Institute of Formal Methods in Computer Science

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Integration of CH/HL-based Route Planning in OSCAR

Sokol Makolli

Course of Study:	Informatik
Examiner:	Prof. Dr. Stefan Funke
Supervisor:	Prof. Dr. Stefan Funke
Commenced:	May 19, 2020
Completed:	November 19, 2020

Abstract

Efficient and fast shortest path routing on road networks is required to meet the demands of millions of users every day. In this work we document an implementation, which combines two popular speed up techniques. The results show that with little space overhead a speed up of one order of magnitude to conventional techniques is achievable, while also giving the choice of using either disk or RAM storage. The routing application is also integrated into the search engine OSCAR, which then allows for interactive exploration of locations along the route.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

1 Introduction

The problem of finding the shortest path in road networks is required to be solved by millions of people on a daily basis.

The most famous algorithms to find shortest paths is Dijkstra's Algorithm [Dij+59]. Given a graph $G(V, E, d)$ such that V is a set of vertices, $E = \{(u, v)|u, v \in V\}$ is a set of edges and $d : E \rightarrow \mathbb{R}^+$ a function which maps a non-negative distance to each edge. Dijkstra's Algorithm finds the shortest paths from one node to all other nodes in $O(|V| + |E|)$ time [Gol08]. It is theoretically the most efficient algorithm, since it considers every node and edge just once. In the specific case of road network routing, the graphs are known in advance. For that reason routing applications usually consist of two steps: a pre-processing step and a query step. In the pre-processing step one tries to efficiently compute and store information about the graph, so that the query step, given the stored information, can perform as fast as possible.

Considering that, a program could try to pre-compute all shortest paths of a graph with Dijkstra's algorithm and then just perform a look up in the query step. Computing all shortest paths between all node pairs takes a lot of time and requires $|V|^2 \cdot M(p)$ of memory space, with $M(p)$ being the amount of space one path occupies. Even on small road network graphs, that is not feasible, since most graphs contain well above one million nodes. Other approaches are needed. Given that example it becomes evident, though, that there is a trade off between query speed and memory usage.

Many pre-processing/query style algorithms have been invented to minimize the memory usage while still greatly improving the query speed [DSSW09]. The speed up is commonly achieved by either reducing the dijkstra search space or via a look up scheme, which does not require graph traversal at all. This work uses a recently discovered method which combines two of those shortest path speed up techniques [Fun20].

It combines Contraction Hierarchies(CH) [GSSD08] with Hub Labels [DGSW14]. Contraction Hierarchies augment a graph with shortcuts so that at query time a dijkstra search can skip many edges. While only roughly doubling the space consumption of a graph, queries require 3 orders of magnitude less time.

Hub Labels(HL) pre-compute for each node a set of labels such that by only looking up the set for the source node and for the target node, one can derive the shortest path. By omitting graph traversal Hub Labels are around 2 to 3 orders of magnitude faster than Contraction Hierarchies, but require a substational amount of additional space. For the road network of Europe Hub Labels would require tens of TB of memory.

The combination of CH and HL enables the user to interpolate between space consumption and speed. The more speed is consumed the faster the queries will be. Even with little overhead in memory the implementation of this work allows for fast efficient continent scale routing in sub millisecond time.

Additionally, the result of this work allows users to query for items such as gas stations or rest stops along the route. That is achived with the help of OSCAR [BF15]. OSCAR is a search engine for open street map data. It structures all open street map items into so-called cells. This work pre-computes the cell ids for every edge so that at query time all cells along the route can be combined. By providing the cell ids, open street map items can be directly retrieved from the oscar data structures.

2 Basics

The most basic description of a road network is that there are points, each with a position, connected with each other by roads. It intuitively makes sense to describe such a system with the mathematical structure of a graph. For a Graph $G = (V, E)$, V is a set of vertices which represent the points on the map and E is a set of edges which represent the roads. It is sufficient for routing purposes to store the edges as a pair of nodes $E = \{(u, v)|u, v \in V\}$. Since one way roads exist, it is important to note that $(u, v) \neq (v, u)$.

The time required to traverse two connected points can be very different on road networks. So, it is also required to map the edges to some kind of distance unit. This can be the raw distance in meters but usually for the users the travel time is more important. The travel time for a edge is in practice calculated by taking the distance between the two connected points and dividing it by the maximum speed that is allowed on that road in m/s. To incorporate the travel time or distance into our graph definition, a cost function $c : E \to \mathbb{R}^+$ is added. It is important to note that we only allow non-negative cost values, because the main algorithm for routing used in this work, Dijkstra's Algorithm, is not always correct if edge costs are negative.

Usually a user also wants to display the found route and travel time onto a map of the world. To do that we also need to add a function which maps each note to a pair of coordinates $m : V \to (x, y)$ with $x, y \in \mathbb{R}$.

Adding those extra functions our definition of a graph can be seen in Definition 2.0.1. Given that definition we can draw a graph such as the one seen in Figure 2.1 with its textual representation in Listing 2.1.

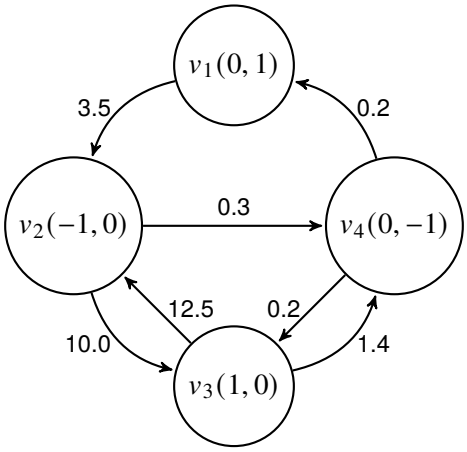
Definition 2.0.1 (Graph Formal Definition)

$$G = (V, E, c, m) \quad E = \{(u, v) \mid u, v \in V\} \quad c : V \to (x, y) \mid x, y \in \mathbb{R} \quad d : E \to \mathbb{R}^+$$

Listing 2.1 Textual representation of the example graph in figure 2.1

$V = \{v_1, v_2, v_3, v_4\}$
$E = \{(v_1, v_2), (v_2, v_3), (v_2, v_4), (v_3, v_2), (v_3, v_4), (v_4, v_1), (v_4, v_3)\}$
$c((v_1, v_2)) = 3.5, c((v_2, v_3)) = 10.0, c((v_2, v_4)) = 0.3, c((v_3, v_2)) = 12.5, c((v_3, v_4)) = 1.4, c((v_4, v_3)) = 0.2, c((v_4, v_1)) = 0.2$
$m(v_1) = (0, 1), m(v_2) = (-1, 0), m(v_3) = (1, 0), m(v_4) = (0, -1)$

Figure 2.1: Visual representation of a graph according to the definition 2.0.1



2.1 Efficient data structures for graphs

To perform routing algorithms on a graph like described previously we need to store that in the memory of a modern computer. Before we choose a data structure for our graph, we have to consider the size of the graphs and the operations we want to support. The structure should, while minimizing the memory consumption, allow for fast operations. These two requirements are usually in conflict with one another.

