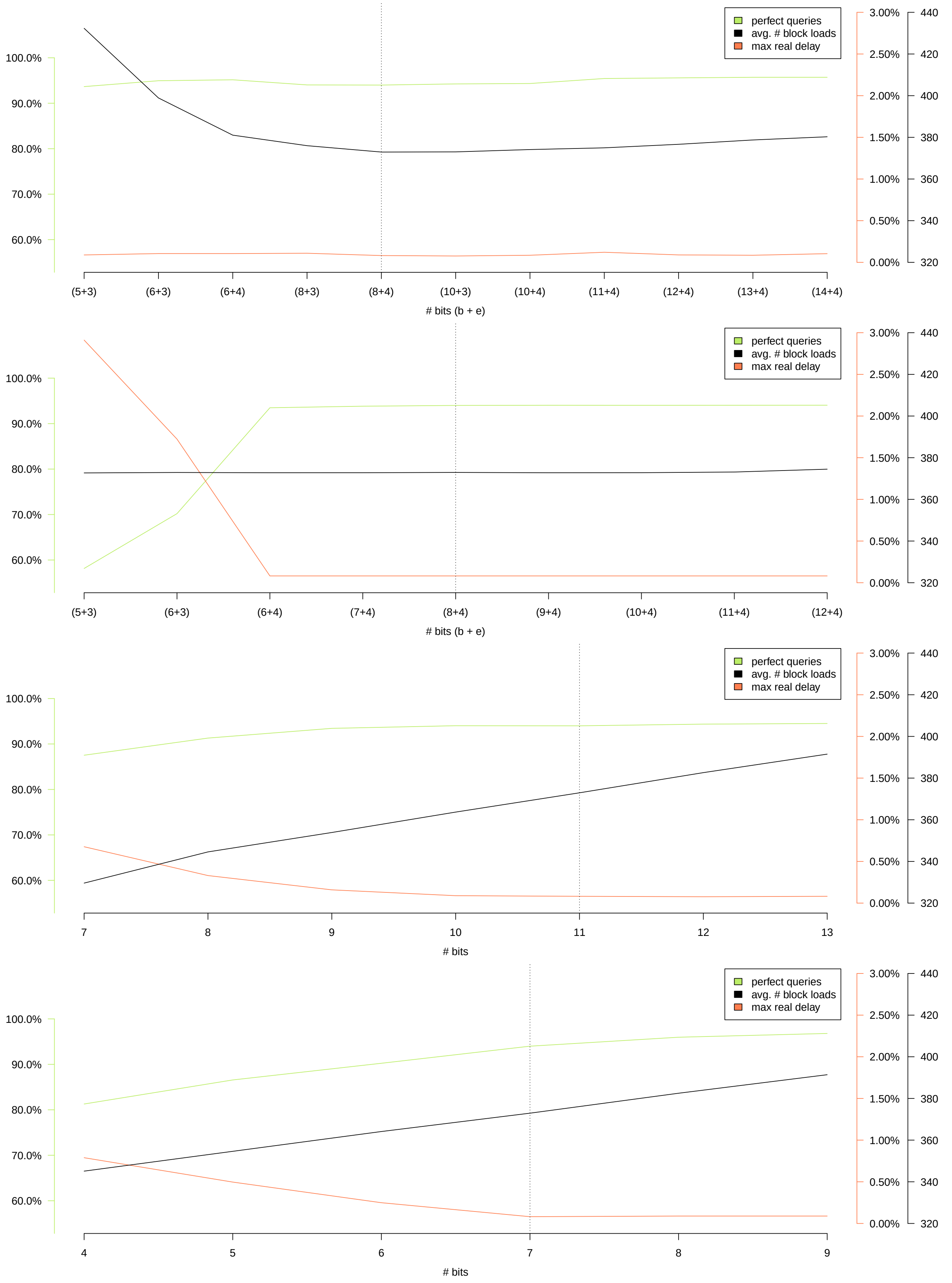

Figure 7: Europe: Influence of changes to the bit-width of a single parameter on the maximum real delay, percentage of perfect queries and average # of block loads. The graphs describe from top to bottom changes to: *min-max*, *const values*, x, y.

of the points. The start and end of traffic changes gets inaccurate and also points that are close in the x value may have to be merged to one point.

