Masterthesis

Optimal Route Planning on Mobile Systems

Adrian Batzill

Hochschule Ravensburg Weingarten
Fakultät T

Bearbeitungszeitraum

03. 11. 2015 – 02. 05. 2016

Betreuer

Prof. Dr. Wolfgang Ertel

Gutachter

Prof. Dr. Wolfgang Ertel

Zweitgutachter

Prof. Dr. Sebastian Mauser

Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Ort, Datum	Unterschrift

Contents

Abstract

While route planning algorithms have seen quite a lot of scientific attention in the last couple of years, many papers focus on maximizing performance. While performance is important for computing **optimal** routes in large road networks, especially on mobile systems, there are more ingredients required for a modern navigation system. This thesis focuses on route planning in low memory and low performance environments, and fills the gap between route planning algorithms and efficient mass storage access.

The proposed external memory implementation allows fast, simple and transparent access to structures that are stored on secondary storage, not only for the shortest-path algorithms, but also for complete navigation systems with everything that is needed beyond route planning. Additionally, techniques to optimize a graph for fast exploration on slow devices are proposed and tested for performance in different hardware configurations.

1 Introduction

1.1 Motivation

Location based services have become more and more important in our daily life. One very popular location based service that has received a lot of attention in the past few years are navigation systems, and especially the route planning algorithms used in these systems. In the early days of navigation systems, most of them worked completely offline. I.e. a mobile device with some kind of large secondary storage that stored the route planning graph. However, as these systems needed to be as cheap and small as possible, their hardware was always fairly slow. To allow the computation of long routes, they usually produced suboptimal, heuristic paths to make computation faster (and yet, according to [SSV08], computing random routes through Europe on a popular mobile device - the TomTom One XL - resulted in an average query time of 59 seconds). In the last few years, with the wider availability of mobile broadband connections via UMTS and LTE, more and more route planning software has switched to online route planning that queries a remote server to compute the route and send it back to the mobile device. This brings several advantages: faster route computation, simpler algorithm engineering and always-up-to-date data and traffic information[1]. However, mobile broadband is still not available everywhere. While this approach works good in cities, the connection is often poor in rural areas. Therefore, it is desirable to have the best of both worlds: fast computation of optimal routes without the need of an Internet connection. This thesis provides the necessities to achieve this: A highly performant graph storage model for secondary storage, memory management for secondary storage to reduce

[1] In several countries, mobile devices can also receive traffic information via a system called TMC, which operates on the radio infrastructure. However, this information is always quite inexact and many systems don't even support it. E.g. it only allows for 64.000 hard coded locations, which even change from time to time, resulting in false or missing alerts when devices are not up to date. Its successor, TPEG, is still lacking support.

RAM requirements of the computation itself, as well as optimizations for modern, fast and optimal algorithms, which - until now - have mainly been used on high-performance servers with large amounts of RAM.

1.2 Related Work

1.2.1 Scientific

Finding the shortest route in a road network is a field that received quite a bit of attention in recent years and several algorithms have been implemented for this thesis, which are discussed in chapter 2. In addition to the ones mentioned there, other methods have been proposed.

The fastest algorithm to date is *Transit Node Routing*, described in [BFSS07], which can compute continent-wide routes in just a couple of microseconds. During precomputation, a large distance table that stores precomputed distances between *important* nodes (transit nodes), such as highway entrances and exits, is computed. Then, for each node, a set of *access nodes* is computed, which contains all transit nodes that are relevant for long distance queries from this node. The query algorithm then mainly consists of a few table lookups for long distance queries: distances from source to its access nodes, from target to its access nodes and the distances between these access nodes are compared to find the combination that minimizes cost of the complete path. While this method is very fast, it is hard to make it dynamic with respect to changing edge weights (traffic) and requires a lot of preprocessing time and storage space.

Another method is *Customizable Route Planning* (CRP) [DGPW11]. The basic idea is a multi-level Dijkstra (see section 2.2 for details): During preprocessing, the graph is split into a constant amount of regions of approximately even size. For these regions, all boundary nodes - nodes that are the source or target of an edge that crosses the region's boundary - are determined. All boundary nodes of a region are then connected to form a full clique and the cost of the shortest paths between the boundary nodes is used as the edge weight for the direct connection. For the query, a bidirectional Dijkstra is used. However, only the target and source regions need to be searched completely. For the regions in between, only the clique needs to be searched, so the search *jumps* from region to region to reduce the search space.

Another performance optimization can be done by dividing the regions even further into multiple sub- and sub-sub-regions. Note that this is similar to the Contraction Hierarchies (CH) algorithm described in section 2.5, but only with a fixed number of levels, instead of one level per node in the graph. The full clique represents the CH shortcuts.

Similarly, *Arc flags*[KMS06] also splits the graph into k regions. Then, for each edge, k flags are precomputed. Flag i is set to 1, iff there exists a shortest path that uses this edge to reach region i. For a query to a node inside region i, a modified version of Dijkstra is used, which skips edges that don't have flag i set to 1 (except for edges that are already inside region i).

However, **mobile** route planning that allows for dynamic edge weight changes has not been investigated as much. For Contraction Hierarchies, an external-memory implementation has been proposed by Sanders et. al. in [SSV08]. While their implementation successfully maximizes performance, it is quite impractical: the only property of edges is their edge weight, which is enough for computing routes, but not enough for a navigation system that requires much more information, such as road types, street names, coordinates, speed limit, etc. Additionally, the storage format is inflexible with respect to dynamic updates. While not all these points are discussed in this thesis, a storage format is proposed that can easily adapt to arbitrary data and is completely flexible and customizable.

In [GW05], an external memory implementation of A* with Landmarks/Triangle Inequality (refer to section 2.3 for more information) is proposed. However, the focus is again put on maximum performance rather than real-world applicability.

1.2.2 Practical

Beside the commercial products on the market, several OpenSource projects have been implementing route planning or complete navigation systems in the past few years. Especially with the rise of OpenStreetMap, which provides free map data, these projects have gained attention.

The most notable projects, which offer decent performance, are GraphHopper[2] and OSRM[3]. Both projects use the modern *Contraction Hierarchies* algorithm described

[2] `https://graphhopper.com`
[3] `http://project-osrm.org`

in section 2.5. However, both are focused on high-performance servers and store the whole graph in main memory. GraphHopper also allows reading the graph from a memory mapped file. But since it maps the whole file at once, this only really works on 64 bit systems, as 32 bit systems don't provide enough memory addresses to map a reasonably large graph.

On the mobile side, there is MoNav[4], which works fairly well but has been unmaintained for several years now and Gosmore[5], which suffers from the same problem on 32 bit systems as GraphHopper does and is also pretty slow on mobile systems.

Finally, Navit[6], which is arguably the oldest of the ones mentioned here (it came to life somewhere around 2005, according to GitHub code statistics), is a fairly flexible solution and can work with many different graph storage formats. E.g. it includes a reverse engineered version of Garmin's[7] proprietary format and can therefore operate directly on Garmin files. It has no problems with large graphs and has been ported to many mobile devices. However, quick experiments show that is is fairly slow. For routing it uses a heuristic Dijkstra that prunes the search space by removing smaller roads when far away from source/target, which may result in suboptimal routes.

[4] `https://code.google.com/archive/p/monav/`
[5] `http://wiki.openstreetmap.org/wiki/Gosmore`
[6] `http://www.navit-project.org/`
[7] A popular commercial navigation system brand: `http://www.garmin.com`

2 Route Planning Algorithms

The route planning problem can roughly be viewed as: Given a road network, find the *best*[1] route between two points. Intuitively, this problem can be solved by implementing the road network as graph, where the nodes represent junctions and the edges represent roads between junctions and then solving the shortest path problem on this graph. In the past few years, this field has seen a lot of attention and many new algorithms have been developed [GH05][GSSD08][DGPW11][BFSS07]. These algorithms usually try to exploit the specific structure of road networks - a sparse graph with hierarchical characteristics - to precompute parts of the route or, in the case of [GH05], precompute a heuristic that guides classical search algorithms.

2.1 Preliminaries

The following definitions and assumptions are preliminary for this thesis and apply to all chapters unless explicitly mentioned otherwise.

- As this thesis is about route planing on road networks, it works with a directed graph $G = (V, E)$, consisting of vertices (also referred to as nodes) and edges, where vertices represent junctions or connections of roads, while the edges represent the roads between these junctions.
- $V = \{n_1, ..., n_i\}$ are the nodes of a graph and $E = \{e_{ij}\}$ are the edges of a graph with source node i and target node j.
- When explaining an algorithm, the term *path* refers to a tuple of vertices $P = (s, v_1, v_2, ..., t)$ of the graph, where s and t represent the start and target vertices respectively, and v_i represent intermediate nodes on the path. Any two consecutive nodes have to be connected via an edge.

[1] Where *best* can have a lot of different meanings.

- Two nodes n_i and n_j are referred to as neighbors, iff there exists a direct edge e_{ij} between them with cost $c(n_i, n_j)$.
- The cost $c(s,t)$ of a path p is regarded as the sum of the cost between the path's nodes, i.e. $c(p) = c(s, v_1) + c(v_1, v_2) + ... + c(v_i, t)$.
- The main goal of the described algorithms is finding the *fastest* route from s to t. Therefore, when referring to *cost* of a path, this usually means travel-time. However, the concepts described here can also be modified to find the *shortest* (in terms of distance), the *most beautiful*, the most *fuel saving*, ... path by using a different cost-function.

2.2 Dijkstra's Algorithm

Dijkstra's Algorithm is arguably the simplest and oldest of the algorithms described in this thesis. It was invented by Edsger W. Dijkstra in [Dij59] and was/is widely recognized as the method of choice for determining the shortest path in a graph. The idea of Dijkstra's Algorithm is based on one simple fact: Given a graph G, as well as a source node s and a target node t, it can be assumed that for any node v_i that lies on the shortest path from s to t, the shortest path from s to v_i is the prefix of the shortest path from s to t. Or in other words, given a shortest path from s to t, a shortest path from s to every node v_i that lies on the shortest path from s to t can be trivially deduced.

With this property, Dijkstra's Algorithm will incrementally construct the path from s to t by determining all potential prefix-paths from s to any node u until reaching t.

Listing 2.1 shows how this idea can be implemented in practice.

Listing 2.1: Dijkstra's Algorithm

```
1 float dijkstra(Graph g, Node s, Node t) {
2   priority_queue<Node> queue; // returns nodes in order of their key
        (= tentative_cost from s) in increasing order
3   s.tentative_cost = 0;
4   s.key = 0;
5   queue.add(s);
6   while (!queue.is_empty()) {
7     Node v = queue.pop();
```

```
8       // Mark node as visited. The shortest path from s to v is now
            known.
9       v.already_visited = true;
10      if (v == t) {
11        return v.tentative_cost; // path found, return cost.
12      }
13
14      // add all unvisted neighbours with their tentative cost
15      for (Node neighbour : g.getNeighbours(v)) {
16        if (!neighbour.already_visited) {
17          neighbour.tentative_cost = v.tentative_cost + g.
                getCostBetween(v, neighbour);
18          neighbour.key = neighbour.tentative_cost
19          queue.add(neighbour);
20        }
21      }
22    }
23    return -1; // queue is empty -> no path found.
24  }
```

Note that this approach will visit quite a lot of nodes until reaching t. The search space can be described as a circle (in terms of cost) with radius $c(s,t)$ around s. An example can be seen in Figure 2.1. Until today, Dijkstra's Algorithm builds the foundation of many other route planning algorithms, or it is used for graph preprocessing to make them faster.

2.3 A* with Straight Line and Landmark Heuristic

As it can be seen above, the search space of Dijkstra's Algorithm is quite large: All shortest paths from s to any node v with $c(s,v) < c(s,t)$ need to be evaluated. However, it seems hard to improve this approach without additional knowledge about the structure of the underlying graph. If more information is present, it can be used to heuristically reduce the search space by making the graph traversal more goal-oriented. Note that this does not mean that the algorithm will compute sub-optimal paths. Heuristics that will still result in optimal paths can be found, as shown below. This kind of heuristic, goal-directed Dijkstra search is called the A*

Algorithm[PEH68]. It works in a very similar way as Dijkstra's Algorithm, with the only exception that nodes are not added to the priority queue with their tentative distance from s as the key, but with an additional heuristic value h representing the estimated remaining cost to the target. Replacing the line

```
neighbour.key = neighbour.tentative_cost;
```

with

```
neighbour.key = neighbour.tentative_cost + h(neighbour, t);
```

where $h(v, t)$ is the heuristic function estimating the cost from node v to t, in Listing 2.1 is the only modification to Dijkstra that is required to implement A*. Hence, Dijkstra's Algorithm can be seen as a special case of A* with $h(v, t) = 0$ for all nodes v. This algorithm is correct (i.e. produces optimal paths), if the heuristic fulfills two properties:

- *monotonicity*: For each node n_i and its successor n_j, $h(n_j, t) + cost(n_i, n_j) \geq h(n_i, t)$ must hold.
- *admissibility*: The heuristic function always needs to underestimate the cost to the target.