The most prominent operations required by Dijkstra's Algorithm is the traversal of all outgoing edges of a node. Sequential memory access is usually faster than jumping around and collecting all the data needed. So the outgoing edges of a node should be stored side by side in a continuous block of memory.

In practice, we can accomplish that with a vector E that stores all the edges sorted by each source node of the edge. Accompanying that we lay out an offset vector O which stores for each node where the position of the first outgoing edge in E is. To iterate through the outgoing edges of v_1 one now retrieves the offset for v_1 from O and iterates through E until the offset of v_2 is hit.

Note that currently we are not storing any information on nodes. For real world applications you need to store at least the coordinates of each node, so that one can support queries which go from one point on the globe to another. For that to work some kind of nearest neighbor search needs to happen to find the corresponding node to the given coordinates. In the implementation of this thesis the search happens with a grid structure.

Listing 2.2 Graph data structure in C++

```
#include <vector>

struct Edge{
  unsigned int source;
  unsigned int target;
  unsigned int cost;
}
struct Graph {
  std::vector<Edge> edges;
  std::vector<unsigned int> offset;

  // returns the outgoing edges for node with id nodeId
  std::vector<Edge> getOutgoingEdges(unsigned int nodeId) {
    std::vector<Edge> result;
    for(int i = offset[nodeId]; i < offset[nodeId + 1]; ++i) {
      result.push_back(edges[i]);
    }
    return result;
  }
}
```

2.2 Shortest path with Dijkstra's Algorithm

Since one can obviously construct a graph that has a shortest path in it which visits every node and every edge, the best algorithm you can find has the run time of $O(|V|, |E|)$. One algorithm which satisfies exactly that run time is Dijkstra's Algorithm [Dij+59]. It works on all graphs which have no negative edges. There are some optimizations you can make, but in Listing 2.3 you can see an implementation of Dijkstra's Algorithm in a C type language.

Listing 2.3 Dijkstra's Algorithm

```
int[] findShortestDistances(int source, Graph g) {
  int cost = int[numberOfNodes];
  for(int i = 0; i < numberOfEdges; ++i) {
    cost[i] = infinity;
  }
  cost[source] = 0;

  // use minheap as queue
  // the minheap always keeps the node at the top
  // which has the smallest cost in the cost array
  MinHeap minHeap;
  minHeap.push(source);

  while(!minHeap.empty()) {
    int currentNode = minHeap.pop();
    std::vector<Edge> edges = g.getOutgoingEdges(currentNode);
    for(int i = 0; i < edges.size(), ++i) {
      Edge currentEdge = edges[i];
      int target = currentEdge.target;
      if(cost[currentNode] + currentEdge.cost < cost[target]) {
        cost[target] = cost[currentNode] + currentEdge.cost;
        if(!minHeap.contains(target))
          minHeap.push(target);
        minHeap.reheap();
      }
    }
  }
  return cost;
}
```

2.3 Shortest path speedup schemes

Dijkstra's algorithm while being correct and optimal for any graph [Gol08], is still to slow for road networks. Performing a shortest path query with Dijkstra in decently sized graphs like for instance the german road network takes a few seconds. To speed up queries, we can use the fact that road networks do not change often. So we can collect auxiliary information about the graph in a pre-processing stage and use that information later to our advantage. Note that for everyday routing applications usually users only want to know the path from one point to another and not one to all.

Contraction Hierarchies (CH)[GSSD08] are such a preprocessing/query scheme, which speed up the query time on road networks significantly. Completing a shortest path query on the german road network only takes a few milliseconds while roughly doubling the amount of edges of the graph. That is a speedup in the order of 1000. The rough idea behind Contraction Hierarchies is that we augment a graph by shortcuts. At query time we perform a slightly modified Dijkstra so that we only traverse the constructed shortcuts instead of the actual paths. That reduces the amount of visited nodes during the search time.

Another speed up scheme are Hub Labels (HL)[DGSW14]. HL are a look up based technique. Instead of traversing a graph one only needs to perform look ups for precomputed labels for each node. Hub Labels closely relate to Contraction Hierarchies. A label of one node contains the path and the distance information of one graph traversal in the CH query for that node. So we can omit the graph traversal completely. With Hub Labels a one to one shortest path query on the german road network takes only a few microseconds. The caveat is that Hub Labels require a lot of memory. The memory usage is roughly 30 times higher then the that of the graph alone.

2.3.1 Contraction Hierarchies

Preprocessing Stage

Contraction Hierarchies are constructed from a graph given a hierarchical node ordering. The ordering metric assigns some kind of importance to each node. The importance is also called level. Then a hierarchy is constructed by contracting each node. Starting with the one of least importance. While removing that node v from the graph, it is important that the shortest paths for all the other nodes remain unchanged. That means that for each incoming edge <u,v> and each outgoing edge <v,w> a shortcut <u,v> with the cost of $c(< u, v >) + c(< v, w >)$ has to be added, if <u,v,w> is the only shortest path from <u,w>. To retrieve the original edges from a shortcut, each shortcut is stored together with pointers to the two edges it replaces. Given a graph $G(V, E, c)$ the computed shortcuts S and the predefined hierarchy l a new graph $G^*(V, E^*, c^*, l)$ is then constructed, such that $E^* = E \cup S$, $c^* : E^* \rightarrow \mathbb{R}^+$, with c^* incorporating all the original edge costs and the shortcut costs. [GSSD08]

Query Stage

To retrieve the shortest path from a source node s to a destination node t two modified Dijkstra queries 2.4 are required. One starts from s and operates on G^* the other one starts from t and operates on the reverse Graph of G^*_r. The reverse graph is constructed by reversing each edge $< u, v > \in E^*$ to $< v, u >$. The speed up of this scheme comes from reducing the search space in each of the modified dijkstras by incorporating the hierarchy that was previously constructed. Instead of relaxing each outgoing edge <u, v> of a node u, edges for which $l(u) > l(v)$ are ignored.