While the behavior can be dependent on the raw data we try to provide reasonable values for real word traffic based on the measurements of the two graphs. We chose a bit-width of 11 bit which means that over the day every 42.2 seconds a point can be placed.

The measurements show that low bit-widths increase inaccuracy while bigger bit-widths increase the average number of block loads significantly.

y values of TTFs The y value of TTF points are fixed point values as well. This will use the min and max values as bounds. Only the combination of the bit-width of min and max values and the bit-width of y values produces the final accuracy of TTF values.

Similarly to the x values increasing the bit-width increases the average number of block loads significantly.

6.6 Cache

In our experience, a block cache that has been filled while evaluating a few queries can greatly reduce the number of needed block load operations. Unfortunately, this does not happen during the typical use of mobile devices. The program is expected to execute one query per run which means the cache is always empty. Route recomputation is an exception that occurs if the driver drives off the recommended route. In this case the route has to be recalculated using the current location and the block cache still holds the blocks needed for the original route calculation. Route recomputation is not examined in this thesis but is expected to be harmless because of the local proximity to the original route. There will probably be very few additional block loads during the calculation. Instead we examine the influence of block cache parameters on the performance of queries where the cache is empty.

6.6.1 Cache size

When a block is loaded not all data is used immediately. It has to be held in the main memory as long as possible because data in the block might be needed at a later point of the route calculation. All previous experiments are done using a block cache large enough to make sure no block has to be removed from the main memory. The largest number of block loads that occurres using the standard configuration was 1818 block loads. This is equivalent to 7.1 MiB data to be held in the main memory. Most mobile devices have no problem to handle this amount of main memory but as we found out a much smaller cache is also sufficient.

We examined the performance of different cache sizes between 32 kiB and 4 MiB and also different replacement strategies. The *random replacement* strategy chooses a block at random and replaces it if the cache is full. This strategy marks a reference for comparison with more intelligent replacement strategies. The *FIFO replacement* always replaces the oldest block in the cache. In contrast the LRU replacement keeps track of the data access in blocks and overwrites the block that has not been used for the longest time.

Figure 8 shows the results of the measurements. Note that if there was no cache at all and only one block could be loaded at the same time the German graph would require 647 block loads on average while the European graph would need 2455 . It can be seen that the LRU replacement strategy always performs best. For the European TCH 2 MiB of block cache is enough space to cache blocks for all queries in an nearly optimal way. If a small cache is more important than additional block loads even 512 kiB perform reasonably well. For the German graph as low as