Note that a monotonic heuristic is also always admissible. A heuristic that is admissible, but not monotonic will result in sub-optimal path if A* is applied as described above. By modifying the algorithm so that nodes can be visited multiple times, optimality can be restored for non-monotonic heuristics. However, this comes at the cost of a great decrease of runtime performance.

2.3.1 Straight Line Heuristic

One simple heuristic used for A* in practice is the straight line heuristic, which simply estimates the distance to the target by assuming that there is a straight line road to it. The straight line heuristic obviously holds the triangle inequality - and is therefore monotonic - and underestimates the distance to the target, so it can be used to find the shortest path with respect to travel *distance*. It can also be modified for optimizing travel *time*, by dividing the straight line distance by the fastest average speed as assumed by the vehicle model (e.g. for a car profile, that would be something around $120\frac{km}{h}$, as this can be considered the average speed on a freeway). I.e., the heuristic assumes that there is a straight line freeway to the target. As this is usually far from the truth the heuristic seems to be mediocre when

optimizing for travel time. Measurements can be seen in section 5.1.

2.3.2 Landmarks and triangle inequality

A purely graph-based approach to a heuristic for A* was presented by A. Goldberg and Chris Harrelson in [GH05]. In contrast to the straight line heuristic, the ALT (**A***, **L**andmarks, **T**riangle inequality) heuristic does not need any additional heuristic information about the graph layout in advance (such as coordinates that allow computing a straight line distance), but precomputes this information by itself after graph creation and reuses it during query time. This makes the approach applicable to all kinds of shortest path problems, including route planning on road networks, where it typically provides much faster query times than the straight line heuristic, as it can be seen in section 5.1.

ALT Precomputation

For precomputation, a set $l = \{l_1, l_2, ..., l_i\}$ of *landmark* nodes has to be chosen from the graph. Any set of nodes will yield optimal paths, but choosing *good* landmarks will increase performance. For now, a random selection of k nodes (where k is typically something from 5 to 60) is assumed. For each of these landmarks l_i, the cost from l_i to all other nodes n_j (including the other landmarks) in the graph is now computed, as well as the cost from all nodes n_j to the landmarks l_i. Hence, the additional information vector $(l_{1f}, l_{1b}, l_{2f}, l_{2b}, ...)$ for each node in the graph, where l_{if} denotes the **f**orward cost to landmark i and l_{ib} denotes the **b**ackward cost from landmark i is precomputed and stored in the graph. To compute these distances in a reasonable amount of time, one can simply run a full Dijkstra, visiting all nodes, from each landmark to receive all backwards costs. By doing the same on the inverted graph, the forward costs can be computed.

ALT Query

As the ALT algorithm is simply a heuristic for A*, only the heuristic function needs to be exchanged to make use of the precomputed information. As the name of the algorithm suggests, the triangle inequality is used for this. Note that the triangle

inequality must not hold for the graph itself (and it usually does not hold for road networks), but it holds for the shortest paths on the graph:

$$c(n_v, n_l) - c(n_w, n_l) \leq c(n_v, n_w)$$

and similarly

$$c(n_l, n_w) - c(n_l, n_v) \leq c(n_v, n_w).$$

Assuming n_l is a landmark, $c(n_v, n_l)$, $c(n_w, n_l)$, $c(n_l, n_w)$ and $c(n_l, n_v)$ was already computed in the precomputation-stage, so the left side of the triangle inequalities is immediately known and can therefore be used as a lower bound for the query. For good A* performance, the heuristic needs to estimate as close as possible to the real cost. Therefore, one can simply use the maximum of the above triangle inequalities. Since these inequalities hold for all landmarks, the triangle inequalities for all landmarks can be computed and the one that gives us the best (=highest) result is used. Note that more landmarks do not necessarily improve query performance. Beside having increased storage consumption because more cost-values need to be stored, there is also the need that for each expanded node, all triangle inequalities for each landmark have to be checked to find the best one, which slows computation down when too many landmarks are used. In [GW05], Goldberg et al. present methods to reduce the set of landmarks that need to be checked during query time to a relatively small subset of *active landmarks* which change regularly. This technique allows having many more landmarks precomputed without as much penalty during runtime.

Good Landmarks

Since all landmark sets yield optimal paths, selecting a good set of landmarks can optimize query time. In [GH05], several algorithms for selecting good landmarks are presented:

- *random*: Simply select i random nodes from the graph as landmarks.
- *farthest*: Start with a single node in the landmark set and iteratively run Dijkstra's Algorithm, starting with all previously selected landmarks in the priority queue, inserting the last visited node of each run into the landmark set.

- *planar*: Only works for geometric-type graphs, such as road networks: Start with a node relatively close to the center c and divide the rest of the graph into pie-slice-like sectors centered at c. Then pick the node farthest away from the center from each section and use these as landmarks. To ensure that no two landmarks are too close to each other, nodes that are close to the borders will be skipped.

Additionally, *optimized* versions for *random* and *planar* are described, where each landmark in the resulting landmark set receives a score: A fixed sample of vertex pairs is taken from the graph and each landmark is checked whether it gives a decent lower bound for these pairs. Landmarks with low scores (i.e. bad lower bounds) can then be replaced by randomly selecting different nodes from the graph as candidates, or in the case of *planar*, dividing the sections into subsections and determining the farthest nodes in these subsections and scoring them.

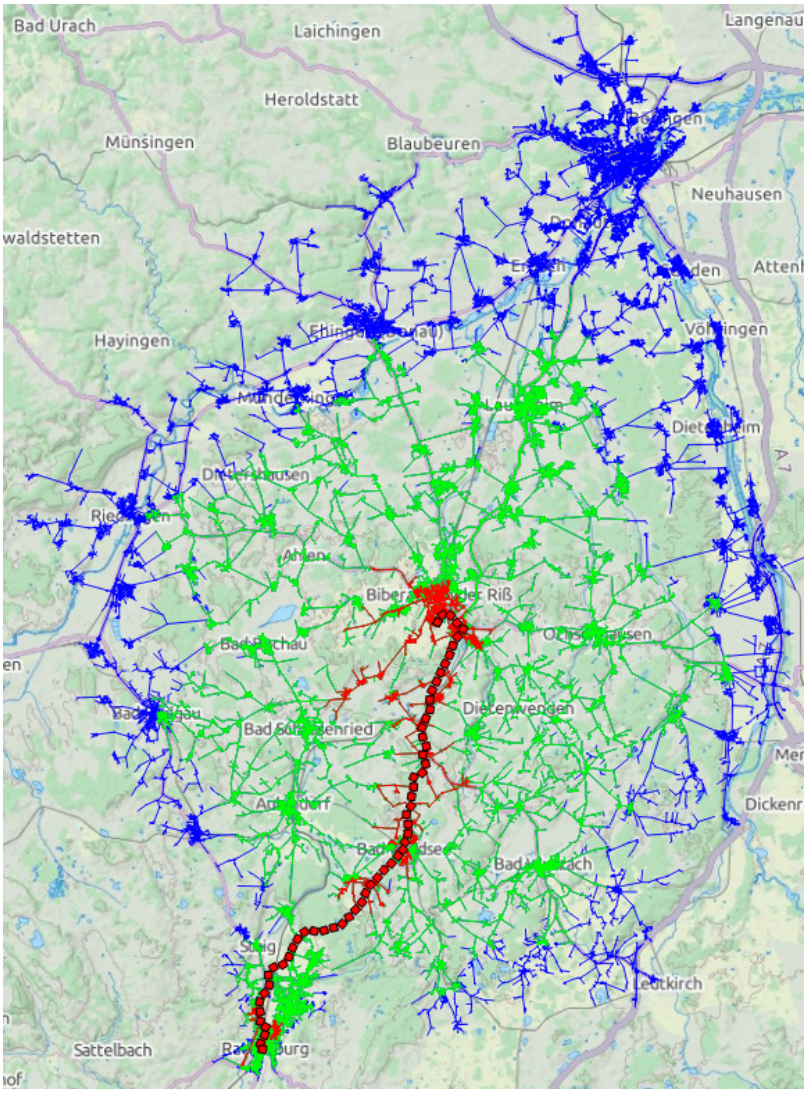

Figure 2.1: Search spaces for unidirectional routing algorithms for a route from Biberach to Ravensburg (~50 km). The Dijkstra search space is shown in blue, the A*-straightline search space in green and the A*-landmarks search space in red (with 20 landmarks on a dataset of Baden-Württemberg).

2.4 Bidirectional Search

Bidirectional search is a common way to speed up classic shortest path algorithms. The basic idea is that instead of searching only from the source to the target node, one also searches from the target to the source node with the same algorithm, but on the inverted graph. The main benefit of this approach is that it can greatly reduce the search space. For Dijkstra's Algorithm, which has a circle-like search

space around the source with radius $c(s,t)$, the search space can be reduced to two circles[2] with radius $\frac{c(s,t)}{2}$.

In case of Dijkstra's Algorithm, a bidirectional version can be implemented by running two Dijkstras d_s and d_t from the source and the target respectively in an alternating way. As soon as d_s settles a node μ that has already been settled by d_t or vice versa, a shortest path has been found with $c(s,t) \leq c(s,\mu) + c(\mu,t)$. Note however, that μ does is not necessarily part of that path. However, the path can then be found by looking at all other nodes that are currently in the priority queue and finding the node that minimizes the total cost.

For A* however, it's not that simple. Since two different heuristic functions are used, one estimating the cost from the source to the target and one estimating the cost from the target to the source, there is no guarantee, that an optimal path is found when these two searches meet [GW05]. A simple method to overcome this problem is using a *consistent* heuristic. Let $h_f(n)$ be the forward heuristic estimating the cost from n to the target and let $h_r(n)$ be the backward heuristic estimating the cost from n to the source in the inverted graph, then h_f and h_r are called consistent iff $h_f + h_r = const$. A simple way to make any heuristic consistent is simply using an *average function* of h_f and h_r[IHI+94], so that

$$h'_f(v) = \frac{h_f(v) - h_r(v)}{2}$$

and

$$h'_r(v) = \frac{h_r(v) - h_f(v)}{2} = -h'_f(v).$$

However, speeding up unidirectional algorithms is not the only strength of bidirectional search. It also allows a new approach to shortest path computation in general: hierarchical routing. Due to the hierarchical structure of road networks (long routes usually travel from slow, local roads towards fast highways and then back to slow local roads), a hierarchical approach that exploits this structure seems intuitive. Different approaches to using this structure in a way that allows for optimal routes have been developed in the past few years, such as *Highway Hierarchies* [SS05], *Multi Level Dijkstra* (in the form of *Customizable Route planning*

[2]Since the radius has a quadratic influence on the area of a circle, the reduction becomes greater for long routes/increasing $c(s,t)$.

[DGPW11]), *Transit Node Routing* [BFSS07] and the - by now quite popular[3] - approach called *Contraction Hierarchies* [GSSD08].