The result for each dijkstra is a set of distances for each node that was visited during the modified search. Those two sets are then merged while adding the distances. The smallest distance of that set is then also the shortest distance from s to t. The node v of the smallest distance represents the node in which the two paths meet.

The shortest path can be retrieved if additionally to the costs, the previously visited node is stored. Since the two paths both visit node v, one can backtrack and concatenate the path from v to s and from v to t. Note that the path from v to s needs to be reversed first. To get the corresponding path in the original Graph G each shortcut in the path has to be recursively unpacked.

The proof on why the query produces correct shortest paths would be out of the scope of this work. You can find it in the original paper by Geisberger et al. [GSSD08].

Listing 2.4 Dijkstra's Algorithm modified to make use of Contraction Hierarchies

```
int[] findShortestDistancesModified(int source, Graph g, int[] l) {
  int cost = int[numberOfNodes];
  for(int i = 0; i < numberOfEdges; ++i) {
    cost[i] = infinity;
  }
  cost[source] = 0;

  // use minheap as queue
  // the minheap always keeps the node at the top
  // which has the smallest cost in the cost array
  MinHeap minHeap;
  minHeap.push(source);

  while(!minHeap.empty()) {
    int currentNode = minHeap.pop();
    std::vector<Edge> edges = g.getOutgoingEdges(currentNode);
    for(int i = 0; i < edges.size(), ++i) {
      Edge currentEdge = edges[i];
      int target = currentEdge.target;

      if(l[currentNode] < l[target])
        continue;

      if(cost[currentNode] + currentEdge.cost < cost[target]) {
        cost[target] = cost[currentNode] + currentEdge.cost;
        if(!minHeap.contains(target))
          minHeap.push(target);
        minHeap.reheap();
      }
    }
  }
  return cost;
}
```

Implementation Details

The implementation of this work uses an open source CH-constructor [1]. It is important to note that this CH-constructor assigns the same level to multiple nodes. As long two neighboring nodes do not have the same level, which the constructor assures, the modified dijkstra query still works.

[1] https://github.com/chaot4/ch_constructor

2.3.2 Hub Labels

Hub Labels were first introduced by Delling et al. in 2014 [DGSW14]. In pre-processing for each node v two labels are computed: a forward label $L_f(v)$ and a backward label $L_b(v)$. The computed labels have to fulfill the cover property, which states that the set $L_f(s) \cap L_b(t)$ contains one so-called hub that is in the shortest path from s to t. If those labels are present, finding the shortest distance s-t becomes a matter of looking up the labels $L_f(s)$ and $L_b(t)$ and finding the hub h. The shortest distance is then the distance to h in $L_f(s)$ added to the distance to h in $L_b(t)$.

If we examine the contraction hierarchy query, we can see that the Dijkstra search spaces of s and t obey the hub label cover property. So one can calculate the hub labels by doing one forward and backward CH-search for each node and store the search results. That means that Hub labels speed up the ch query by doing the graph search in pre-processing. At query time there is no need to do a graph search anymore. Note that the search space of the CH-query does not only contain shortest paths. That means some values in the labels are not required. To minimize the amount of memory and the merge speed needed, those non-shortest distance values should be removed.

Hub labels speeds up the query time from a few milliseconds to a few microseconds in the example of the german road network. But while contraction hierarchies roughly double the edge count of a graph, hub labels require around 30 times as much memory as the graph itself. For the german road network graph that means that fully computed hub labels require around 175 GB of data. For an accurate road network of the whole planet over one TB of memory is required.

Implementation Details

Our implementation produces exactly the same labels as doing a CH-query for each node, but we do it from top to bottom. We start at the top of the hierarchy and calculate the labels of all nodes which have the same level at a time. The invariant is that, at the time we want to construct a label L(v) all the labels of nodes with higher level than v have already been computed. So for each outgoing edge <v, w> we look up the labels L(w) if the level of v is lower than the level of w and merge them into a new label, while also adding to each hub the cost for <v. w>. Since this scheme also does not only contain hubs with shortest distance, we do a self pruning scheme for each label after construction, by comparing the actual shortest distance to the node in the hub with the distance of the hub and removing the hub if necessary. To make use of multi-core processors we calculate the labels of one level in parallel.

3 Interpolating between CH and HL

Funke [Fun20] has proposed the idea that if Hub Labels are only computed for some nodes one can use these Hub Labels to then speed up the CH-query. The search space is reduced by stopping at nodes for which hub labels already exist. The more Hub Labels we compute, the more the search space is going to be reduced. Since we already have a kind of importance ordering given by the contraction hierarchy, finding out for which nodes to compute the Hub Labels for becomes quite trivial: we start with nodes of highest hierarchy level and stop at some level where we think that the query speed has increased enough. As for memory consumption, the hope is that if we use the contraction hierarchy, the search space is vastly reduced while only pre-processing a fraction of the Hub Labels.

In Table 3.1 the space consumption is listed in relation to the levels from 0 to 14 (out of 341). Level 0 in the table means that the Hub Labels for all nodes have been computed. Level 1 means that all Hub Labels have been computed, except for the nodes with level 0, etc. For the first few levels the space consumption drops fast. Then the drop flattens out. We can see that just after a few levels it is feasible to store the Hub Labels in main memory even on older and lower specced systems. The goal is now to show that when omitting the Hub Labels for the first few levels, the speed up is still significant.

Table 3.1: Comparison of Space Consumption of Hub Labels calculated for different Levels. Graph: Germany(Nodes: 24608237, Edges: 90097053, Max. Level: 341).

Labels calculated until level	Space Consumption
0	$173,6GiB$
1	$96,1GiB$
2	$55,0GiB$
3	$33,6GiB$
4	$22,7GiB$
5	$17,3GiB$
6	$15,63GiB$
7	$10,4GiB$
8	$7,28GiB$
9	$5,3GiB$
10	$4,04GiB$
11	$3,19GiB$
12	$2,63GiB$
13	$2,23GiB$
14	$1,95GiB$

3.1 Pre-processing Stage

Pre-processing is just a slight deviation from the normal HL pre-processing. The CH pre-processing remains the same. Since our implementation operates on a given CH-graph, we will only focus on the HL construction.

The idea is that we go through the nodes from highest level to lowest. For every node v we construct a set of hubs $(n, c(n), p)$, with n being the node which is located at the hub, $c(n)$ being the cost of the path $v \rightarrow n$ and p being the node that is located right before n on the path $v \rightarrow n$. p is needed to reconstruct the path from the hub label set. For a node v we then calculate the hub labels as seen in algorithm 3.1. Then we go through the up-graph $G^{\uparrow}$ of the CH-graph G, with $G^{\uparrow} = \{(u, v) \in G \mid l(u) < l(v)\}$. So $G^{\uparrow}$ only contains edges which have their origin at a lower hierarchy node than their destination. When calculating the Hub Labels for a node n, we collect the Labels from all the neighbors in $G^{\uparrow}$ and merge them together into a new Label. The merge routine can be seen in algorithm 3.2.