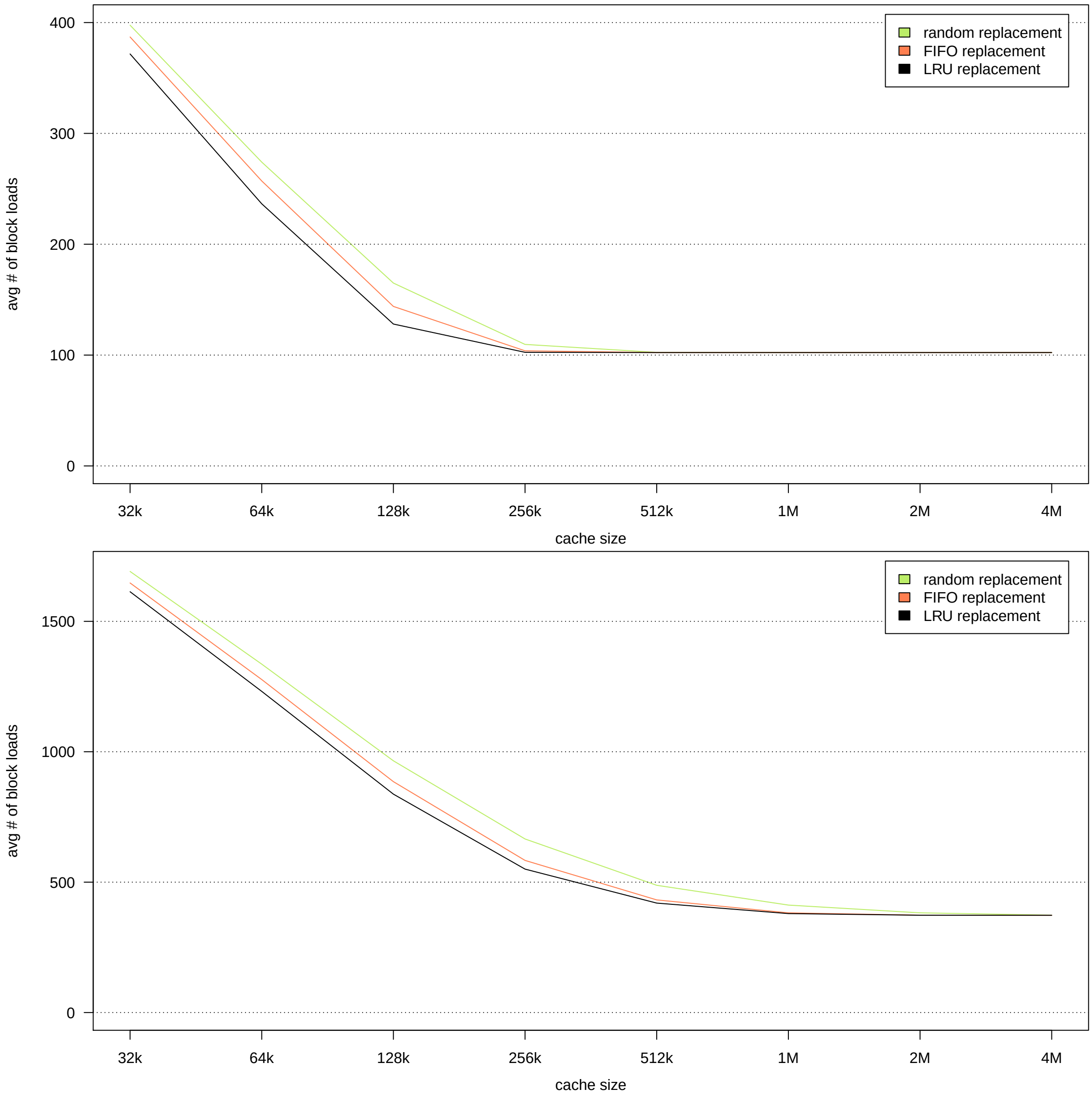

Figure 8: Scaling of the block cache with different sizes and replacement strategies.