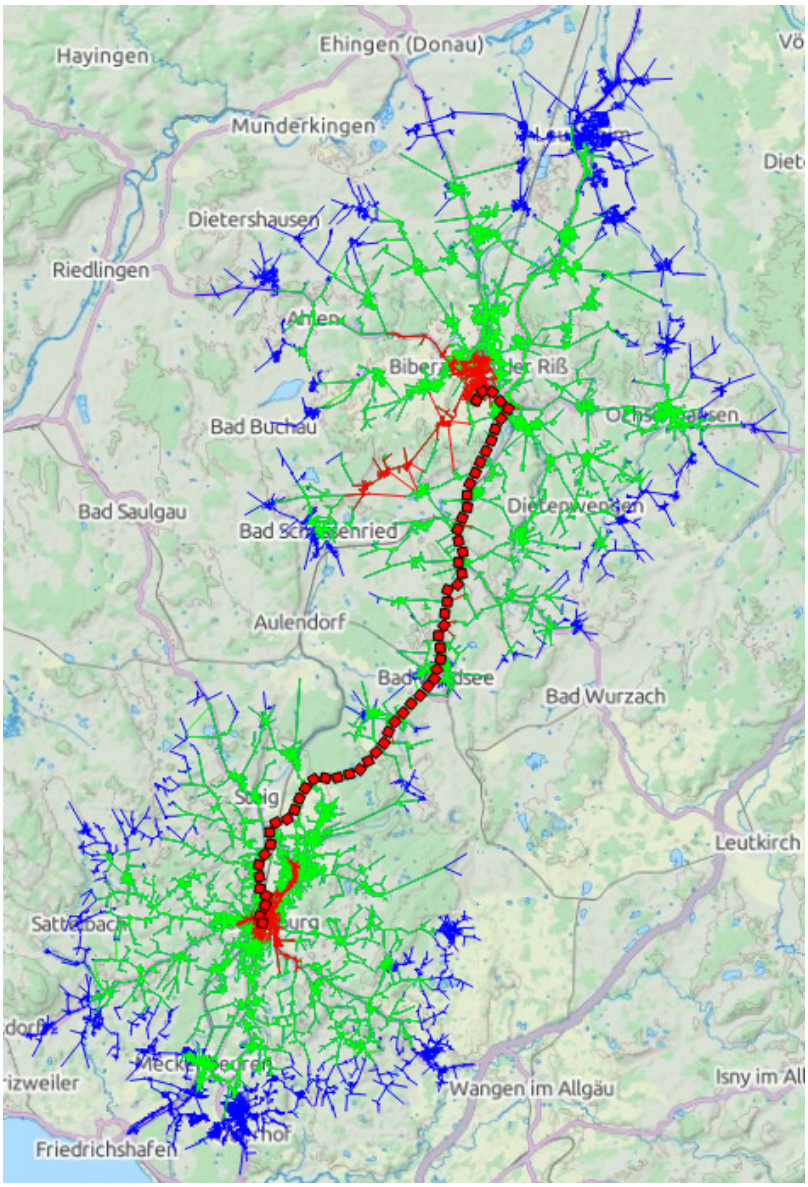

Figure 2.2: Search spaces for bidirectional routing algorithms for a route from Biberach to Ravensburg (~50 km). The bidirectional Dijkstra search space is shown in blue, the bidirectional A*-straightline search space in green and the bidirectional A*-landmarks search space in red (with 20 landmarks on a dataset of Baden-Württemberg).

[3]It can be found in various open source projects such as GraphHopper, OSRM, MoNav and others.

2.5 Contraction Hierarchies

Contraction hierarchies is a purely graph based approach to hierarchical route planning. Similarly to ALT, it is a two-phase algorithm. During preprocessing, hierarchical properties of the graph are found and exploited to generate *shortcuts*. During query time, these shortcuts are then used to efficiently compute the shortest path [GSSD08].

Preprocessing

Assume that all nodes of a given graph are numbered *1..n* in order of their ascending *importance*[4]. In this order, all nodes of the graph are now *contracted*. A node v is contracted by removing it from the graph in such a way that shortest paths in the remaining graph G' are preserved. This can be achieved by replacing all paths (u, v, w) by a shortcut (u, w). The shortcut, however, is only needed if (u, v, w) is the **only** shortest path from u to w[GSSD08]. To decide if a shortcut is needed, a modified Dijkstra search for each source node u that is the source of an inbound edge of v is used. This search needs to be modified to

- ignore node v during the search and
- stop when all nodes w that are the target of an outbound edge of v are visited.

The search can also stop when it has reached the cost limit of $c(u, v)+max\{c(v, w)\}$. This search is called *witness search* and its result will show which shortcuts (u, w) are needed when v is removed.

Node Ordering

Finding a good node order is crucial for a well performing query algorithm, as it has a major influence on the number of shortcuts that have to be added. Intuitively, a node order that adds as few shortcuts as possible is desired. Hence *unimportant* nodes should be replaced first, so that few shortcuts need to be added in the beginning, resulting in a smaller graph for the contraction of important nodes later on. According to [GSSD08], a simple linear combination of several local properties

[4] Any order will result in optimal routes, but having a better node order will result in a smaller graph with better query performance.

is enough to determine a viable node order. The most important terms mentioned are:

- *Edge Difference*: The difference of the edges that will be removed and the shortcuts that have to be added when contracting v.
- *Uniformity*: The number of already contracted neighbors of v.
- *Cost of contraction*: The search space sizes of the witness search.

Since all these terms can change while contracting the complete graph, it is necessary to update the ordering after each contracted node. Since the complete recomputation of the ordering can take a lot of time, several strategies are presented to make these updates more efficient:

- *Lazy update*: Before v is contracted, recompute its priority (Edge difference, Uniformity, ...) and check if it is still the most attractive node to contract.
- *Neighbor update*: After contracting v, the priority of all its uncontracted neighbors is updated.
- *Periodic update*: Periodically reevaluate all priorities of the uncontracted nodes.

Query

For the shortest path query, a bidirectional Dijkstra search is employed. However, only edges to a node of higher order are visited during the search: When visiting a node n with hierarchy level l, it is known, that all nodes with a level $< l$ have been contracted before n. As explained earlier, the contraction of these nodes did not change the length of paths in the graph. By searching upward from both directions as explained in section 2.4, these two searches will eventually meet, resulting in a path in G' that has equal cost as the shortest path in G. To obtain the route in G, the path now needs to be unpacked to retrieve the *original* edges in G. This can be achieved by not only adding shortcut edges during preprocessing, but remembering which nodes are actually shortcutted. These shortcuts can then be recursively unpacked to get the original path in G.

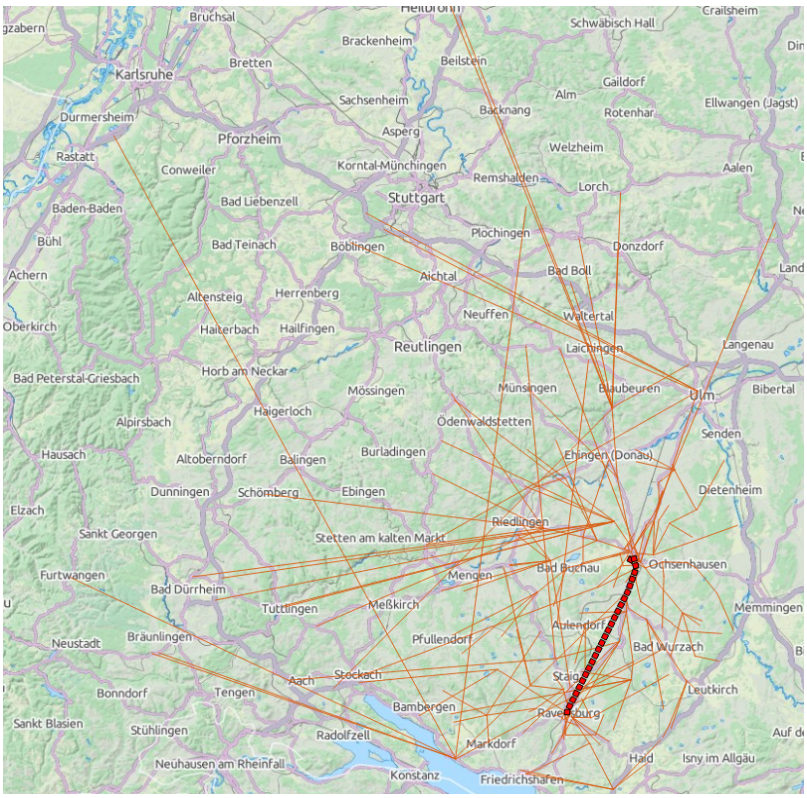

Figure 2.3: Search space for Contraction Hierarchies for a route from Biberach to Ravensburg (~50 km). Edges of the path have not yet been unpacked, so the resulting route shows a very long straight edge - a shortcut over an important road. Due to the precomputation, the search space is far more spread out, but the total number of visited nodes is lower (hence, resulting in faster route computation). Obviously, routes over such far away nodes are suboptimal and a technique called *stall on demand* is described in [GSSD08], which can heuristically prune such nodes.

3 External Memory

For many years now, personal computers - and also mobile systems - usually provide a storage hierarchy, ranging from fast and expensive (and usually volatile) *primary storage* (CPU registers, caches, DRAM, ...) to large, persistent and cheap *secondary storage* mechanisms (harddisks, solid state drives, SD-Cards, ...). From a programmer perspective, it is usually very easy to access the faster storage mechanisms. E.g. in the C and C++ programming languages, CPU registers and caches are handled completely automatically by the compiler or hardware itself and access to DRAM is directly provided by the programming language in a very natural way by the means of malloc/free/new/delete and the concept of pointers. Access to harddisks, however, is much more tedious if one wants to use it for runtime data. Since route planning data for multiple countries is usually quite large - depending on the representation usually several gigabytes or for the whole world even tenth of gigabytes - storing this data in main memory (DRAM) is not always feasible. While it might work on a modern, high performance server machine, reasonably priced mobile devices usually don't have enough DRAM to do so. Another disadvantage of only storing route planning data in main memory is the fact that it is volatile, meaning that the graph would have to be rebuilt/parsed after each reboot.
This is where the concept of *External Memory* comes into play. The main idea is to make access to secondary storage just as convenient as to primary storage, so that the developer can focus on implementing algorithms and doesn't have to fiddle with the underlying storage mechanism.

3.1 STXXL

STXXL is a high level approach to external memory for C++. Instead of facilitating direct access to secondary storage, it provides convenience containers and algorithms with a similar API to the ones that can be found in the C++ *Standard*

Template Library (STL). The most notable ones (at least for route planning) are secondary storage implementations of *vector, stack, set, priority_queue* and *map*. The advantage of such a high level abstraction is that STXXL can do many optimizations under the hood to improve performance. According to [DKS08], STXXL offers

- transparent support of parallel disks (algorithms work on multiple disks in parallel),
- support for problem sizes of up to dozens of terabytes and
- overlapping of I/O and computation to improve utilization of computer resources.

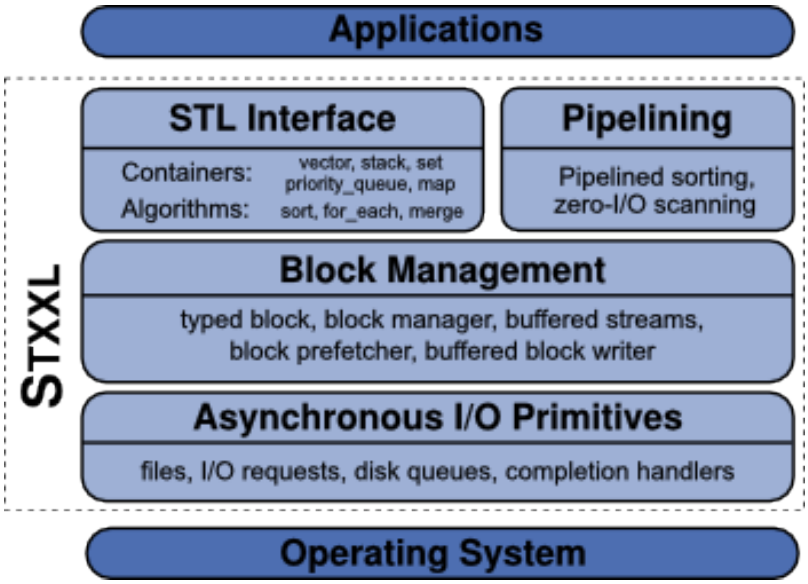

Figure 3.1: STXXL library design, showing that the application can use an STL-like interface. Read/write requests to STXXL containers are forwarded to the block management layer, which handles abstraction from external memory algorithms to I/O by implementing block allocation/deallocation, parallel disk support (striping, randomized striping, cycling, etc.), prefetching and caching. The lowest level is an abstraction of asynchronous I/O from the operating system APIs to make the library easily portable to different operating systems. [DKS08]

STXXL has been used successfully in several route planning applications, such as OSRM (*O*pen *S*ource *R*outing *M*achine), which is a route planning engine for server systems that uses STXXL to store its graph, and multiple research projects from KIT (*K*arlsruhe *I*nstitude of *T*echnology).

3.2 Memory Mapped Files

Memory mapped files are a more classical and low level approach to external memory. They are well known from different database systems that utilize those memory mapped files for low level storage[1]. However, mmap[2] based I/O can also be found in route planning engines[3]. The basic concept of memory mapped I/O is that instead of using conventional file reading/writing mechanisms (read()/write() system calls) to copy a buffer of data from a disk into main memory, the buffer is only mapped into main memory by using the operating system's memory management mechanisms. Figure 3.2 shows the basic idea behind mmap.