For the contraction hierarchy query to work, we need to perform two graph-searches: one on the actual graph starting from the source and one on the backward graph starting from the target. And since at query time we want to omit the graph search, we also need to calculate the Hub Labels of the backward graph. The used algorithms are the same as for the forward hub labels.

Algorithm 3.1 Calculating hub labels for a node in pre-processing

```
procedure CalcHubLabels(node, HubLabelStore)
    L ← {(node, 0, ∞)}                          // We set the distance to the node itself to zero
    for all (u, v, cost) ∈ OutgoingEdges(node) do
        if level(u) < level(v) then
            M ← HubLabelStore(v)
            L ← Merge(L, M, node, cost)
        end if
    end for
    HubLabelStore(node)← L
end procedure
```

Algorithm 3.2 Merging Labels for pre-processing

```
procedure Merge(L, M, node, cost)
    i, j ← 0
    N ← ∅
    while i < |L| ∧ j < |M| do
        (n_i, c_i, p_i) ← L_i
        (n_j, c_j, p_j) ← M_j
        if n_i < n_j then
            if p_i < ∞ then
                N.insert(n_i, c_i + cost, p_i)
            else
                N.insert(n_i, c_i + cost, node)
            end if
            i ← i + 1
        else if n_i < n_j then
            analog to first if just with (n_j, c_j, p_j)
            j ← j + 1
        else if n_i = n_j then
            analog to first if but take the element with smaller c_x
            j ← j + 1
            i ← i + 1
        end if
    end while
    return N
end procedure
```

To visualize the results of the Hub Label construction, we have given an example graph 3.1, its up-graph 3.2 and its hub labels 3.3.

Figure 3.1: Example graph with shortcuts in blue and contraction order/level in green.

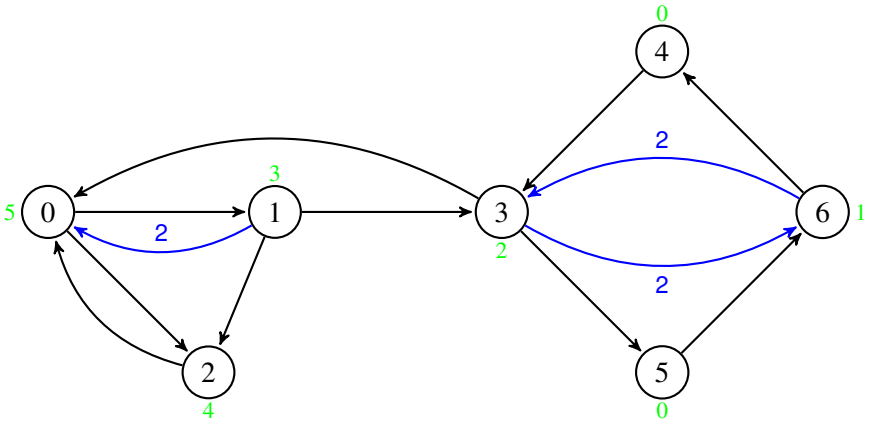
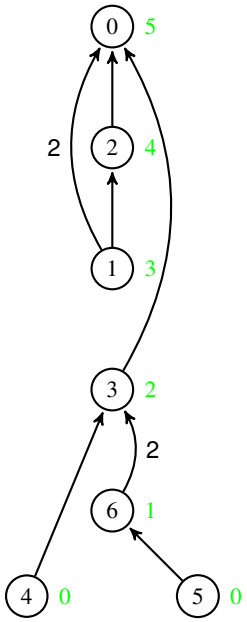


Figure 3.2: Example up-graph with contraction order/level in green.

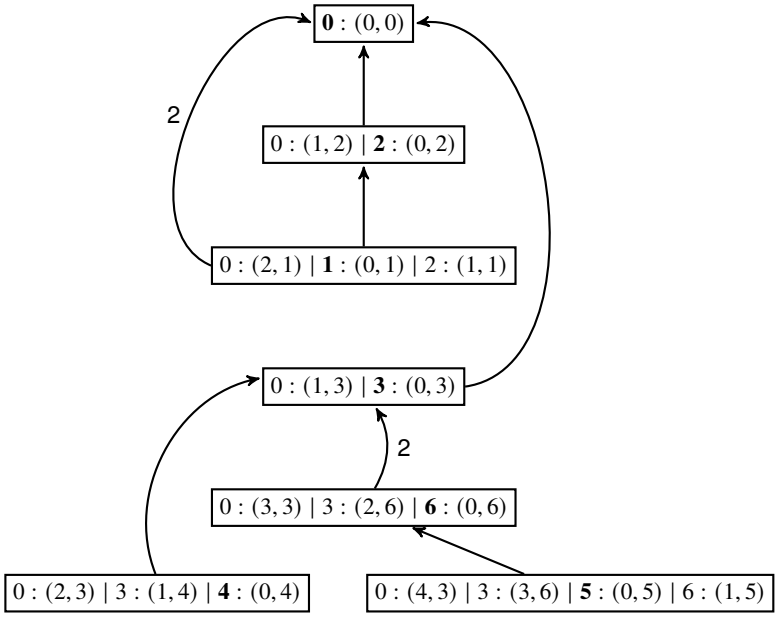


3.1.1 Implementation Details on Pre-processing

While we construct the Hub Labels, we go from highest level to level 0. The novelty of the interpolation approach is that we can stop at any level ℓ and still enable a query speed up. For the nodes with level $< \ell$, the Hub Labels will be generated at query time.

The used CH-construction contracts nodes of one independent set at a time and assigns all of the nodes contained in the set the same level. It starts at level 0 and tries to find a large independent set (usually containing around half of the nodes). The graph gets roughly halved by each level. That means that $1/2$ of nodes have level 0, $1/4$ of nodes have level 1, $1/8$ level 2 and so on. So if we only calculate the hublabels for the maximum level to level 1, we consume only half of the memory

Figure 3.3: Hub Labels (with corresponding nodes in bold) for up-graph 3.2.



space (roughly). To speed up the construction process on multi-core CPU's, the nodes of every level can be processed in parallel, because hub label construction only depends on the labels of nodes with higher level.