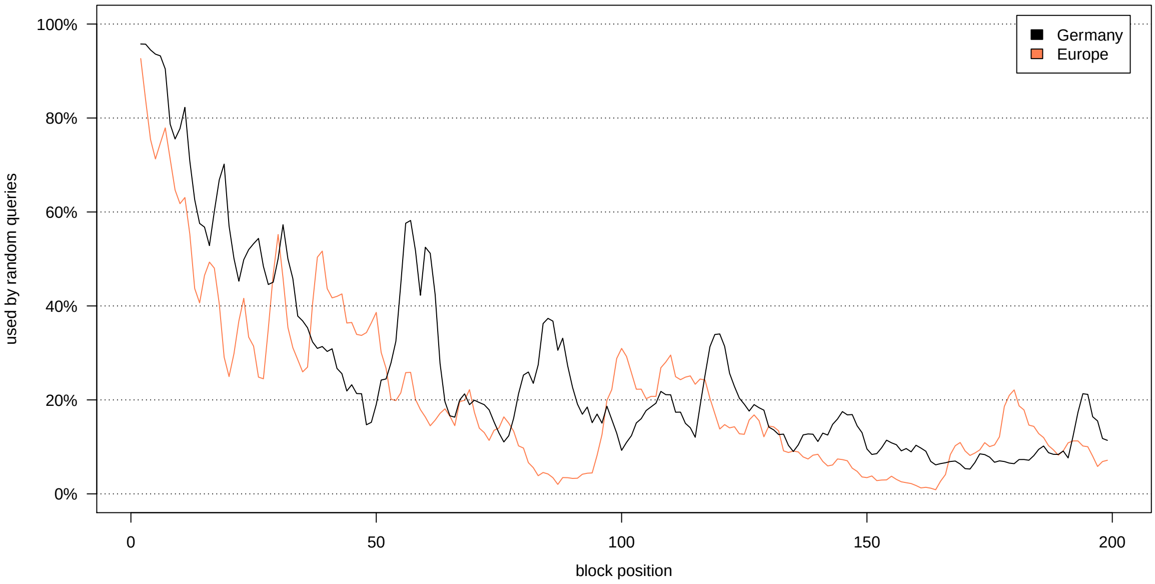

Figure 9: Percentage of 10000 random queries that need to load a specific block in the process. Smoothed with a moving average of 3.

256 kiB still deliver nearly optimal results. Because all mobile devices can spare this amount of main memory, no further analysis has to be done.

6.6.2 Artificial Cache Filling

Since the cache is always empty in the beginning it might be desired to prefill the cache with often accessed blocks. For the BDFS node arrangement there is a simple heuristic to find blocks that are needed for most queries. For this we use the node blocks that are written first and therefore contain the nodes at the beginning of the arrangement. Figure 9 shows the percentage of random queries that, at some point of the query, load a block out of the first 200 blocks that were written to the flash memory. If the cache is prefilled using this method these blocks do not have to be loaded during the query. Figure 10 shows how much the average number of block loads is reduced compared to an empty block cache, when prefilling the cache with a number of blocks.

Nonetheless this is not a recommended practice because this only shifts the problem to a different point of time. More than the time that is saved using the query has to be spent for prefilling the cache. Therefore the query can be calculated faster but the program starts slower.

6.7 Block Size

In all previous tests we chose 4 kiB blocks as reasonable block size. Here we will examine how the number of block loads scales with the access block size. This information is essential to estimate the performance on different kinds flash memory. With ideally arranged data, doubling the access block size would halve the number of required block loads. This is not the case here. Instead an increasing access block size inevitably decreases the percentage of relevant data per block and query.

Figure 11 presents the measurements for access block sizes between 512 B and 16 kiB using the standard configuration. From the results we can conclude that with the exception of the TTF data in the German graph the data scales reasonably well to increasing block sizes.