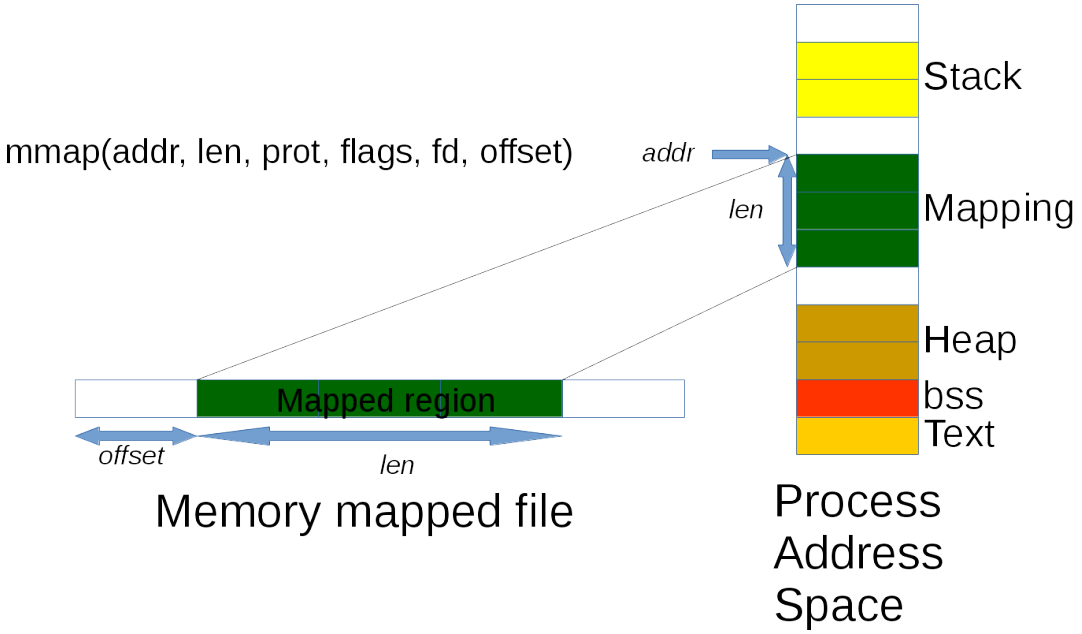

Figure 3.2: The mmap() system call maps a region of a file into the process's address space, returning a memory address (pointer) that can be dereferenced to access the file's content. Note that the mapped data is usually not loaded into memory when calling mmap(), but will be loaded on demand in chunks by the operating system. If the *addr* parameter is *NULL*, the operating system will choose a suitable location automatically and return it.

The programmer can open a file and use the mmap() system call to map a specific

[1] MongoDB uses it by default, MySQL's MyISAM and SQLite can be configured to do mmap-based I/O.

[2] mmap() is the POSIX system call to map a file into memory. Windows provides a similar interface with the CreateFileMapping and MapViewOfFile system calls.

[3] Gosmore uses it by default, GraphHopper provides an mmap storage backend for low memory environments.

region of that file into main memory. The mmap() call will return a pointer to the beginning of this region, which can be dereferenced to access the data in that file. Upon calling mmap(), no data is actually read from the file (except if *MAP_POPULATE* is specified in the *flags* argument). Instead, all memory pages in the mapping will have the *dirty-bit* set. When a pointer into the mapping is dereferenced and the *dirty-bit* is set, the hardware's *MMU* (*M*emory *M*anagement *U*nit) will synchronously signal a *page fault* to the kernel, which will then read the required bytes from disk into the requested memory location. When writing to a memory location inside a mapping, the behavior depends on the *flags* argument of mmap(). If *MAP_SHARED* is specified, the changes will be written back to disk. If *MAP_PRIVATE* is specified, they will only only be reflected in the process's address space (also called *copy-on-write mapping*). If physical memory is filling up and no more pages can be read, old, unused pages will be evicted from the mapping (and potentially be written back to disk)[4]. The main advantages of memory mapped file to conventional I/O with read()/write() are:

- When reading via read(), the kernel will read the requested bytes from disk into a buffer in the kernel's address space and then copy these bytes into the process's address space. When using mmap(), performance can be improved as these bytes don't necessarily need to be copied twice, but will be read directly into the process's address space.
- Since memory mapped regions are handled by the operating system's memory management code (and ultimately the hardware memory management unit), the memory can be managed in a cooperative way. I.e., the application can use all available physical memory as a cache without becoming unstable or crashing other applications on the system. If physical memory is needed for an allocation, the kernel will swap out mapped pages and potentially write them back to disk, freeing up memory for other needs.
- It is fairly simple to use: one can store any POD (*p*lain *o*ld *d*ata) objects (or in the case of C++, *standard layout* types) in the mapped region and handle them as if they were in main memory. All I/O abstraction is done by the operating system automatically.
- It can be assumed, that the memory management code of modern operating

[4]For more details, refer to the manpage at `http://man7.org/linux/man-pages/man2/mmap.2.html`.

systems is stable and fairly well optimized with respect to caching and page fault handling.

However, it also comes with disadvantages:

- High dependency on the operating systems memory management code, which is aimed for general purpose use. As this code can't be changed easily, it is hard to optimize any further than the operating system maintainers have done already; Using domain-specific logic to influence caching and prefetching becomes harder.
- While a lot of work is done by the operating system, managing the data in the mapping still needs to be done by the programmer. E.g. it is not recommended to store pointers in a memory mapped file, or pointers to mapped data. Pointers inside the mapped file will become invalid when the application is restarted and pointers to mapped data will become invalid when calling munmap() and then mmap() again, as the mapping might be at a different memory location.
- As of today, most mobile systems are still often running on 32 bit processors. This means, that a process can theoretically map at most $2^{32}-1$ bytes on those devices. In practice, however, this number has turned out to be a lot smaller: for a mapping, mmap requires *len* **continuous** free addresses. However, some of these 2^{32} bytes are already used by the program stack, heap, code, etc. Especially on systems with *address space layout randomization*, continuous address space has shown to often be around 250-500 MiB.

Especially the latter disadvantage makes memory mapped files seem unusable for route planning, as graph files often span multiple gigabytes. However, this is where *chunked memory mapped files* help.

3.2.1 Chunked memory mapped files

Chunked memory files are the natural technique to overcome the limitations of memory mapped files on 32 bit systems. Instead of mapping the complete file at once, which fails for large files, only equally sized chunks of size N are mapped, until a specific maximum of M chunks. When more pages are accessed, old pages are evicted/unmapped in an LRU[5] fashion. Note that this LRU mapping/unmapping

[5]least recently used - the chunk that hasn't been used for the longest time is evicted first

happens at one level higher than what the operating system's page faulting/loading/swapping: For each mapped chunk, the system's mechanisms are still intact, meaning that N can be at roughly 250 MiB without issues or slowdowns. Larger N would actually speed up the program, as less evict/unmap/mmap cycles would need to be done for most memory access patterns.

For the first implementation of chunked memory mapped files for this thesis in C++, a custom pointer class was developed that represents a *pointer into the memory mapped file*. Internally, it uses a `uint64_t` to store an address - an offset into the file - and it overloads the common operators * and ->. Using operator -> will ensure that the requested offset is actually mapped and returns a real C++ pointer into the mapping. Using operator * will also ensure that the offset is mapped and return a real C++ reference into the mapping. Whenever such access happens, it will be determined to which chunk the requested offset belongs and the whole chunk will be mapped. The resulting C++ memory address can then be determined by computing `chunk.begin_ptr + (ptr.offset - chunk.offset)`. Since the dereferencing of the pointer could potentially cause access to multiple bytes (and in case of a huge POD data type, even a few Kb), a constant amount of P Kb more than the chunk size N is mapped, to ensure that enough bytes ares available.

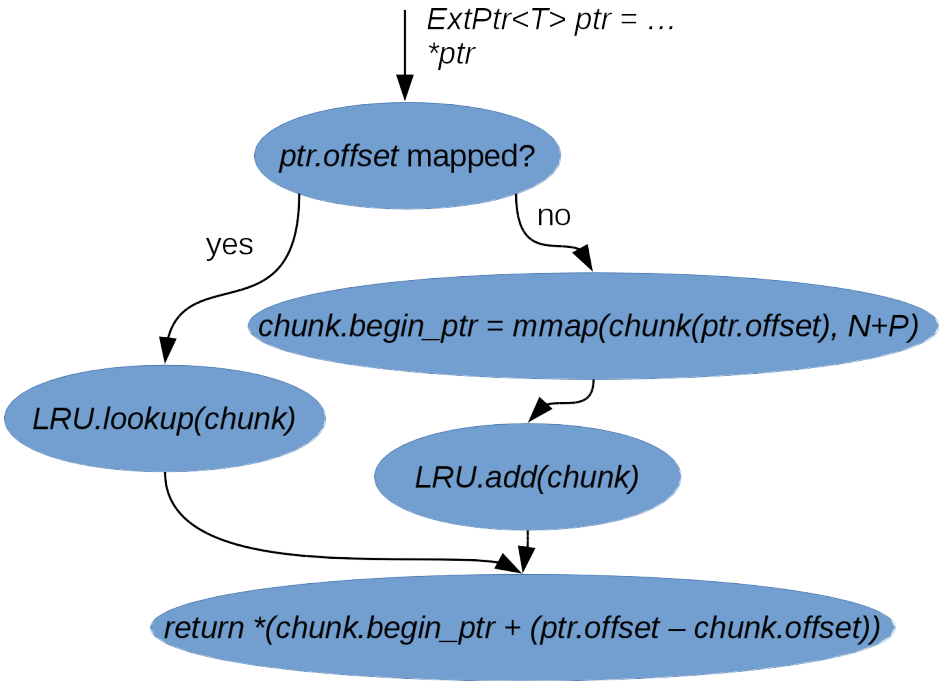

Figure 3.3: Chunked mmap flowchart - When a specific offset in the file is requested, it is either mapped already and can be returned, or needs to be mapped first.

While this method works for C-like structs, it has problems with complex C++ classes. Consider a methods of an object that resides in a mapped region of a chunked memory mapped file as shown in Listing 3.1.

Listing 3.1: Method of a C++ object in external memory.

```
1 void ExtClass::func() {
2   for (size_t i = 0; i < this->size; ++i)
3     this->largeExternalArray[i] = 7;
4 }
5
6 // calling code
7 ExtPtr<ExtClass> ptr = ...;
8 ptr->func();
```

Problems

When `ptr->func()` is called, the memory mapped file manager will ensure that the object pointed to by `ptr` will be mapped and available for read/write. Assuming that the call to `func()` will do a lot of allocations, deallocations or just memory access to external memory, the LRU mapping/unmapping logic might decide that the memory location pointed to by `ptr` is not needed any more and will unmap that region. Therefore, the `this`-pointer might become invalid during execution of `func()` and the dereferencing of `this->largeExternalArray[i]` or `this->size` will become an invalid memory access, yielding a segmentation fault or other undefined behavior. The main reason for this is that the `this`-pointer is not an `ExtPtr`, but a conventional C-pointer, which doesn't ensure that the pointee is available.
To solve this problem, an additional *keep-flag* was introduced for memory chunks that are managed by the external memory manager. This flag will modify the LRU behavior such that chunks that have this keep-flag set will not be evicted/unmapped, but kept available. To set such a flag, C++ has a concept called *proxy classes* as shown in Listing 3.2.

Listing 3.2: C++ Proxy class. ExtPtr<T>'s operator-> will not return the pointee of type T but a PtrProxy<T> object. This temporary PtrProxy object will set the keep flag in the constructor and release it in the destructor. The object is destructed after the evaluation of the dereference operation including executions of long running methods.

```
1  template <typename T>
2  class PtrProxy<T> {
3  public:
4    PtrProxy(T* p, uint64_t fileOffset) : _p(p), _offset(fileOffset) {
5      setKeepFlag(_offset);
6    }
7    T*       operator->()       { return _p; }
8    const T* operator->() const { return _p; }
9    operator T&() { return *_p; }
10   ~PtrProxy() {
11     releaseKeepFlag(_offset);
12   }
13 private:
14   T* _p;
15   uint64_t _offset;
16 };
```

The same problem exists for C++ references: For some features of C++, an unmanaged C++ reference is strictly required, such as for copy constructors. For other features they are just convenient to use. However, they suffer from the same problem as unmanaged pointers: code that does a lot of memory access tends to invalidate them. Sadly, until today there is not much that can be done about this automatically, as there is no proxy-class-like feature for references in C++[6]. For now, this problem is solved by some explicit code to set the keep-flag when required, utilizing RAII[7] to make the process as simple as possible, as shown in Listing 3.3.

Listing 3.3: RAII-style KeepFlagger class that sets the keep-flag in its constructor and releases it in the destructor to make sure that *ref* stays valid.

```
1 void foo() {
2   KeepFlagger<T> flagger(ptr);
3   T& ref = *ptr;
```

[6]However, there is a proposal for C++17 to allow overloading the .-operator, which would make reference-proxies possible and allow for setting the keep-flag automatically in a similar way as for pointers[SDR14].