3.2 Query Stage

The query consists, just like the CH-query, of two different stages:

- graph traversal to generate the Hub Labels for the source and target node
- intersecting those two Hub Labels to find the common optimal hub

The goal of the graph traversal is to calculate the Hub Labels for a given node n. If the Hub Labels already exist due to pre-processing we just use those labels and we are done. If they do not exist, we perform a slightly modified ch-query, which makes use of the already generated Hub Labels. If, while performing the search, we encounter a node for which the Hub Labels have already been computed, we do not search on that path any further. We store that node in a temporary store. Now, the result of the graph search is a Label with distances to nodes we have met on the path and a store, which contains the nodes which we have not been investigated further. Keep in mind that the labels of that store have all been computed in pre-processing. To then generate a complete hub labels for n, we collect all the labels in the store and merge them with the incomplete label of the graph search. When we have completed the graph traversal for the source and target node, we just need to find the common hub with the lowest added cost.

3.2.1 Finding the shortest path

Until now, we have only calculated the shortest distance. But for our application, we also want to find the path corresponding to that distance. To do that we store for each label (query and in pre-processing) also the node that we visited previously in the path. Now given the source and target labels and the hub which lies on the shortest path, we can find the path from the source to the hub and from the hub to the target and merge those paths to get the path from source to target. The algorithm traces the path from the hub back to the source/target. Because of that the resulting path for the source node is in reverse order. The order for the target node is correct since the hub label creation was performed on the back graph.

The resulting path is the shortest path from source to target on the contraction hierarchy graph G^*. For that reason it might contain shortcuts. The used ch-implementation stores for each shortcut a pointer to the child nodes it replaces. To now get the path in the original graph G, we need to unpack each shortcut. The children of a shortcut might be shortcuts themself and every shortcut contains exactly two children. Given those properties, we can think of a shortcut as a binary tree, with the edges we need to unpack, as its root. The leafs of the tree are all original edges in G. Traversing the leafs from 'left' to 'right' yields the desired original path. We can achieve that with an in-order traversal, while only storing leafs. The traversal can be implemented recursively or iteratively using a stack. We used the iterative approach, since we wanted to avoid additional method calls for each recursion.

3.3 Storing and searching on disk

For this section it is important to note that the implementation is written in C++ and the most commonly used data structures are implemented in the standard template library(stl) of C++. The most prevalent data structure is a container called vector. The stl vector is stored in a continuous block of memory and can contain data structures of any kind. Its size does not need to bee known at compilation time and it can be resized dynamically at run-time. If, while adding a new element to the vector, the allocated space is exceeded, the vector doubles its allocation space to avoid a reallocation at each addition. Data access and input is of constant time. The stl even allows to store vectors inside of vectors. That is realised by storing only the pointers to each inner vectors inside the outer vector.

It seems that a vector of vectors is the perfect data structure for our Hub Labels. We can store each of the Hub Labels in a separate vector and the store all the pointers to the node vectors inside another vector. If we want to retrieve the Hub Labels of the node with id i, we can just call vector[i]. The first prototype of this implementation was written exactly in that manner. That approach comes with a number of problems which lead us to not use vectors for storage at all.

Even though we try to minimize data consumption, some systems don't provide enough memory (ram) to store even the needed graph data let alone the hub labels. For that reason our implementation also supports accessing the data directly from an external memory disk (preferably an SSD). Using an stl vector is no feasible in that case, because it always uses the main memory as storage. Conveniently Linux systems provide the system call mmap, which provides the programmer with a pointer into the disk memory. It also allocates the specified amount of memory. With mremap the allocated memory size can be adjusted at run time.

The need to use disk storage forces us to use pointers in our algorithms instead of vector references. To avoid duplication of code we want to use the same data structure code for main memory and external memory. To achieve that, instead of using vectors, we use alloc and realloc, which are equivalent to mmap and mremap but operate on main memory. Now only the allocation part differs and one can decide for each data structure if they want to use main memory or external memory with only a configuration value.

To minimize the used pointer amount and to easier store the data on disk, we decided to use one continuous block of memory for all Hub Labels. That requires an additional data structure which stores for each node the location and size of its Hub Labels.

We implemented different executables for pre processing and query operations. The pre processing executable calculates the desired amount of Hub Labels and stores all the graph and Hub Label data into disk. Every data structure has its own file and the file names and sizes are stored into a configuration file. That configuration file also contains the information for the query executable, whether to load the given data structures into main memory or operate on them directly from the disk. That decision can be made for each file individually. In Chapter 3.4 we will explore how disk storage affects performance.

3.4 Benchmarks

For benchmarking we used the German road network. It was constructed with an open source CH-constructor[1] from the Open Street Map graph data[2]. Uncompresed in binary data, the graph takes up 5.3 GB of storage. The used hardware was a Xeon E5-2650v4 with 768 GB RAM and with an SSD(1,8 GB/s sequential read speed).

The benchmark data compares the speeds and space consumption for the Hub Labels on different levels. Level 0 means that all Hub Labels have been constructed, level 1 means that all Hub Labels except for nodes with level 0 have been constructed, etc. We decided to use the level range from 0 to 14, because after that the required space compared to actual graph becomes negligible. Each level was benchmarked with 1000 random source target pairs and the caches were deleted after each query.

In Table 3.2, you can see how the total time to get the distance of the shortest path is composed of. As fewer Hub Labels are calculated, the graph search time rises and more Hub Labels have to be pulled, so the merge time also rises.

[1] https://github.com/chaot4/ch_constructor
[2] https://download.geofabrik.de

Table 3.2: Detailed comparison of the query speed between level.

Level	Search Time(µs)	Look Up Time	Merge Time(µs)	Total Time(µs)
0	0	13	6	19
1	8	12	25	45
2	11	12	44	67
3	14	9	52	75
4	14	10	53	77
5	18	10	56	84
6	18	11	57	86
7	19	10	61	90
8	21	9	67	97
9	24	8	70	102
10	26	17	72	106
11	31	9	85	125
12	37	19	96	151
13	45	25	110	180
14	58	19	135	212

Table 3.3: Comparison of query speed based on how many Hub Labels have been calculated.