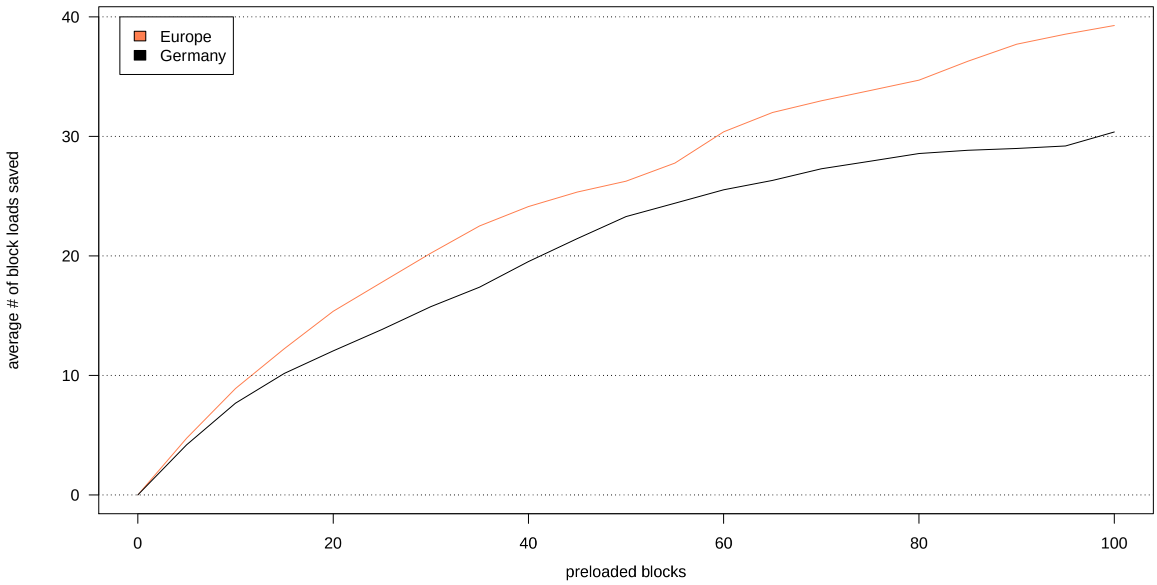

Figure 10: Average reduction of block load operations by preloading the first n blocks.

Please note that the *native block size* of the hardware not necessarily has to be the same as the *access block size*, which is the unit of data the algorithm chooses to load as a whole and for which nodes are aligned during non-continuous writing (see Section 5.1). For the idealized flash memory model we used in this thesis the best access block size is always the same as the native block size of the flash memory. If blocks smaller than the native size are accessed the same time as native block access is needed regardless. For access block sizes larger than the native block size the access time increases proportionally while the percentage of relevant data per block and query decreases. On real devices we estimate that flash memory will provide a higher reading speed for sequential instead of random access. With increasing access block size the random access data rate will approach the level of sequential reading. At the same time the percentage of relevant data in blocks will decrease. Combining these two effects an optimal block size can be determined. Doing that for specific devices could be subject of future research.

7 Conclusions

In this thesis we have presented a data format that can store node, edge and time-dependent travel time data of Time-Dependent Contraction Hierarchies while using as little space as possible. This includes a lossy compression of travel time values. We also presented an algorithm to arrange the nodes in a way where relevant data is as close as possible. This arrangement can then be written in the blocks of the flash memory for which an algorithm was introduced that solves circular dependencies to provide a small representation. All of these measures increase the locality of the data and therefore reduce the number of needed block loads, which is the main performance bottleneck in this case.

Using real world graphs of Germany and Western Europe we reduce the average number of block loads while still maintaining a maximum real delay of 0.1%. The average of totally random route queries across Germany was 102 block load operations which takes about 133 ms on an example device assuming that a block access takes 1.3 ms. This is near 100 ms which a user experiences as instant. For Western Europe 373 block load operations are needed which