[7]Resource Acquisition Is Initialization

```
4   // do some extensive work with ref without making it invalid
5 }
```

3.2.2 Managed memory mapped files

The *C++ standard template library* (STL) already provides a concept called *allocators*. Allocators are classes that handle memory allocation. By default, these allocators are simple wrappers around *new* and *delete* (which, in their simplest form, are just *malloc()* and *free()* plus a constructor/destructor call). While this default allocator works fine for most needs, it might sometimes be useful to write a custom allocator for performance tuning. E.g. a simple *memory pool* allocator could be used to speed up allocation of many small objects by reducing the amount of system calls to only one initial allocation of the pool itself.
Managed memory mapped files is the concept of using an allocator to manage/allocate memory inside a mapped region so that external memory can simply be used with all standard container classes. By simply writing

```
vector<int, MemoryMappedAllocator> v;
```

one could have a vector that stores its data on disk. To enable this, an allocator in its simplest form needs to look like the code provided in Listing 3.4.

Listing 3.4: Allocator concept

```
1 template <typename T>
2 class MyAllocator {
3 public:
4   typedef T value_type;
5   // to convert allocators
6   template <typename U> MyAllocator(const MyAllocator<U>& other);
7   T* allocate(std::size_t n);
8   void deallocate(T* p, std::size_t n);
9   template <typename U>
10  bool operator==(const MyAllocator<U>& other);
11  template <typename U>
12  bool operator!=(const MyAllocatur<U>& other);
13 };
```

However, for an allocator for chunked memory mapped files this is not enough: The allocator in Listing 3.4 uses `T*` as a pointer-type and `std::size_t` as a size-type, but

C++ pointers `T*` might become invalid at any point, as described in subsection 3.2.1 and the type `std::size_t` is usually equal to `uint32_t` on 32-bit systems, making large allocations impossible. To fix this, the allocator can be extended with some typedefs to cope with these problems:

Listing 3.5: Allocator with custom pointer and size types

```
1  template <typename T>
2  class MyAllocator {
3  public:
4    typedef ExtPtr<T>       pointer;
5    typedef ExtPtr<const T> const_pointer;
6    typedef uint64_t        size_type;
7    typedef uint64_t        difference_type;
8    pointer allocate(size_type n);
9    void    deallocate(pointer p, size_type n);
10   // ... rest is same as before
11 };
```

Sadly however, support for these advanced allocators is not consistent among C++ standard library implementations. In fact, Scott Meyers mentions in his *Effective STL* book that standard libraries are allowed to assume that an allocator's pointer typedef is a synonym for T*[Gaz15]. Therefore, custom *vector*, *list*, *map*, ... classes, which respect these extended allocators, need to be implemented when needed. A very popular library implementing this is the Boost.Interprocess library, which offers several popular allocators with different allocation strategies that manage a memory mapped file's contents. Additionally, Boost.Interprocess provides implementations of many STL containers which respect the `pointer`, and `size_type` typedefs. However, since Boost.Interprocess doesn't know **chunked** memory mapped files, a custom implementation of an allocator and several container classes was implemented for this thesis.

Stateful vs. stateless allocators

Allocators can be divided into two categories: Either they need state information or they don't. The C++ standard allocator is a stateless allocator: `new` and `delete` are globally defined and there is only one memory pool to allocate from. Intuitively, a memory mapped file allocator could also be a stateless allocator, if only one globally

defined memory mapped file, which is used for allocations and deallocations, is allowed. However, this isn't very practical, as processes might want to have many different files open as external memory at the same time. Hence, Boost.Interprocess uses stateful allocators: they have a constructor that takes information about the memory mapping that they are supposed to manage and store that information as their state. Therefore, when one wants to use a stateful allocator with a container, the allocator needs to be created, assigned its state and then passed to the object, as seen in Listing 3.6.

Listing 3.6: Usage of stateful vs. stateless allocators

```
1 // Usage of a stateless allocator: The container only receives the
      type and can create the allocator by itself:
2 vector<int, MyStatelessAllocator<int>> myVector;
3
4 // Usage of a stateful allocator: creation, state assignment and
      passing to the container:
5 MyStatefulAllocator<int> allocator;
6 allocator.setMemoryMappedFile(theFile);
7 vector<int, MyStatefulAllocator> myVector(allocator);
```

In addition to the increased code amount for creation, stateful allocators need to store their state internally, which requires memory and therefore increases the size of the vector object from `sizeof(Allocator::pointer)+ sizeof(Allocator::size_type)` to `sizeof(Allocator::pointer)+ sizeof(Allocator::size_type)+ sizeof(Allocator)`. For my implementation I have chosen a compromise between both worlds: A stateless allocator that does not require any state variables, but **encodes its state in its type**, as shown in Listing 3.7.

Listing 3.7: A stateless allocator that encodes its state into its type

```
1 template <typename T, ExternalMemoryManager* memoryManager>
2 class ExternalMemoryAllocator {
3   // ... use "memoryManager->..." as state that is encoded in the
        type/template arguments and requires no space
4 };
5
6 // Usage:
7 ExternalMemoryManager memoryManager; // this must be an externally
      linkable symbol and therefore a global variable
8 int main() {
```

```
9   memoryManager.openFile("/home/.....");
10  vector<int, ExternalMemoryAllocator<int, &memoryManager>>();
11  // ...
12 }
```

The advantage of this method is that the allocator requires no storage space and is therefore stateless. This allows us to not only store the vector's data on external memory, but the complete vector object itself. A clear disadvantage is that the ExternalMemoryManager object needs to be available during compile time and must be an externally linkable symbol (i.e. a global variable). This makes the approach less flexible. Especially, it is impossible to create an *arbitrary* amount of ExternalMemoryManager objects that is not known at compile time.

Pointers in external memory

For now, only pointers from main memory into external memory have been discussed, which can be easily constructed with the previously mentioned allocators:

```
1 class IntList {
2   int value;
3   IntList* next;
4 };
5 ExtPtr<IntList> ptr = ExternalMemoryAllocator<IntList, &
      memoryManager>().allocate(1);
```

However, the *next* pointer is dangerous: If it is a pointer into main memory, it makes no sense to store it: When reading the data in the next run of the program, the *next*-pointer will most likely not be valid any more, as memory locations might have changed arbitrarily. If it is a pointer into external memory, it is even more dangerous: When the target memory location is unmapped (either manually or by LRU eviction of the chunk), the pointer becomes invalid. Hence, a raw pointer must not be stored, but an ExtPtr as provided by an `ExternalMemoryAllocator::pointer` typedef. From the compiler's view, this can simply be done, as ExtPtr is a POD data type (it only contains an `uint64_t` offset). But when dereferencing this ExtPtr, how does it know into which **file** (in the form of an ExternalMemoryManager) this offset points? Similarly to the allocators, this information can be encoded in the type itself:

```
1 template <typename T, ExternalMemoryManager* memoryManager>
2 class ExtPtr {
3   // ...
4 };
```

This, on the one hand, saves storage space, and on the other hand, is actually required to store ExtPtrs inside external memory: Assume that ExtPtr contained a member of type ExternalMemoryManager* instead of a template argument. When this ExtPtr is stored to disk and then re-read in another program run, this pointer would be invalidated.

Generally: When pointers into external memory should be stored, at some point a *bridge* between main memory and external memory needs to exist: Dereferencing a pointer needs to access the ExternalMemoryManager, which can only reside in main memory as it contains runtime information (i.e. a file handle and information about mapping and the LRU cache). For my implementation, this bridge is completely encoded into the C++ type system, with the advantage that it doesn't require any additional storage space on a per-object basis and no additional logic that tells external objects the file they need to use to dereference nested pointers, at the disadvantage that all ExternalMemoryManager instances need to be known at compile-time.
This leaves us with a final storage engine design as shown in Figure 3.4.

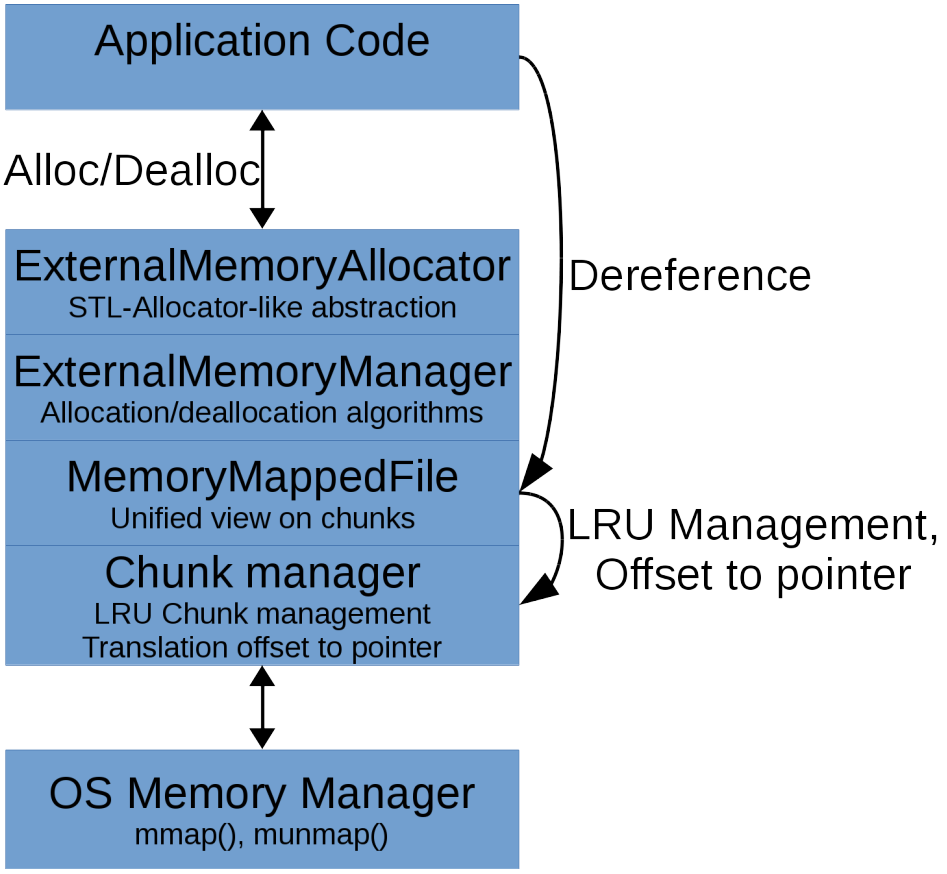

Figure 3.4: Storage engine design: application/container classes use the allocator for memory management. Dereferencing of a pointer will be forwarded to the MemoryMappedFile class, which will use the chunk manager to ensure that the required offset is mapped into memory and translate it to a real C++ pointer.

4 Main Memory vs. External Memory

While a nice abstraction of disk I/O to make it almost as convenient to use as main memory has been implemented, the fact that these are two different physical hardware components with different properties can't be ignored. Especially the fact that disks are far slower in sequential access, and even more though in random access patterns, needs special attention. A quick benchmark on a conventional desktop machine, to either an ext4 formatted drive or a tmpfs ramdisk[1], yielded the following results[2]:

Access pattern	Block size	Main memory	SSD	HDD
Sequential Read	1 MiB	10174 MiB/s	351 MiB/s	128 MiB/s
Sequential Read	64 KiB	9688 MiB/s	343 MiB/s	128 MiB/s
Random Read	1 MiB	9388 MiB/s	317 MiB/s	62 MiB/s
Random Read	64 KiB	7193 MiB/s	119 MiB/s	7,4 MiB/s
Sequential Write	1 MiB	11163 MiB/s	223 MiB/s	108 MiB/s
Sequential Write	64 KiB	11093 MiB/s	221 MiB/s	112 MiB/s
Random Write	1 MiB	9850 MiB/s	227 MiB/s	33 MiB/s
Random Write	64 KiB	8201 MiB/s	119 MiB/s	3,5 MiB/s

Note that these results are far from scientific, as many factors need to be considered, such as the data that is written (some SSDs may do on the fly compression to reduce actual I/O), block sizes, file system, caches, and other side effects, but they give a basic idea about relative performance of different physical storage options: While main memory is usually at about 9-10 GiB/s, SSDs are somewhere around 200-400

[1] A virtual, volatile filesystem for Linux that makes it possible to mount parts of main memory into the file system tree and store files to it

[2] For measuring performance, a small script was created that reads/writes 1 MiB and 64 KiB blocks sequentially or randomly from a large file. I/O caches were cleared between runs.