Level	HL Space Consumption	speed up to normal ch (3763µs)
0	$173,6GiB$	$19,800\%$
1	$96,1GiB$	$8,322\%$
2	$55,0GiB$	$5,616\%$
3	$33,6GiB$	$5,017\%$
4	$22,7GiB$	$4,887\%$
5	$17,3GiB$	$4,480\%$
6	$15,63GiB$	$4,375\%$
7	$10,4GiB$	$4,181\%$
8	$7,28GiB$	$3,879\%$
9	$5,3GiB$	$3,689\%$
10	$4,04GiB$	$3,550\%$
11	$3,19GiB$	$3,010\%$
12	$2,63GiB$	$2,492\%$
13	$2,23GiB$	$2,090\%$
14	$1,95GiB$	$1,775\%$

In table 3.3 the required space for the Hub Labels if compared to the relative speed up from a normal ch query, without HL-Interpolation. At level 14, even though only a fraction of the graph space is used, the speed up is still by more than 1 order of magnitude.

Table 3.4: Comparison between main memory and disk.

Level	HL on disk	Graph on disk	Graph & HL on disk
0	2584	891	3372
1	3335	2486	5528
2	3525	2806	7433
3	3559	3012	8707
4	3639	3380	8721
5	3734	3421	9408
6	3796	3611	9138
7	3847	3745	10374
8	3914	3892	10816
9	3917	4103	11192
10	4015	4265	12236
11	4101	4438	12586
12	4098	4893	13638
13	4145	5034	14639
14	4213	5260	15114

In the previous benchmarks the data was always stored in main memory. In table 3.4, you can see the timings for data access from disk. The first timings result from storing only the graph data in RAM and the HL data on disk, the second from storing the HL in RAM and the Graph data on disk, the third from storing all data on disk. With the graph on the disk, a normal CH query takes on average 55ms. So we can see a great speed up with our interpolation technique.

While storing the graph on disk and the Hub Labels in RAM, does provide at lower levels an advantage compared to storing the HL on disk and the Graph on RAM, the path recovery is around 30 times slower(1ms RAM vs 30ms disk). So if the use case also requires to retrieve the path instead of just the distance, it is preferable to store the graph in main memory. In practice, without deleting caches sub-millisecond timings can regularly be achieved.

4 OSCAR Integration

Until now, we have shown how we have implemented a fast one to one routing application only from an algorithmic point of view, without taking a look at how it can be accessed by the user and solve his or her problems. A possible usage of the routing is to integrate with the search engine for open street map data OSCAR [BF15]. The goal is to allow for search queries along the shortest route from one point to another. An obvious use case would be to allow the users to search for gas stations or restaurants which are located close on the path to the desired target. But given the extensive query language of OSCAR, users can search for any item in the open street map data.

The implementation of this work exposes an HTTP-Endpoint in the OSCAR backend to query for routes and integrates into the OSCAR query implementation to allow for queries along a given route. To allow for a faster query routine the edges of the routing graph are mapped to the OSCAR data structures in a pre processing step. The routing APIs are then accessed and visualized by the OSCAR web application.

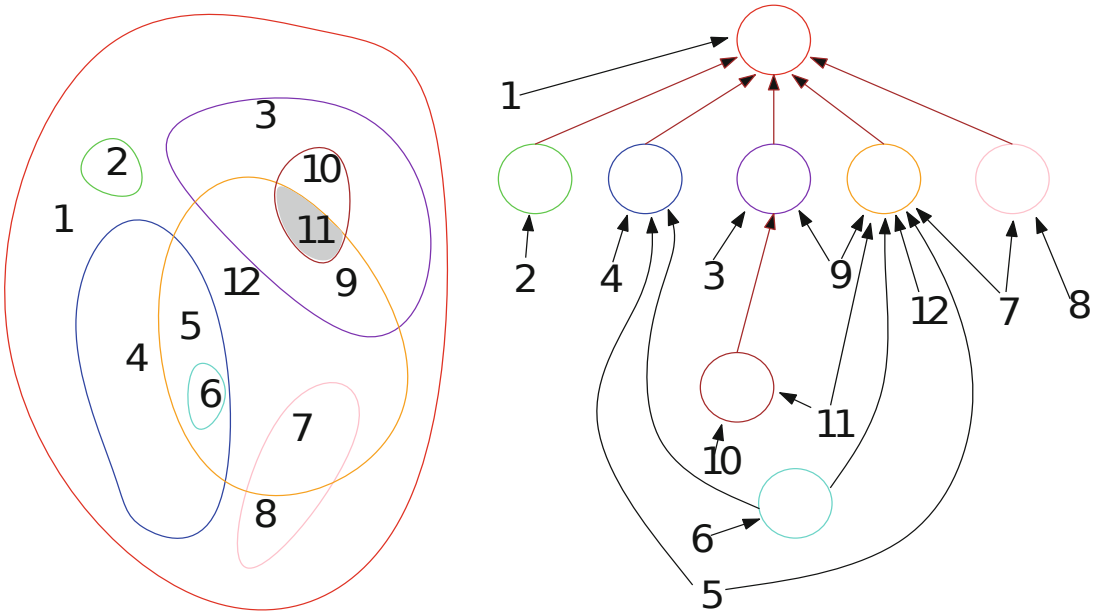
4.1 OSCAR basics

OSCAR is a search engine for Open Street Map(OSM) [1]. OSM is a free database for global map data such as streets and buildings. OSCAR indexes all OSM items to allow for a fast spatial and text searches.

The pre processing of OSCAR creates a *cell arrangement of regions*. It arranges all the regions of the OSM data into cells. The cells get tagged with the item data contained in them. At query time only those cells are further explored which contain tags that match the query string. The cells have a natural hierarchical order (see Figure 4.1) Since in practice the number of items greatly exceeds the number of cells, a fast query is possible.

[1] https://www.openstreetmap.org/

Figure 4.1: OSCAR cell arrangement of regions (left) and their resulting hierarchy (right). Source: [BF15] Page 159



The oscar query language is quite extensive and will only be covered partly here. The query string can contain a key value tag so that only items which also contain that key value tag get returned. For example a key value tag query could be `@amenity:restaurant`. This query would return all restaurants in the OSM Database. One can also query for items in a region. For example, `@amenity:fuel "Germany"` would return all the gas stations located in Germany.

OSCAR can be visually explored via a web application[2]. The visualization of the query result for `@amenity:fuel "Germany"` can be seen in Figure 4.2. One can see that if you are zoomed out the results get divided into their regions with choropleth maps. If you zoom in, each individual result will be displayed.