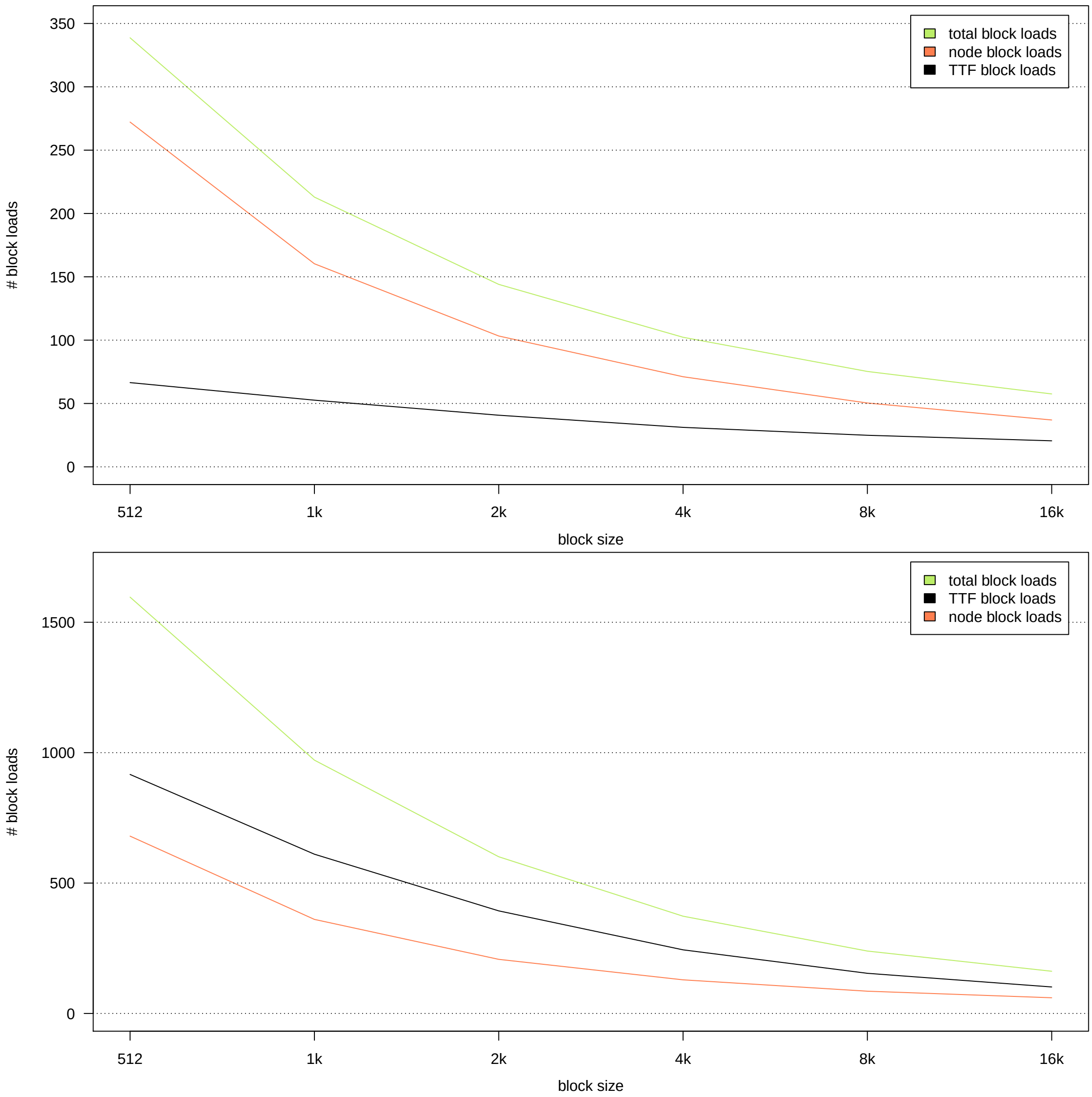

Figure 11: Scaling of the performance of the standard configuration with different access block sizes. For an ideally arranged data the number of block loads would be reciprocally proportional to the block size.

takes 485 ms. This is considerably worse but will not be considered as waiting time by the user. Everyday route queries will be calculated a lot faster.

A block cache that takes at most 2 MiB of main memory is enough which allows this to run on nearly every mobile device that exists. We also took a look at the scaling behavior with different block sizes. This is important to adapt these results to various devices with different data storage. For bigger block sizes the percentage of data that is needed for a query decreases but scales reasonably well.

All these results lay a foundation for an implementation of time-dependent Contraction Hierarchies on mobile devices with a pleasant user experience.

8 Future Work

Shortcut Unpacking. To provide the user with actual navigation information, shortcut edges in the calculated path have to be unpacked to edges of the original graph which represent real road segments. The unpacking information has to be stored on the disk for efficient access. Equivalent to the method proposed by Vetter [16] it might be beneficial to store unpacked paths for some of the shortcuts explicitly. This avoids unpacking these shortcuts recursively where every access to the set of middle nodes can trigger an additional block load.

Storing TTFs More Efficiently. On the graph of Western Europe TTF accesses make up for most of the block load operations. A way to store TTF data more efficiently could improve this situation.

Modeling Turn Costs. To create an even more realistic model of the roads and traffic, turn costs have to be considered. These allow to calculate additional time that is needed for some transitions between roads. Also banned turns can be respected. The easiest method is to create an edge-based graph where nodes represent road segment and edges represent transitions between them. However this increases the number of nodes and edges significantly. For further information refer to Geisberger and Vetter [11].