MiB/s and conventional spinning disks can be slower than 100 MiB/s for 1 MiB block size. For smaller block sizes (and hence, more random jumps to read/write the same amount of data), the difference becomes even larger. An important observation is, that while sequential vs. random access patterns don't seem to matter too much for graphs that are stored in main memory, the difference is quite large for hard disks. So in order for external memory to work efficiently, access patterns of the algorithms need to be optimized to make access *as sequential as possible*. Since the access pattern of e.g. Dijkstra's Algorithm can not really be changed a lot, a different approach is taken to minimize random access to the graph: Potentially decreasing the size of the graph to reduce the amount of accesses in general, as well as reordering nodes and edges of the graph in a way that fits the implemented algorithms.

4.1 Graph Simplification

4.1.1 Contraction of simple nodes

Since a navigation system is about more than just route planning, one should not only care about the weighted graph of a road network, but also about geometric properties, for example to make rendering of a map or routes possible. Depending on the source data, this geometric information might be encoded into the graph itself[3]: Each node in the graph has a geographic coordinate and the edges resemble the road between these two coordinates. An edge can have different properties that may be interesting for the user or the graph weighting, such as speed limits, road types or road name. However, since the properties of a road may often stay the same for a long road and only the coordinates change, this representation is quite inefficient: For each geometric road segment, all road properties need to be stored, even though they stay the same. And additionally, the algorithms need to traverse all these road segments for no particular reason: The algorithms only needs the weight of a road, which could easily be computed as the sum of the weight of all road segments. They do not care about geometric properties at all[4]. Hence, the graph can easily be simplified by short-cutting nodes that have the sole purpose of

[3] E.g. for OpenStreetMap, this is the case.

[4] Exception: if the geometry is part of the weighting, such as a weighting that e.g. prefers curvy roads for motorcyclists.

representing geometric information, as shown in Figure 4.1.

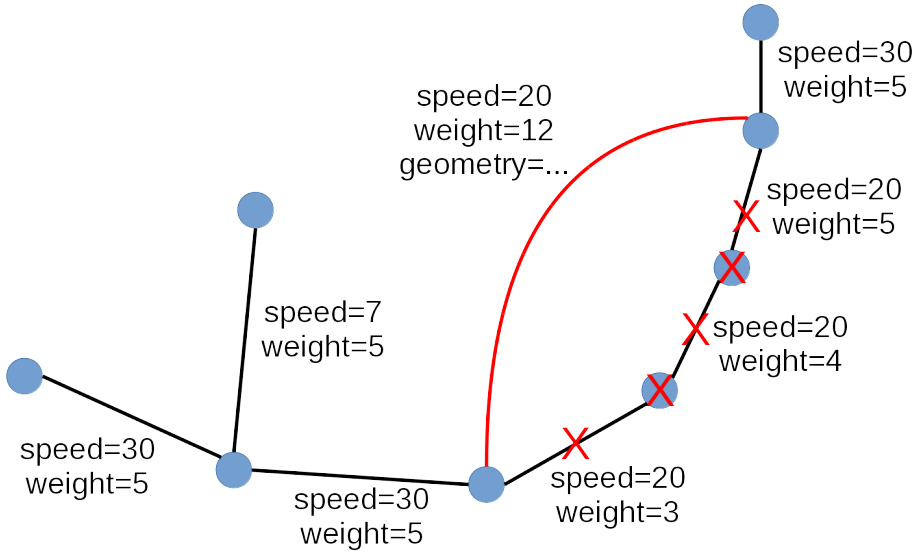

Figure 4.1: Graph simplification by removing unneeded edges: Only the red edge could be added as a short cut, the two nodes between it and the respective edges can be removed and instead added as geometry to the shortcut-edge. All other nodes serve a non-geometric purpose, as they either change the road properties (maxspeed) or serve as the base of a junction or dead-end road.

4.1.2 Removal of small subnetworks

In real world datasets, one can often observe very small unconnected subnetworks that only consist of a fairly small number of nodes. Especially at the cutting borders of the dataset, these are fairly common, as shown in figure Figure 4.2. These

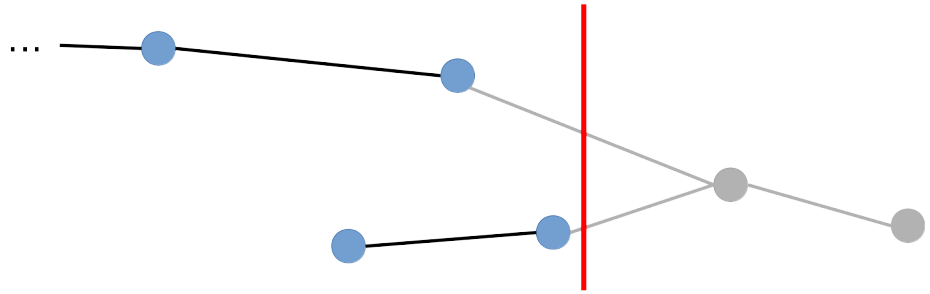

Figure 4.2: The red line is the cutting border of the dataset, leaving a small unconnected subnetwork in the lower part of the image, that was previously connected to the main graph.

subnetworks are usually undesired: If routing starts at one of those, the result would

be a *route not found* error. If small subnetworks are removed, routing would start at the next best road that is connected to the larger road network, which is usually more desirable. Additionally, this slightly reduces the size of the graph and therefore improves locality during routing.

4.2 Graph Sorting

The most common graph representations in memory are *adjacency lists* and *adjacency matrices*. An adjacency list stores for each node its neighbors and is therefore well suited for graph traversal starting at one node. Adjacency matrices use a matrix of size $nodeCount \times nodeCount$ and whenever two nodes i, j are connected, the edge information is stored at the respective matrix element. Since road networks are very sparsely connected, matrices are a fairly inefficient storage method as they require quadratic space and searching all neighbors of a node takes linear time w.r.t. to the amount of nodes in the graph. Hence, for road networks usually adjacency lists are used. A basic adjacency list can be written as `vector<vector<size_t>>`, so that an access to element $[i][j]$ returns the jth neighbor of node i. In my implementation, the `size_t` stored in the list is an edge id - an index into a separate `vector<Edge>`. As Dijkstra's Algorithm (and all its descendants) traverse the graph in a very specific way, their memory access patterns can be improved by sorting the nodes and edges in a way that fulfills that traversal order as good as possible.

4.2.1 Node Sorting

Several different node sorting orders have been implemented and tested for this thesis:

- *unordered*: Don't sort at all, use the arbitrary order as given from the dataset. Note that for many datasets this is not completely random. As the dataset is usually edited in batches that are relatively close to each other, these IDs have at least a small sense of locality in some cases.
- *BFS*: A breadth first search starting at an arbitrary node in the graph is used and nodes are stored in the order in which they are visited. As the graph might have multiple unconnected or weakly connected components, the BFS is run for each component independently.

- *Z-curve or Hilbert-curve order*: A space filling curve that preserves locality can be used to map the two dimensional coordinates into a one-dimensional key/coordinate. Nodes can then be sorted by this key. This approach can be viewed as *sorting by spacial proximity*. For this thesis, Z-curve and the Hilbert-curve sorting has been implemented. While the Hilbert-curve provides better locality, the Z-curve is slightly faster to compute and simpler to implement. The Z-curve is mainly known from a concept called *geohash*.
- *Hierarchy level*: For contraction hierarchies, the graph is always traversed in an upwards direction w.r.t. the node order. So nodes can be sorted by their hierarchy level.

While *BFS* and *Hierarchy Level* are graph-specific, Z-Order and Hilbert-Order are based on geographic properties (node location). A comparison of the two can be seen in Figure 4.3.

Figure 4.3: Hilbert-Order on the left side vs. Z-Order on the right side. The line between nodes shows their order. I.e. node 1 is connected to node 2, which is connected to node 3, ... Hilbert-Order seems to have fewer big jumps, suggesting better locality.

4.2.2 Edge Sorting

As a large array is used to store all edges of the graph, these can also be sorted in a way that suits the algorithms. For my implementation, the following orders have been tested for performance:

- *unordered*: Edges are taken as they are presented in the original dataset.

- *BFS*: The graph is traversed in BFS order, starting at a random node and Edges are sorted in the order they are visited by the BFS.
- *Source node*: As the nodes have already been sorted in a useful way, this information can be reused to sort edges by their source node.

5 Evaluation

5.1 Performance

This chapter will present an overview of the performance of the algorithms implemented for this thesis. For benchmarks, the following hardware specifications have been used where applicable:

- *performant-mem*: A Desktop PC running Ubuntu Linux 16.04 with an Intel I7 4790K CPU clocked at 4.00 Ghz and 16 GiB of DDR3 memory, with main memory used as a graph storage,
- *performant-ssd*: The same machine as *performant-mem*, but using a Crucial C300 SSD for graph storage,
- *performant-hdd*: The same machine as *performant-mem*, but with a 7200 RPM harddisk used for graph storage,
- *lowend-mem*: A rather old netbook with an Intel Atom N270 CPU clocked at 1.6 Ghz and 1 GiB of memory, using main memory for graph storage,
- *lowend-hdd*: The same as *lowend-mem*, but with a conventional 1.8“ HDD with 5400 RPM used for graph storage.

For the road network, the *Germany* extract from OpenStreetMap as provided by Geofabrik[1] on March 29, 2016 was used, which, after parsing and extracting the road network, contained 23,761,387 nodes and 48,100,686 edges. As this road network takes quite a while to parse and process, a smaller extract was used for parameter and algorithm tuning: the *Baden-Württemberg* extract with 3,531,003 nodes and 7,162,210 edges. Later, this tuning was applied to the larger network. In an unoptimized graph, even this rather smaller network was too large to work well on external memory on slow hardware, as can be seen in section 5.2.

[1] `http://download.geofabrik.de/europe.html`

5.2 Baseline Performance

As a baseline, the performance of the implemented algorithms on the road network as provided by Geofabrik/OpenStreetMap without any optimizations for both datasets and all applicable hardware configurations was measured. Each bar represents the average runtime over 100 random routes. File system caches were cleared after each batch. I.e. computing the 100 routes started out with empty caches, but the second route used the caches initialized by the first route, the third route used the caches as initialized by the first two routes, and so on. Computations that took more than 10 seconds per route were considered unfeasible and results have been omitted in the diagram below to make it more readable. The results can be seen in Figure 5.1 and Figure 5.2.

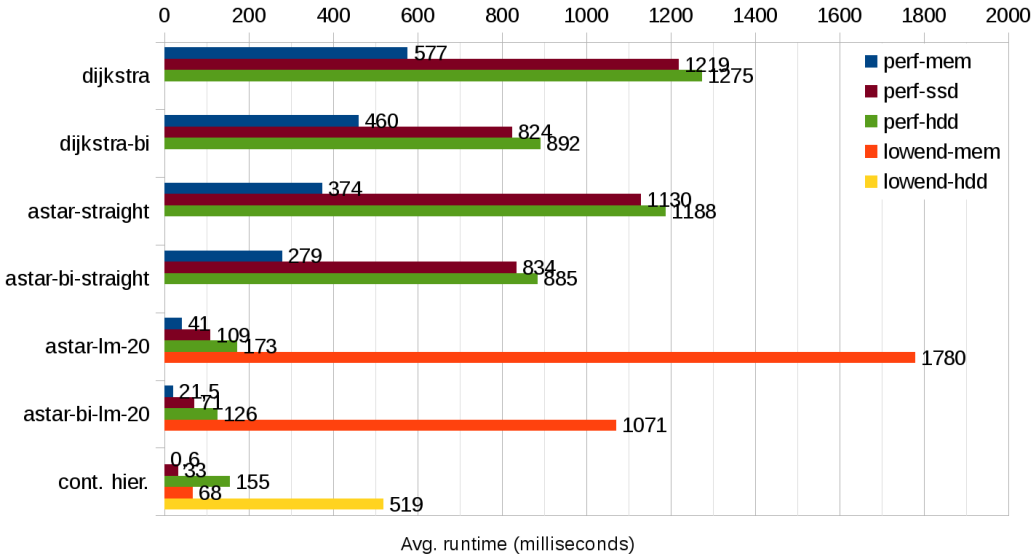

Figure 5.1: Performance for the *Baden-Württemberg* dataset without any graph optimizations applied. The *lowend-mem* benchmark only yielded feasible results for landmarks and contraction hierarchies, while the *lowend-hdd* configuration only worked reasonably for contraction hierarchies

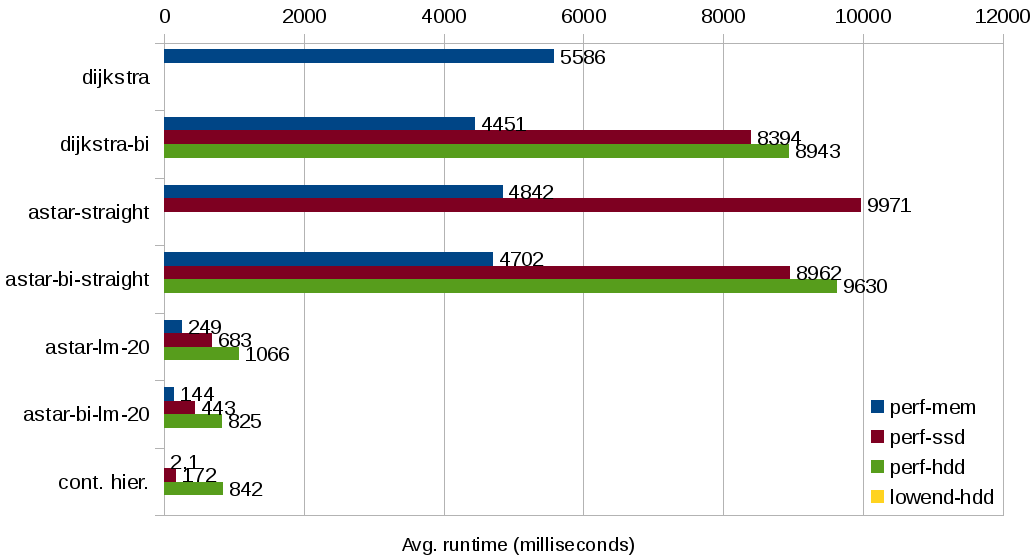

Figure 5.2: For *Germany*, the lowend machine did not yield any feasible results. Note however, that only *lowend-hdd* was tested, because the graph did not fit into main memory. Again, results of more than 10 seconds have been skipped to improve readability.

5.3 Node Sorting

While this baseline is at least somewhat usable on a fast PC, it is absolutely unacceptable for large graphs on mobile devices. Therefore, the optimizations described in chapter 4 have been implemented and tested for their performance gain. Some of these optimization always make sense to speed up routing, such as the contraction of simple nodes and the removal of small subnetworks (for the present tests, all unconnected subnetworks with less than 200 nodes **after** contraction have been removed). For all subsequent tests, consider these optimizations to be applied. After optimizing, the resulting graph for *Baden-Württemberg* consisted of 1,020,517 nodes and 2,097,927 edges and *Germany* consisted of 7,378,316 nodes and 15,155,656 edges. For the other optimizations - node and edge sorting - there are quite a few possible combinations: Nodes can be unsorted, sorted by BFS, Z-Order locality, Hilbert locality or, in case of contraction hierarchies, by their level in the hierarchy. Edges can be unsorted, sorted by BFS or by their source node. This results in a total of 15 possible combinations for each algorithm on each hardware configuration, which makes a total of $15 \cdot 7 \cdot 5 = 525$ configurations *without* considering real-world scenar-

ios like *empty vs. filled caches*. While many of these combinations have been tested, only the most relevant ones are presented in Figure 5.3, Figure 5.4 and Figure 5.5.

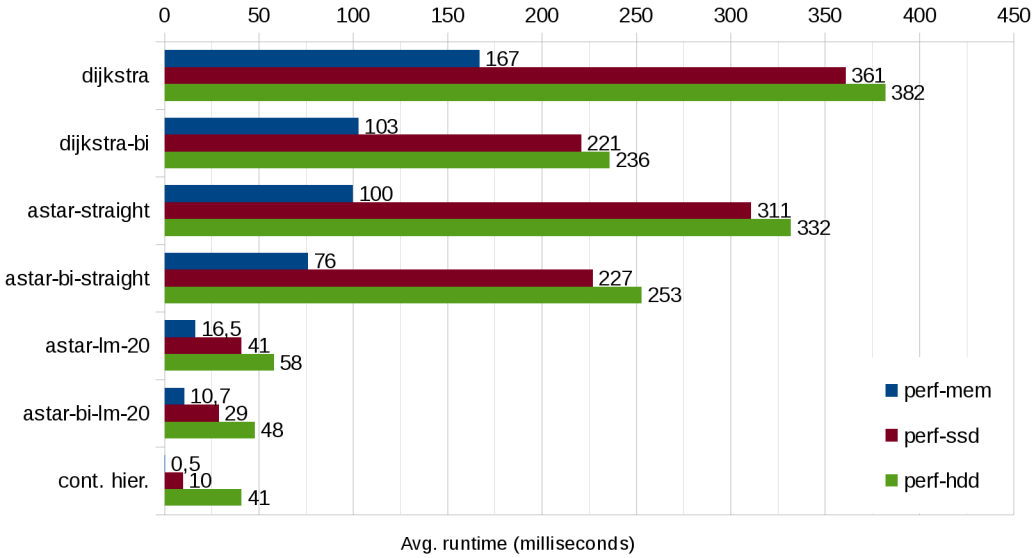

Figure 5.3: Performance on the *Baden-Württemberg* road network when contracting simple nodes and then ordering them in BFS order. Significant performance gains can be observed already with these two optimizations.

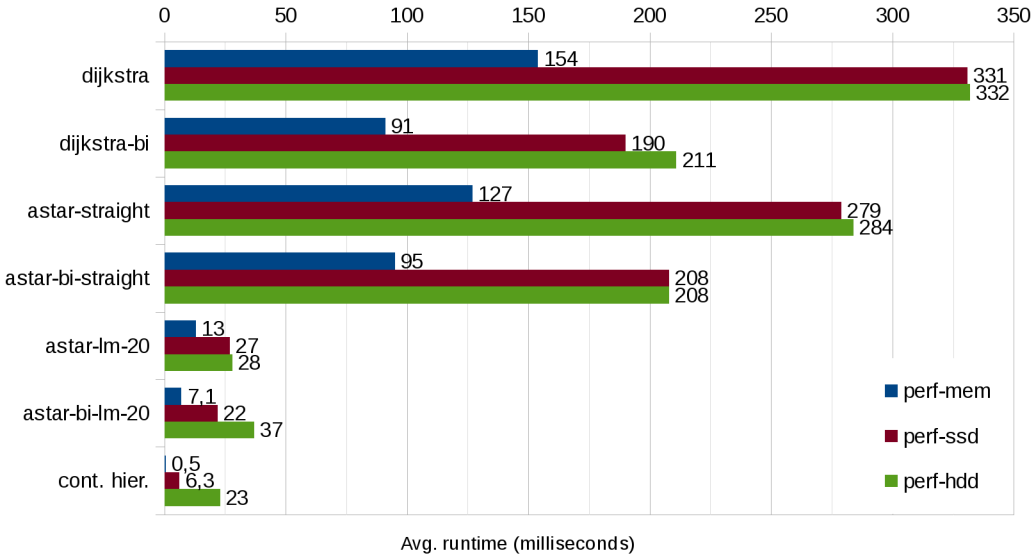

Figure 5.4: Instead of BFS order, nodes are ordered by their geohash (Z-curve order). This shows another small improvement over BFS. Most notably, the gap between SSD and HDD performance is smaller, which confirms that less random access is required.

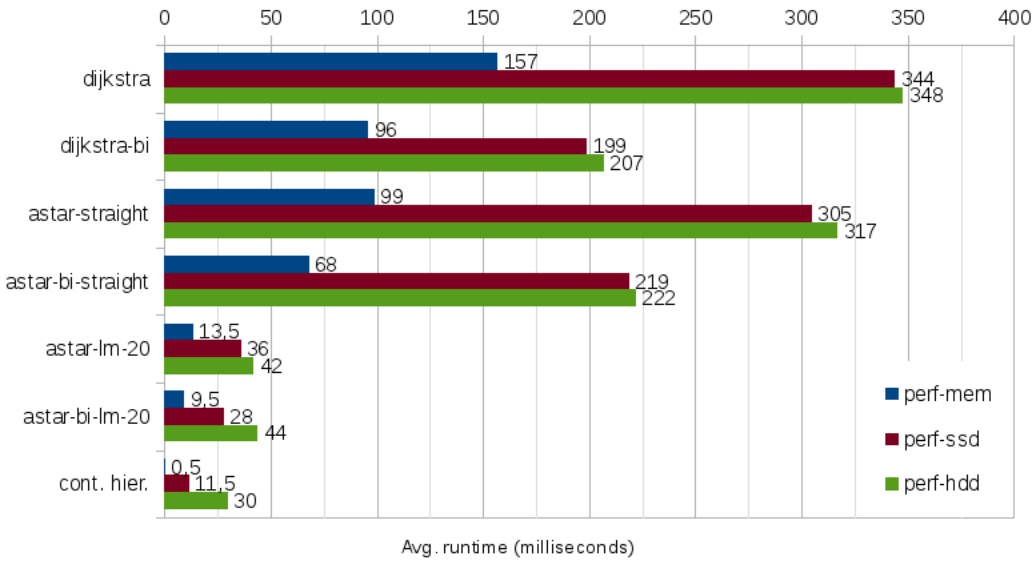

Figure 5.5: Even though the Hilbert-curve order yields better theoretical locality, routing is still generally slightly slower than with Z-curve order.

These benchmarks show that the Z-curve order seems to be the fastest with respect to node ordering.

5.4 Edge Sorting

Sorting the edges as well yields in another slight performance boost. However, it is not as big as for the node ordering, as shown in Figure 5.6 and Figure 5.7.

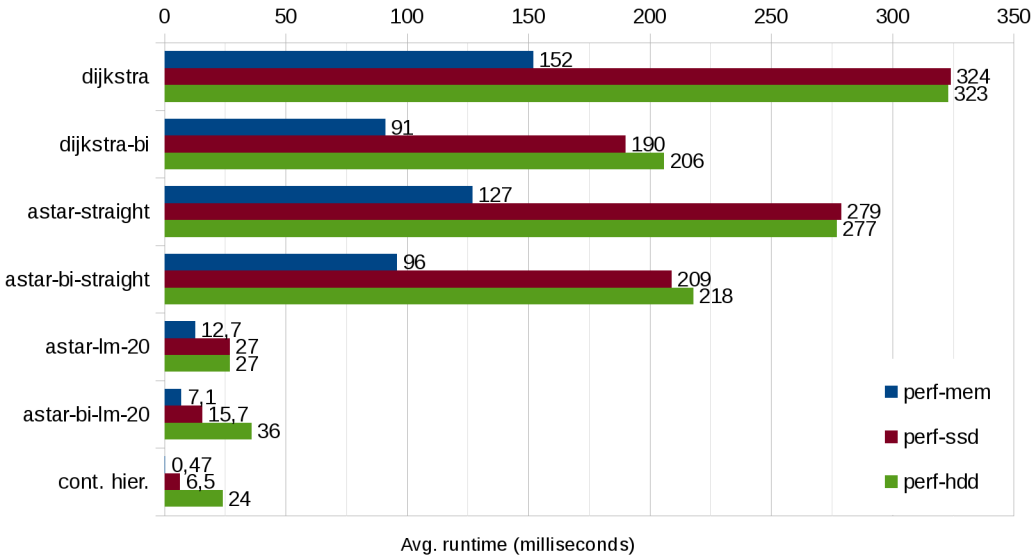

Figure 5.6: Z-Ordering for nodes, in conjunction with BFS ordering of the edges give another slight improvement. For unidirectional algorithms, HDD and SSD performance is very close, indicating decent locality.

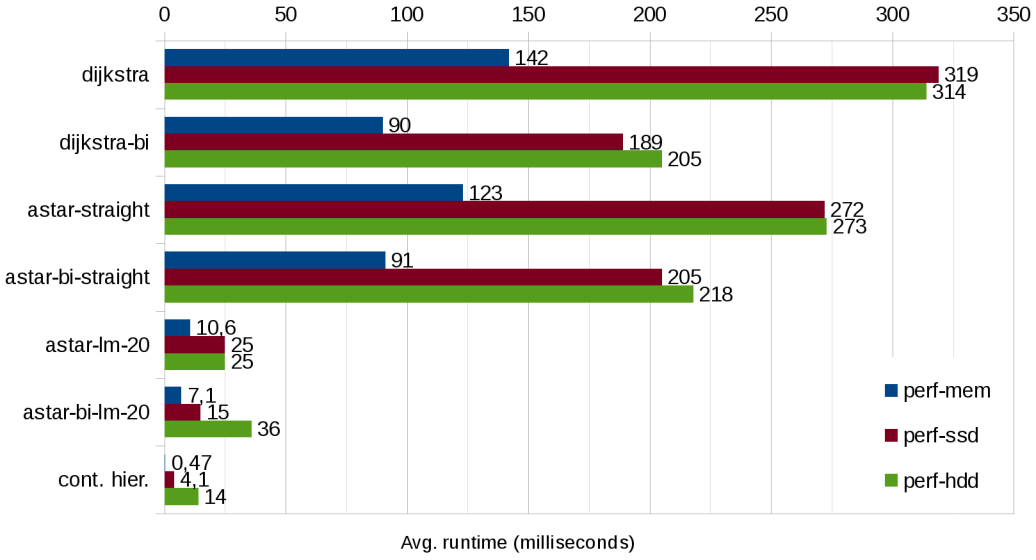

Figure 5.7: Even though the difference is quite small, sorting edges by source node is even faster.

For most algorithms, the Z-Order node sorting with source-node edge sorting yielded good performance. Therefore, they were applied to the whole *Germany* road-

network and retested on the *lowend* configuration. The final results can be seen in Figure 5.8.

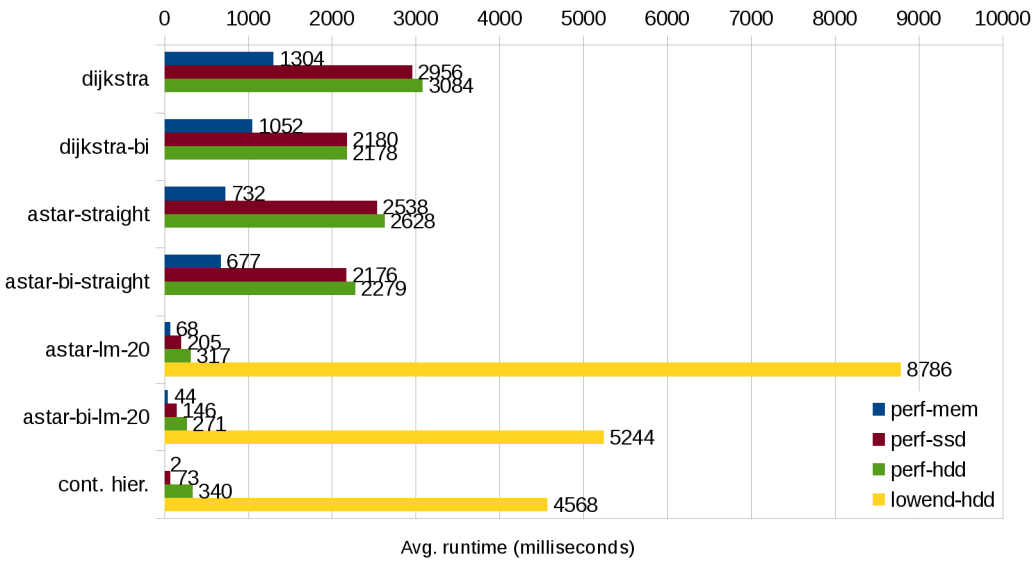

Figure 5.8: Final performance on all hardware configurations with Z-Order node sorting and source-node edge sorting on the *Germany* dataset.

These results, however, show a major flaw: contraction hierarchies visit far less nodes than bidirectional landmarks. Hence, they are much faster for an in-memory graph (2 ms vs. 44 ms). However, this advantage vanishes when looking at the secondary storage results: For the *lowend-hdd* configuration, contraction hierarchies were only slightly faster than bidirectional landmarks. For the *perf-hdd* configuration, they were even slower than landmarks. This suggests that the ordering is still suboptimal for contraction hierarchies, so that random access to the graph file will decrease performance. To overcome this, another set of benchmarks was done, which tested all sorting combinations for contraction hierarchies on the *perf-hdd* configuration. The results can be seen in Figure 5.9. This shows that sorting by contraction hierarchy level is much better when routing with contraction hierarchies. And sorting edges by BFS instead of their source node brings another small improvement. Using this ordering also improved performance on *lowend-hdd* drastically, resulting in a final runtime of **589 ms** on the slowest hardware configuration, instead of the previously measured 4568 ms with Z-Order sorting. In [SSV08], Sanders et. al. present another approach for sorting nodes in external memory in a contraction-hierarchies-friendly way, which is supposed to work even better: First, the nodes are sorted by their CH-

level and split into multiple groups. For each group, a modified depth first traversal is used, where a node v is not visited until all nodes u with an edge $(u, v) \in E'$ have been visited. Nodes are then stored in this order.

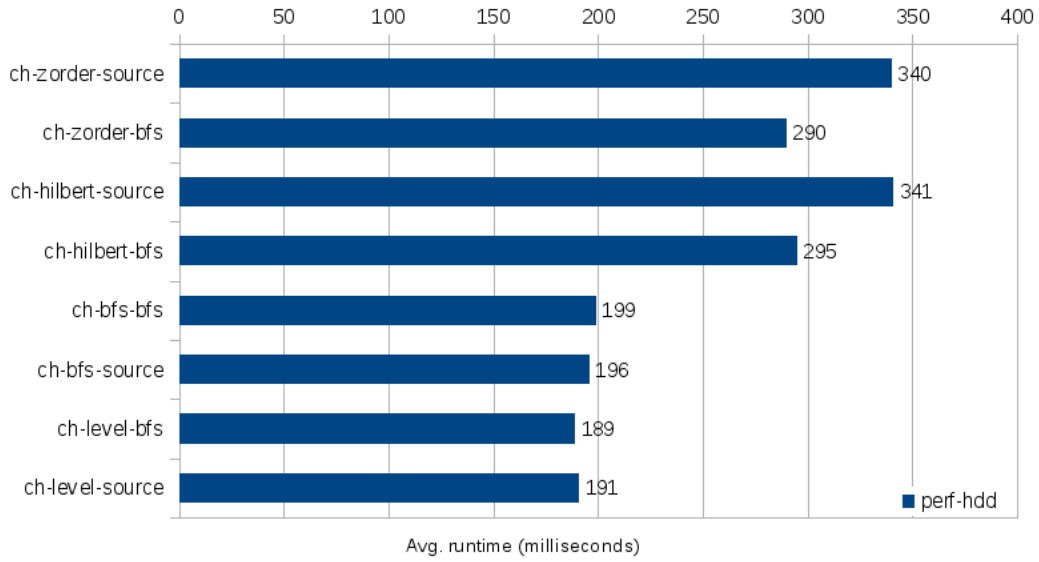

Figure 5.9: Comparison of all combinations of node/edge sorting for contraction hierarchies, showing that sorting nodes by node-order and sorting edges by BFS is much faster for CH.

6 Summary

In the previous chapters, an external memory/storage engine was developed that works

- *transparently* - the developer can write templated code that works in external memory or main memory by just using a different allocator,
- *simple and generic* - no special care needs to be taken for serialization - the only requirement is the use of *standard layout types*,
- *fast* - buffer copies are mitigated by using memory mapped files, an LRU cache and the operating systems memory manager to cache and preload as much data as possible,
- *cooperative* - by using the operating system manager, the resulting application uses as much RAM for caching as possible without limiting other processes.

While this storage engine allows storing arbitrary data - for navigation systems e.g. an R-Tree for reverse geocoding, compressed string storage for location names and search, ... is interesting, this thesis focused on the most performance-critical part: external memory graph storage for route planning. As the storage backend - e.g. a conventional harddisk - is significantly slower than a main-memory graph storage (see chapter 4), special care needs to be taken to circumvent the physical limitation. An important observation is that especially for physical harddisks, random access patterns can reduce performance a lot compared to serial access. Therefore, in section 4.1, methods were described how the graph can be *simplified* to reduce the potential search space of route planning algorithms in general. In section 4.2, several ways of sorting the graph's nodes and edges were proposed to make access to external memory more *local*, hence more performant.
While the simplification methods always make sense, the sorting methods have been tested for their performance benefit in section 5.1, with the result that sorting nodes by Z-order and sorting edges by their source node has shown the highest performance

gain for most algorithms. Only contraction hierarchies, which accesses the graph in a more wide-spread manner (see Figure 2.3), did not benefit from sorting by locality. Instead, sorting nodes by their level in the hierarchy and sorting edges with a breadth first search significantly outperformed the Z-order sorting, as shown in Figure 5.9. The result is an average query time of **589 ms** for random routes throughout Germany on a low-end laptop with only 1 GiB of RAM for **optimal** routes, which significantly outperforms many commercially available navigation systems.

7 Outlook

Besides the optimizations described in this thesis, there are other possible options: [SSV08] describes another method for sorting nodes and edges for contraction hierarchies, that is based on a combination of CH-level and a modified depth first search and might outperform the simple method described here, which sorts nodes only by hierarchy level. Also, it remains open why Z-order curve sorting results in better query performance as Hilbert curve sorting, even though [FR89] has shown that the Hilbert curve provides better locality properties - which I can intuitively confirm after looking at some of the resulting node orders of my test graphs.
Finally, this thesis builds the foundation of a project implementing a full-fledged navigation system toolkit. The route planning algorithms and external memory framework presented in this thesis provide a fast and generic basis to achieve this. However, there are still a lot of things missing. Most notably are

- data structures for reverse geocoding/gps to road segment lookup. This can be achieved with different spatial index structures, such as R-Tree/R*-Tree, Quad tree, K-d-tree, or even by utilizing the already computed Z-order or Hilbert sorting with a reduced key-length,
- data structures for forward geocoding to make it possible to search places by name,
- rendering of map-data,
- voice guidance,
- inclusion of real-time traffic data,

as well as all the small details that make up a good navigation system.

Bibliography

[BFSS07] Bast, Holger ; Funke, Stefan ; Sanders, Peter ; Schultes, Dominik: Fast routing in road networks with transit nodes. In: *Science* 316 (2007), Nr. 5824, S. 566–566

[DGPW11] Delling, Daniel ; Goldberg, Andrew V. ; Pajor, Thomas ; Werneck, Renato F.: Customizable route planning. In: *Experimental algorithms*. Springer, 2011, S. 376–387

[Dij59] Dijkstra, Edsger W.: A note on two problems in connexion with graphs. In: *Numerische Mathematik 1* (1959), S. 269–271

[DKS08] Dementiev, Roman ; Kettner, Lutz ; Sanders, Peter: STXXL: standard template library for XXL data sets. In: *Softw., Pract. Exper.* 38 (2008), Nr. 6, S. 589–637

[DSW14] Dibbelt, Julian ; Strasser, Ben ; Wagner, Dorothea: Customizable contraction hierarchies. In: *Experimental Algorithms*. Springer, 2014, S. 271–282

[FR89] Faloutsos, Christos ; Roseman, Shari: Fractals for secondary key retrieval. In: *Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems* ACM, 1989, S. 247–252

[Gaz15] Gaztanaga, Ion. *Allocators, containers and memory allocation algorithms*. 2015

[GH05] Goldberg, Andrew V. ; Harrelson, Chris: Computing the shortest path: A search meets graph theory. In: *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms* Society for Industrial and Applied Mathematics, 2005, S. 156–165

[GSSD08] Geisberger, Robert ; Sanders, Peter ; Schultes, Dominik ; Delling, Daniel: Contraction hierarchies: Faster and simpler hierar-

chical routing in road networks. In: *Experimental Algorithms*. Springer, 2008, S. 319–333

[GW05] Goldberg, Andrew V. ; Werneck, Renato Fonseca F.: Computing Point-to-Point Shortest Paths from External Memory. In: *ALENEX/ANALCO*, 2005, S. 26–40

[IHI+94] Ikeda, Takahiro ; Hsu, Min-Yao ; Imai, Hiroshi ; Nishimura, Shigeki ; Shimoura, Hiroshi ; Hashimoto, Takeo ; Tenmoku, Kenji ; Mitoh, Kunihiko: A fast algorithm for finding better routes by AI search techniques. In: *Vehicle Navigation and Information Systems Conference, 1994. Proceedings., 1994* IEEE, 1994, S. 291–296

[KMS06] Köhler, Ekkehard ; Möhring, Rolf H. ; Schilling, Heiko: Fast point-to-point shortest path computations with arc-flags. In: *9th Dimacs implementation challenge* (2006)

[PEH68] Peter E. Hart, Bertram R.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths. In: *IEEE Transactions on Systems Science and Cybernetics, Vol. SSC-4 No. 2* (1968), S. 100–108

[SDR14] Stroustrup, Bjarne ; Dos Reis, Gabriel: Operator Dot. (2014)

[SS05] Sanders, Peter ; Schultes, Dominik: Highway hierarchies hasten exact shortest path queries. In: *Algorithms–Esa 2005*. Springer, 2005, S. 568–579

[SSV08] Sanders, Peter ; Schultes, Dominik ; Vetter, Christian: Mobile route planning. In: *Algorithms-ESA 2008*. Springer, 2008, S. 732–743

[Zei13] Zeitz, Tim: Weak contraction hierarchies work. In: *Bachelor thesis, Karlsruhe Institute of Technology* (2013)