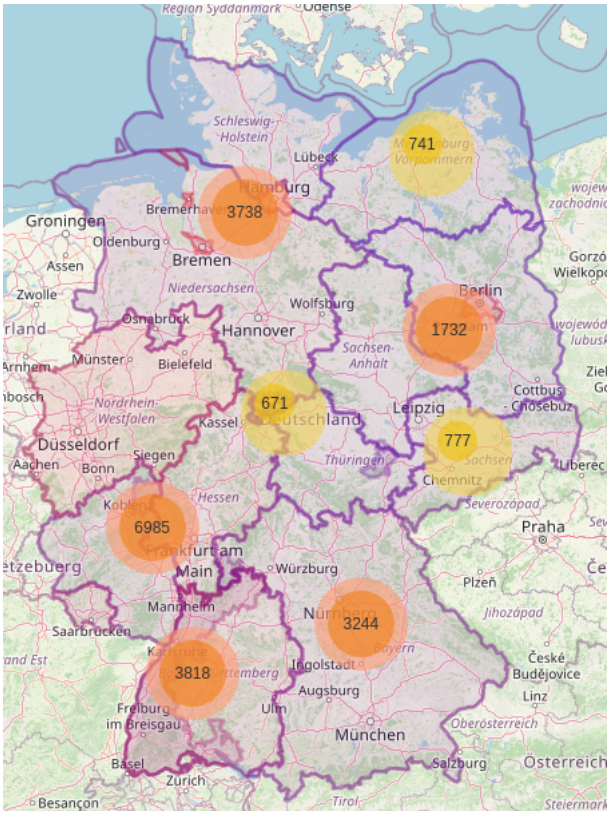
4.2 Backend implementation

For our backend integration we want to allow users to query for OSM items along the shortest path from one point to another. The API is directly implemented into the OSCAR WEB API [3] all method calls go over the same route as the OSCAR endpoint calls. There are two parts which need to be integrated to enable our use case.

- Expose an endpoint which returns the shortest path from a given source and destination point.
- Augment the OSCAR query language with the possibility to define a source and destination point to only return items that are located along the path.

[2] http://oscar-web.de
[3] https://github.com/dbahrdt/oscar-web

Figure 4.2: Visualisation of the gas stations in Germany displayed by oscar web.



4.2.1 Routing Endpoint

To get the shortest path and a distance a HTTP GET endpoint is exposed. The method call is located at the route `/oscar/routing/route`. It takes in a query parameter 'q' which has to be an array of geographic points, each with a longitude and a latitude. The response will be the shortest path, which starts at the first point visits every point in the given order and ends at the last point in the array. The returned path is also an array of points and the distance is dependent on the type of distance function the graph uses, but most commonly will be in seconds. An example request and response can be seen in Listing 4.1. Currently the response will also include some debug information, but that can be ignored.

Listing 4.1 An example request and reply for the OSCAR routing API.

```
request: "http://routing.oscar-web.de/oscar/routing/route?
  q=[[48.13676,11.5684],[52.5229,13.3813],[53.566,10.01953]]"

reply:
{
  "path": [[48.1363,11.5684],[48.1375,11.5691], ... , [52.5229,13.3813],
            ... , [53.566,10.01953]],
  "distance": 1503019058
}
```

4.2.2 Query Integration

As previously described OSCAR uses a cell arrangement to enable fast querying. When the search engine receives a query it will find the cells which contain information specified by the query. To find only items which are located along the path, the engine only needs to explore cells that the path intersects. We could do this at runtime, but for long paths that will be quite resource intensive. To speed it up, we calculate for each edge of the graph the cells it intersects and store the cell ids. At query time we collect all those cell ids and instruct the search engine to only consider the specified cells.

To then find gas stations along the route from point(48.7489, 9.1625) to point(50.19800, 10.4095) the query `@amenity:fuel $route(0,0,48.74894,9.162597,50.1980,10.40954)` is needed. Just like the for the routing endpoint, arbitrary many geographic points can be visited.

4.3 Frontend implementation

The existing frontend is a web application written with HTML, CSS and JavaScript[4]. It already gives the user the option to limit their search result by a user drawn rectangle, polygon or linear path. We extended that with the option to limit the search result by a shortest path for which the user can choose the endpoints by clicking on the map.

Once the user has selected at least two points a request to the routing endpoint will be send and the returned path will be drawn. The user can give the drawn path a name and reference the path in their search result with &name.

To draw a route the user has to go to the Geometry tab of the web applications side bar and select Route in the dropdown menu. After clicking on 'create' the user can start clicking on the map to specify their desired routing endpoints. The route will appear almost immediately after the user has set their second endpoint. The user can keep clicking to define points which should also be visited by the route. Once the user is satisfied with the route, they have to click on 'finish' and name their

[4] https://oscar-web.de

route to use it for OSCAR-queries. In figure 4.3 you can see the side bar of the web application after a route has been drawn. An example route can be seen in figure 4.4. The search result for gas stations along that route can be seen in figure 4.5.

Figure 4.3: Oscar side-bar after the route has been drawn.

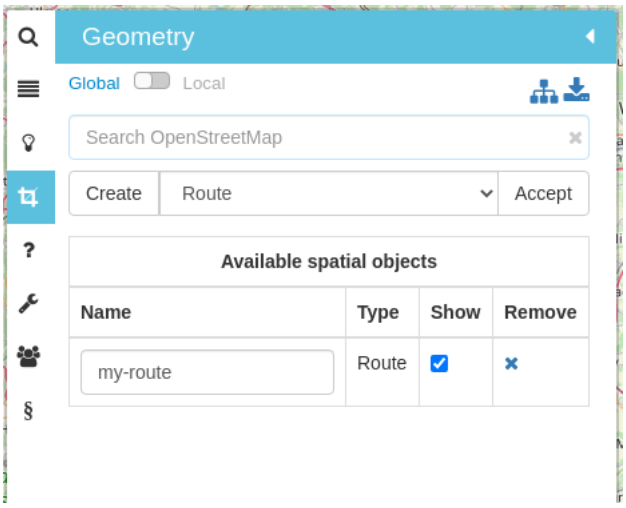


Figure 4.4: Route from Stuttgart to Würzburg drawn in orange.