Generalized Objective Functions. Batz and Sanders examine generalized objective functions [2] that describe TTFs with additional constant costs like energy consumption, distance or tolls. This problem is NP-hard and has been approached using a heuristic. An efficient implementation on mobile devices can be subject of future work.

References

[1] Gernot Veit Batz, Robert Geisberger, Peter Sanders, and Christian Vetter. Minimum Time-Dependent Travel Times with Contraction Hierarchies. *ACM Journal of Experimental Algorithmics*, 2010. Accepted for publication.

[2] Gernot Veit Batz and Peter Sanders. Time-Dependent Route Planning with Generalized Objective Functions. In Leah Epstein and Paolo Ferragina, editors, *Proceedings of the 20th Annual European Symposium on Algorithms (ESA'12)*, volume 7501 of *Lecture Notes in Computer Science*. Springer, 2012.

[3] Brian C. Dean. Shortest Paths in FIFO Time-Dependent Networks: Theory and Algorithms. Technical report, Massachusetts Institute Of Technology, 1999.

[4] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Engineering Route Planning Algorithms. In Jürgen Lerner, Dorothea Wagner, and Katharina A. Zweig, editors, *Algorithmics of Large and Complex Networks*, volume 5515 of *Lecture Notes in Computer Science*, pages 117–139. Springer, 2009.

[5] Daniel Delling and Dorothea Wagner. Time-Dependent Route Planning. In Ravindra K. Ahuja, Rolf H. Möhring, and Christos Zaroliagis, editors, *Robust and Online Large-Scale Optimization*, volume 5868 of *Lecture Notes in Computer Science*, pages 207–230. Springer, 2009.

[6] Ugur Demiryurek, Farnoush Banaei-Kashani, and Cyrus Shahabi. A case for time-dependent shortest path computation in spatial networks. In Divyakant Agrawal, Pusheng Zhang, Amr El Abbadi, and Mohamed F. Mokbel, editors, *Proceedings of the 18th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS'10)*, pages 474–477, 2010.

[7] Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

[8] Stuart E. Dreyfus. An Appraisal of Some Shortest-Path Algorithms. *Operations Research*, 17(3):395–412, 1969.

[9] Robert Geisberger and Peter Sanders. Engineering Time-Dependent Many-to-Many Shortest Paths Computation. In *Proceedings of the 10th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'10)*, volume 14 of *OpenAccess Series in Informatics (OASIcs)*, 2010.

[10] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In Catherine C. McGeoch, editor, *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 319–333. Springer, June 2008.

[11] Robert Geisberger and Christian Vetter. Efficient Routing in Road Networks with Turn Costs. In Panos M. Pardalos and Steffen Rebennack, editors, *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*, volume 6630 of *Lecture Notes in Computer Science*, pages 100–111. Springer, 2011.

[12] H. Imai and Masao Iri. An optimal algorithm for approximating a piecewise linear function. *Journal of Information Processing*, 9(3):159–162, 1987.

[13] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, 2008.

[14] Sabine Neubauer. Space Efficient Approximation of Piecewise Linear Functions. Studienarbeit, Universität Karlsruhe (TH), Fakultät für Informatik, 2009. http://algo2.iti.kit.edu/download/neubauer_sa.pdf.

[15] Ariel Orda and Raphael Rom. Shortest-Path and Minimum Delay Algorithms in Networks with Time-Dependent Edge-Length. *Journal of the ACM*, 37(3):607–625, 1990.

[16] Peter Sanders, Dominik Schultes, and Christian Vetter. Mobile Route Planning. In *Proceedings of the 16th Annual European Symposium on Algorithms (ESA'08)*, volume 5193 of *Lecture Notes in Computer Science*, pages 732–743. Springer, September 2008.

[17] Dominik Schultes and Peter Sanders. Dynamic Highway-Node Routing. In Camil Demetrescu, editor, *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*, volume 4525 of *Lecture Notes in Computer Science*, pages 66–79. Springer, June 2007.