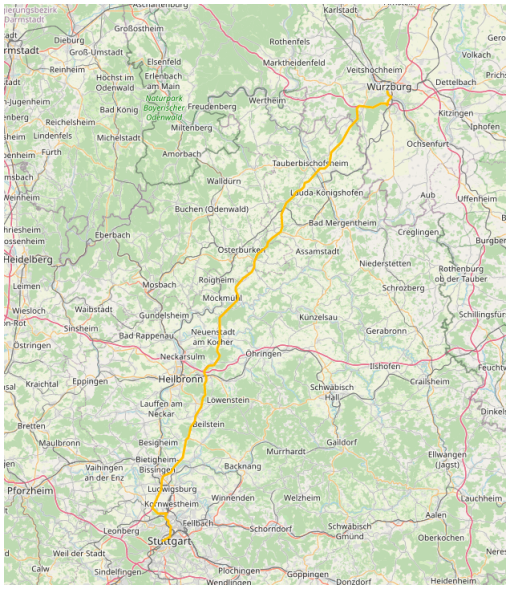
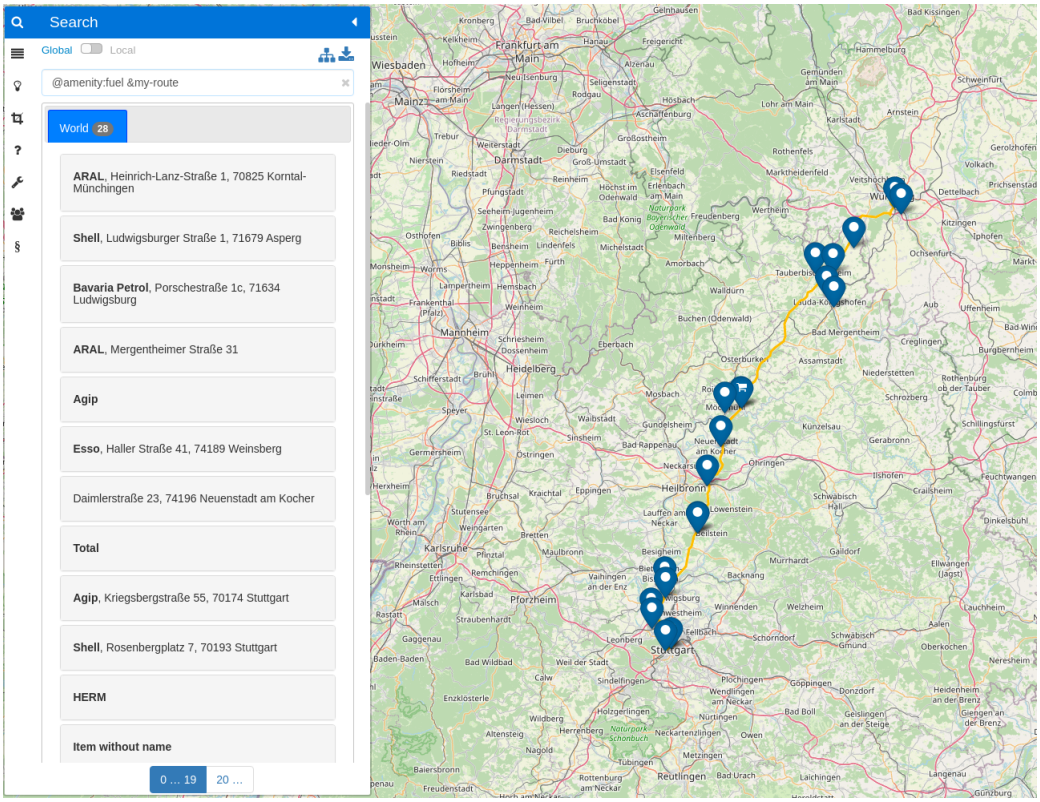


Figure 4.5: Search engine result after querying for gas stations along the route.



5 Conclusion and Outlook

We achieved our goal to implement a fast and reliable routing application, while also integrating it with the OSCAR search engine. The data can be selectively stored on disk and in RAM, both offer a significant speed up to the normal Contraction Hierarchy query. The backend and frontend integration with OSCAR enables the user to interactively explore the surrounding area along the shortest route. A demo of the project can be found at http://routing.oscar-web.de. The source code for the routing library can be viewed here: https://github.com/somakolli/path-finder-cli. The integrated OSCAR source code can be observed under https://github.com/dbahrdt/oscar-web/tree/routing.

As for future work we would propose implementing further routing functionality into the OSCAR frontend. For example, routing from one address to another, would enable the user to find their desired route more easily. Further research into efficient Hub Label storage, such as compression, would probably yield improvements in speed and in space consumption.

Bibliography

[BF15] D. Bahrdt, S. Funke. “OSCAR: OpenStreetMap Planet at Your Fingertips via OSm Cell ARrangements”. In: *Web Information Systems Engineering – WISE 2015*. Ed. by J. Wang, W. Cellary, D. Wang, H. Wang, S.-C. Chen, T. Li, Y. Zhang. Cham: Springer International Publishing, 2015, pp. 153–168. ISBN: 978-3-319-26190-4 (cit. on pp. 9, 27, 28).

[DGSW14] D. Delling, A. V. Goldberg, R. Savchenko, R. F. Werneck. “Hub Labels: Theory and Practice”. In: *Experimental Algorithms*. Ed. by J. Gudmundsson, J. Katajainen. Cham: Springer International Publishing, 2014, pp. 259–270. ISBN: 978-3-319-07959-2 (cit. on pp. 8, 14, 16).

[Dij+59] E. W. Dijkstra et al. “A note on two problems in connexion with graphs”. In: *Numerische mathematik* 1.1 (1959), pp. 269–271 (cit. on pp. 8, 12).

[DSSW09] D. Delling, P. Sanders, D. Schultes, D. Wagner. “Engineering Route Planning Algorithms”. In: *Algorithmics of Large and Complex Networks: Design, Analysis, and Simulation*. Ed. by J. Lerner, D. Wagner, K. A. Zweig. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 117–139. ISBN: 978-3-642-02094-0. DOI: 10.1007/978-3-642-02094-0_7. URL: https://doi.org/10.1007/978-3-642-02094-0_7 (cit. on p. 8).

[Fun20] S. Funke. “Seamless Interpolation Between Contraction Hierarchies and Hub Labels for Fast and Space-Efficient Shortest Path Queries in Road Networks”. In: *Computing and Combinatorics*. Ed. by D. Kim, R. N. Uma, Z. Cai, D. H. Lee. Cham: Springer International Publishing, 2020, pp. 123–135. ISBN: 978-3-030-58150-3 (cit. on pp. 8, 17).

[Gol08] A. V. Goldberg. “A Practical Shortest Path Algorithm with Linear Expected Time”. In: *SIAM J. Comput.* 37.5 (Feb. 2008), pp. 1637–1655. ISSN: 0097-5397. DOI: 10.1137/070698774. URL: https://doi.org/10.1137/070698774 (cit. on pp. 8, 13).

[GSSD08] R. Geisberger, P. Sanders, D. Schultes, D. Delling. “Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks”. In: *Experimental Algorithms*. Ed. by C. C. McGeoch. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 319–333. ISBN: 978-3-540-68552-4 (cit. on pp. 8, 13, 14).

All links were last followed on November 19, 2020.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature