Implementing Customizable Route Planning

Michael Wegner and Matthias Wolf

Abstract. We implement a routing engine described by Delling et al. in their paper *Customizable Route Planning in Road Networks* – short *CRP*. It supports all major features of a real-world application: fast queries on road networks of continental scale and quick adaptation of metrics due to changed traffic conditions or personal preferences. We extensively evaluated the performance of *CRP* and show that it is capable of handling road networks with more than 20 million vertices at an average query time of around 1 ms.

1 Introduction

Finding the fastest or shortest path from A to B is nowadays easier than ever. We just ask our favorite route planner to compute it for us. To make this ease of use possible, the routing engine has to answer queries within milliseconds since there usually is more than one user querying the system at the same time. Traditional approaches like Dijkstra's algorithm [3] would be too slow to compute shortest paths on country or continent-sized road networks. Therefore, we rely on speed-up techniques [1]. All of these techniques do some sort of preprocessing on the graph representing the road network before the system is ready to answer queries. The goal of preprocessing is to efficiently reduce the search space during queries and thus, improving their performance.

The problem with most speed-up techniques is the fact that the preprocessing phase uses the metric (i.e. the function that assigns a weight to each edge in the road network). This in turn means that the metric and therefore the weights cannot change between two preprocessings as this would result in wrong query answers.

However, in a real-world road network the metric will change very often since there might be traffic jams that make a highway temporarily slower than a smaller street, for example. And because traffic jams build up very quickly and only last for a few hours the metric gets updated frequently. For many speed-up techniques this means to rerun the preprocessing whenever the metric gets updated, which is often impossible because the time to run the preprocessing exceeds the time between two metric updates.

CRP [2] is one of the first routing engines that allows fast changes to the metric while having decent query times comparable to many other speed-up techniques. The basic idea is to split the preprocessing phase into two subphases – one that only uses topological information about the graph and one that uses the result of the first subphase and the metric information. The second subphase has to be repeated whenever the metric is changed and therefore should be fast. The first phase only needs to be recomputed when the topology of the graph changes (e.g. a new street is built) which does not occur as often.

In a road network, intersections occur at vertices of the graph. However, at some intersections it might not be possible to make u-turns or to turn left, for example. Such restrictions are called turn restrictions. This cannot be modeled with a single intersection vertex. In order to take this into account, we implicitly blow each vertex of the graph

up to a complete bipartite graph between the entry points of incoming and exit points of outgoing edges (see Figure 1). The edge weights of the bipartite graph can be used to model turn restrictions and are called *turn costs*. For example, when we cannot turn left from entry point 1 to exit point b (see Figure 1), we set the corresponding turn cost (edge weight) in the bipartite graph to infinity.

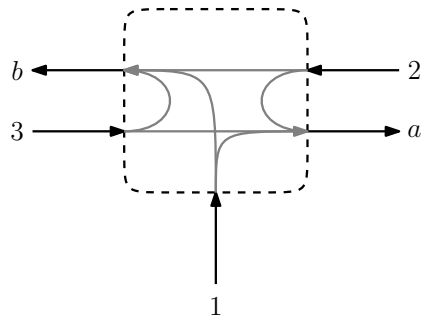

Fig. 1: Graph vertex with entry and exit points connected to a complete bipartite graph.

Outline. In the next section we describe *CRP* in more detail. In Section 3 we provide further details about our implementation and in Section 4 we show the results of our experiments before giving a conclusion in Section 5.

2 Customizable Route Planning

CRP [2] is a separator–based routing algorithm, that is, the input graph is subdivided into smaller cells. *CRP* uses a multi-level partition that is created by partitioning the graph recursively. To accelerate the route calculation, an overlay graph of shortcut edges is created. More precisely, for each cell the overlay graph contains a complete graph between the cell's vertices. See Figure 2 for an example. The weight of an overlay edge uv is defined as the length of the shortest path from u to v inside the cell. This ensures that lengths of shortest paths in the original and overlay graph are the same.

2.1 Preprocessing

During the preprocessing phase we set up the basic data structures used by *CRP*. First, we recursively partition the graph, which yields a multi-level partition. We number the levels from 1 to ℓ, where level 1 contains the smallest cells. We then determine all cut edges (i.e. edges that have end vertices in different cells) and add a vertex to the overlay graph for each of its endpoints. Hence, each overlay vertex belongs to exactly one original (cut-) edge. Note that this means if a vertex of the graph has multiple incident cut edges, there are as many overlay vertices as there are incident cut edges. We call an overlay vertex v an *entry vertex*, if the cut edge is an incoming edge of v, and *exit vertex* otherwise.

Each cell C can only be entered at an entry vertex and left via an exit vertex. Therefore we connect each entry vertex to every exit vertex of C, thus yielding a complete bipartite

graph between C's entry and exit vertices. Note that the edge weights depend on the metric and therefore are calculated during the customization. Additionally, data structures to map from original graph vertices to overlay vertices and vice versa are created.

2.2 Customization

Given a weight function, we determine the weights of the overlay edges. The weight of an overlay edge (u, v) in a cell C is the weight of the shortest path from u to v inside C in the original graph respecting the turn costs. Thus, the overlay vertices have no turn costs associated with them. We compute the weights in a bottom-up fashion, i.e., we first calculate the weights of edges in cells on level 1 and use this information to build the levels up to level ℓ.

2.3 Query

A query consists of a source edge su that is closest to the geographical start point and a target edge vt that is closest to the geographical target point defined by the user. The source vertex is then defined as the start vertex s of the source edge and the target vertex as the end vertex t of the target edge. As we want to support turn costs, we have to use a turn-aware version of Dijkstra's algorithm on the original graph. On the overlay graph, however, the turn costs are included in the edge weights and, thus, a normal version of Dijkstra's algorithm is sufficient. To accelerate the computation a bidirectional search may be used [1].

Routing from source to target vertex is then done by a modified Dijkstra algorithm. Whenever we scan a neighbor v of a vertex u, we first compute v's query level in that we want to search further from v. The query level of v is the highest level on which both s and v as well as both v and t are in different cells. Note that if v lies in the same cell as s or t then we scan v in the graph itself. In all other cases, we scan v in the respective query level in the overlay graph.

Figure 3 shows an example query from s to t. Starting from the source vertex s we apply Dijkstra's algorithm [3] on the original graph until we reach a border vertex v_1 where one of its neighbors v_2 is in another cell than s. The vertex v_2 is later considered on the first level (green) of the overlay graph. Note that we cannot consider v_2 on the second level (brown) since both s and v_2 are contained in the cell on the second level. When we later pop v_2 from the priority queue, we scan its neighbors in the overlay graph on level one. From vertex v_3 we reach a neighbor vertex v_4 that is again in another cell than s and t, but this time even on the second level (brown). Therefore, we can scan the neighbors of v_4 on the second level of the overlay graph. This process is continued this way until we are back on the graph and reached the target vertex t. Note that even though there is only one up–going and one down–going path in terms of levels in this example, there can be queries where we go up and down in levels multiple times.

Turn Costs To support turn restrictions at intersections (vertices) we have to know from which edge we entered the vertex. Since we can enter a vertex through multiple edges when the vertex has more than one incoming edge, it is not enough to use the vertex as a key in the priority queue. A solution is to use tuples (v, i) as keys where i is the i^{th} incoming edge. Note that this is only needed when we route on the graph. The weights of

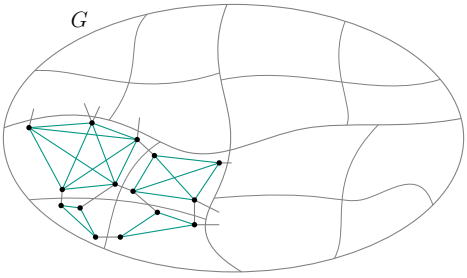

Fig. 2: Overlay graph on lowest level for one of the cells on the highest level.

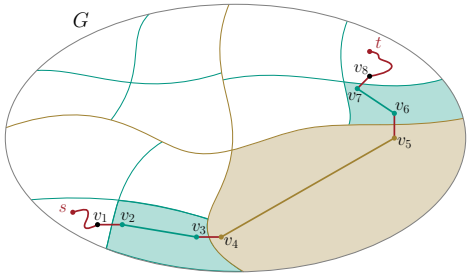

Fig. 3: Shortest path computation from s to t using the shortcuts of the overlay graph.

edges incident to overlay vertices are the shortest path costs to the neighboring overlay vertices that already include the turn costs.

3 Implementation

In this section we describe our implementation of *CRP* in more detail. We start with a description of our graph data structures, which form the core of our implementation. Afterwards, we explain how the three phases of *CRP* are implemented using these data structures.

3.1 Data Structures

The two main data structures are the *Graph* and the *OverlayGraph* class that are described in further detail in this section.

In addition to those two classes we use a data structure *MultiLevelPartition* that stores for every vertex in which cell it is located on every level of the overlay graph. Based on this information *cell numbers* are derived to determine on which level we have to consider the vertex during queries. To do so, all cells are numbered, such that all cells in the same super cell have unique numbers. Packing the numbers of the cells in levels 1 to ℓ containing vertex v into one 64-bit integer yields v's *cell number*. High-level cells can be addressed by truncated cell numbers since the cell numbers of all vertices inside a cell have a common prefix. How the numbers are packed is stored in a *LevelInfo*. A detailed description can be found in Section 5.1.1 in [2].

To store the weights computed during the customization phase we have a class *OverlayWeights* that is just a wrapper of the weights vector to allow easy access to the weight of an edge in the overlay graph.

Graph. For the graph we use an adjacency array representation [4]. To support backward searches, all edges are stored in forward and backward direction in two arrays such that all vertices can easily access their incoming and outgoing edges. The indices at which an edge is stored in forward and backward direction is the edge's *forward ID* and *backward ID*, repectively. All edges have attributes such as length associated with them.

To support turn costs, each vertex has a *turn table* associated with it. As there are fewer different turn tables than vertices, each vertex only stores an offset into a large array containing all turn tables in row-major order. Similarly, each vertex only stores a pointer to a cell number, which only uses 32 bits instead of the 64 for the cell numbers themselves.

To map from graph vertices to the overlay vertices, we also have a hash table that maps a *SubVertex* to the ID of the corresponding overlay vertex. A *SubVertex* is a triple (v, i, f) consisting of a vertex v, a turn order i and a flag indicating whether the i-th entry or exit point of v is meant.

OverlayGraph. The overlay graph is represented by its vertices and information about the cells. Each overlay vertex v stores the ID of the original vertex and the ID of the cut edge it corresponds to. Exit overlay vertices store the edge's forward ID, whereas entry vertices store the backward edge. Additionally, v contains the ID of the neighbor overlay vertex, i.e., the other endpoint of the boundary edge, and its cell number.

As the edges inside all cells form complete bipartite graphs, we represent their weights as a matrix. All these matrices are written in row-major order in one vector. Since this vector depends on the metric, it cannot be part of the metric-independent overlay graph. Instead, it is stored in an *OverlayWeights*-object.

The layout of this vector, however, is metric-independent and can be stored in the overlay graph. For each cell there is a structure containing an offset into the weights vector and the cell's ID. Additionally, the number of entry and exit vertices are stored. Similar to the edge weights, the entry and exit vertices of all cells are stored in one vector and each cell holds an offset. The cells themselves are stored in hash maps – one for each level – that map the cell numbers to the actual cells.

3.2 Preprocessing

Parsing. We use OpenStreetMap (*OSM* [1]) road networks since they include the turn restrictions at each intersection as well as additional information for each edge (e.g. street type) that can be used by metrics. OSM graphs are encoded as XML files that we parse within the class *OSMParser*. Parsing includes extracting the vertices and edges including their attributes as well as the turn restrictions.

Turn restrictions for an intersection vertex v are modeled as a matrix of dimension $inDeg(v) \times outDeg(v)$ where we store restriction types such as "only right turn". Here, $inDeg(v)$ denotes the in-degree of v and $outDeg(v)$ the out-degree of v. Since there is only a limited number of different intersection types it would be wasteful to store a distinct matrix for each vertex. We therefore hash the matrices to prevent duplicates. Since two matrices can have the same number and type of entries but in a permuted order we apply an additional sorting of rows and columns before hashing the matrices.

All parsed information is then stored in our Graph class and written to disk. To reduce disk space usage we compress it with the bzip2 library accessed via boost [2].

We provide the program *osmparser* that can be called from the console with the following command:

[1] http://download.geofabrik.de
[2] http://www.boost.org

`./osmparser path_to_osm_file.bz2 path_to_graph.bz2`
where `path_to_osm_file.bz2` is the path to the OSM file that is compressed with bzip2 and `path_to_graph.bz2` is the path to the graph file that will be generated by the parser. This program will create an additional graph file in the METIS [3] file format. This will be later used during the partitioning step.

Partitioning. Having the graph extracted from the OSM file, we then call an external tool Buffoon [5] that partitions the graph recursively and outputs an instance of our *MultiLevelPartition* class. Based on this information we build our *LevelInfo* and set the cell numbers of all vertices.

OverlayGraph. Building the overlay graph is done in the constructor of *OverlayGraph* and consists of two steps: building the overlay vertices and the cell information.

We first determine all boundary edges by iterating over all edges and add an overlay vertex for each endpoint if these lie in different cells. The overlay vertices are stored in a vector, where the index of a vertex represents its ID. They are sorted by the highest level in which they are at the boundary of their cell, i.e., the highest level in which a vertex and its neighbor are in different cells on this level. The vertices on the same level are sorted by the IDs of their corresponding vertices in the original graph. This improves the locality of our code since the original vertices in the same cell have contiguous IDs.

We also populate the hash map in the graph that maps triples of (original) vertex, turn order and a flag indicating whether we search for an entry or exit point to the corresponding overlay vertex. This map enables us to map from the graph to the overlay graph.

After the creation of the overlay vertices, we collect all entry and exit vertices for each cell. Using this information we build the structures representing the cells: The number of entry and exit points (n_{entry} and n_{exit}, respectively) can be retrieved easily. The vectors containing the entry and exit vertices are joined together to form one large vector. While doing so, we set the offsets into this vector accordingly. See Section 5.1.2 in [2] for a detailed description. The offset into the weights vectors can be calculated as the prefix sum over the product $n_{\text{entry}} \cdot n_{\text{exit}}$, since each cell contains exactly that many edges.

3.3 Customization

During the customization we calculate the weights of the overlay edges. This is done in the constructor of *OverlayWeights*. As the weight of an overlay edge (u, v) inside a cell C corresponds to the length of the shortest path from u to v inside C, we run shortest path queries from each entry point to all exit points of C restricted to the interior of the cell. We handle the cells in a bottom-up fashion, i.e., first the cells on level 1 and then the levels up to ℓ. Doing so, we do not need to calculate the shortest paths on the graph but only on the level right below the current level.

As cells on the same level are completely independent of each other, we parallelize the computation of edge weights for edges in the same level. We do not, however, parallelize the weight calculation for edges inside the same cell.

[3] http://glaros.dtc.umn.edu/gkhome/metis/metis/overview

3.4 Query

We provide several versions of the query algorithm. The unidirectional version of the algorithm is defined in the class *CRPQueryUni*. A bidirectional version of this algorithm is implemented in *CRPQuery*. We additionally provide a parallel variant of the bidirectional algorithm in *ParallelCRPQuery*.

CRPQueryUni. As mentioned in Section 2.3, we have to use tuples (v, i) to address the priority queue for vertices v of the graph where i denotes the i^{th} incoming edge at v. This is necessary to enable the usage of turn restrictions. Since each incoming edge of the graph has a unique identifier, we use this edge identifier to address the priority queue instead of a tuple.

Note that this approach makes it necessary to create distance and parent arrays as well as priority queues that require $\mathcal{O}(|E|)$ space since we associate the tentative distance to a specific entry point of a vertex. This seems very wasteful since the only graph edges we look at during a single query are the ones within the cell of s and t. We therefore apply the following optimization to this approach: During preprocessing, we sort the edges according to their cell on the lowest level and store the largest number e_{max} of edges a single cell has on level one. Then it is enough to reserve $2 \cdot e_{max}$ instead of $|E|$ for each array and the priority queue. Since both the number of vertices per cell and the average degree are constant, we reduced the needed space to a constant. This resulted in significant performance improvements.

We additionally implemented another optimization that is described by Delling et al. in [2] as *stalling*. Having visited a vertex from an entry point i automatically adds an upper bound to all the exit points of that vertex defined as the sum of the distance from s to entry point i and the turn cost to reach the exit point. By storing those upper bounds, we can then prune the search from entry point j if we cannot improve the upper bound implied at any exit point. This obviously reduces the search space and therefore results in another performance improvement.

CRPQuery. The *CRPQuery* class implements the bidirectional version of *CRPQueryUni*. We therefore need the distinction between forward and backward search making it necessary to have additional distance and parent arrays as well as a priority queue for the backward search. We switch between forward and backward search by always selecting the queue that contains the lowest value. The search is aborted whenever the queues run empty or the length of the current shortest path is less than the sum of minimum values of forward and backward priority queues.

ParallelCRPQuery. We also implemented a parallel version of our bidirectional query algorithm. In *ParallelCRPQuery* the forward and backward search are run in parallel, each using one thread. The only part where there might occur race conditions is when we update the shortest path. This can be the case when both the forward and backward search update the shortest path at the same time, leading to potentially wrong results. We evaluated two solutions to this problem. For the first solution we introduced a lock that allows atomic access to the shortest path update. However, in our experiments we observed that this resulted in no speed-up at all. In fact, the parallel algorithm was even

slower than the sequential counterpart. For the second solution we simply treated the shortest path variable as an upper bound to the actual shortest path. Whenever the forward or backward search update the shortest path variable we actually found a valid path. No matter which search direction found the smaller path, both found lengths are an upper bound to the actual one.

Path Unpacking. To extract the shortest path, we need to store more information during the query. When we decrease the tentative distance of a vertex v while iterating over the neighbors of another vertex u, we store u as v's *parent* in the parent array. To unpack the path, we follow the parent pointers from the target to the source. This results in a path that might use overlay edges.

This overlay path is unpacked by a *PathUnpacker*, which iterates over the edges in the path and replaces all shortcuts by paths in the original graph. This is done in a top-down fashion, i.e., if the shortcut e lies on level k, a shortest path query inside e's cell on level $k-1$ is run. This process is repeated recursively until all paths are on the original graph.

PathUnpacker expects the overlay path to contain both start and end point of its shortcut edges. Especially, when the path enters or leaves the overlay graph, the entry and exit overlay vertices must be included.

4 Experiments

We evaluated our implementation of *CRP* [2] by testing the individual components as well as the complete system on differently sized road networks. Testing a routing engine can be split into two main categories: performance and quality. The performance of a routing engine is much easier to evaluate than the quality of the route found by the engine. The quality of routes is not always possible to verify easily and depends on a broad set of attributes of the metrics. We therefore mainly focused on the performance category but also checked whether our routing engine always finds the shortest paths and interprets any turn restrictions correctly occurring on some of our test routes.

4.1 Benchmarking Setup

All experiments were done on a machine equipped with a 3.7 GHz Intel Xeon CPU E5-1630 v3 having 4 cores, each having 2 hardware threads. It is equipped with 10 MB L3 cache and 128 GB of main memory clocked at 2133 MHz. The machine runs 64 bit SUSE Linux and we compiled our code with g++-4.8.3 and OpenMP 3.1.

For parsing the OSM files and converting them into our graph data structure we used another machine that was equipped with 256 GB of main memory since the parsing step used more than 128 GB on some graphs.

4.2 Benchmark Instances

We used two road networks of different size, the main properties can be found in Table 1. All our instances are OSM road networks since they include turn restrictions and several edge attributes we use for our metrics.

Road Network	# Vertices	# Edges	Source [4]
karlsruhe-regbez	754179	1511705	http://download.geofabrik.de/
germany	23435399	47409424	http://download.geofabrik.de/

Table 1: Benchmark Instances used in our experiments.

4.3 Results

As already mentioned, we mainly focused on the performance of our *CRP* implementation since it is much easier to objectively evaluate than the quality of the computed routes.

Performance. We evaluated the performance of both the customization and the query phase. Since the preprocessing was dependent on an external partitioning tool, we decided to omit preprocessing times in this evaluation.

Customization. We measured both the sequential and parallel runtime of the customization phase for every benchmark instance of Table 1. Since we have multiple metrics with different turn costs and edge weights, the runtimes slightly vary when customizing for different metrics. The average customization runtimes for all metrics can be seen in Table 2. On *karlsruhe-regbez*, we get a speed-up of 1.7 and on *germany*, we get a speed-up of 2.2. The theoretical maximum speed-up on our benchmark machine would be 4. Therefore, the speed-up could be much better, but we still benefit from parallelization. The parallelization of our sequential implementation could be improved in the future and it would be interesting to see how much we would benefit from a better implementation. Nevertheless, the times are short enough to allow quick re-customizations once the system is running.

Query. For evaluating the query time, we took 100000 random s, t-pairs and evaluated the runtime of all our three algorithms *CRPQueryUni*, *CRPQuery* and *ParallelCRPQuery* (see Section 3.4). Figure 4 shows the average query times of our algorithms on the benchmark instances using the distance metric. We also informally tested the running times on other metrics but the results are comparable to the ones obtained with the distance metric. Note that the query times do not include the time to unpack the actual path (see Section 3.4). We can see that *CRPQueryUni* has an average query time of 0.12 ms and therefore outperforms both *CRPQuery* with 0.18 ms and *ParallelCRPQuery* with 0.16 ms on *karlsruhe-regbez*. Since the bidirectional search consumes more space, we suspect that the main reason for that result are cache misses, which are caused by switching often

[4] accessed on January 8, 2016

between forward and backward search. We can see that the parallel version of *CRPQuery* is indeed a bit faster, but with a speed-up of 1.12 by not very much.

Regarding the query times on *germany*, we see that *CRPQueryUni* is still slightly ahead of *CRPQuery* (0.908 ms vs. 0.978 ms) but the difference is much less (50% on *karlsruhe-regbez* vs. 8% on *germany*). Since the routes of our average queries are longer on *germany*, the search space grows and therefore, the bidirectional search gets better. Comparing *CRPQueryUni* to *ParallelCRPQuery*, we see that *ParallelCRPQuery* is 50% faster. Also, the speed-up is now 1.64 which is already quite close to the maximum of 2. It would be interesting to check whether *CRPQuery* outperforms *CRPQueryUni* on even larger graphs like the road network of Europe in the future.

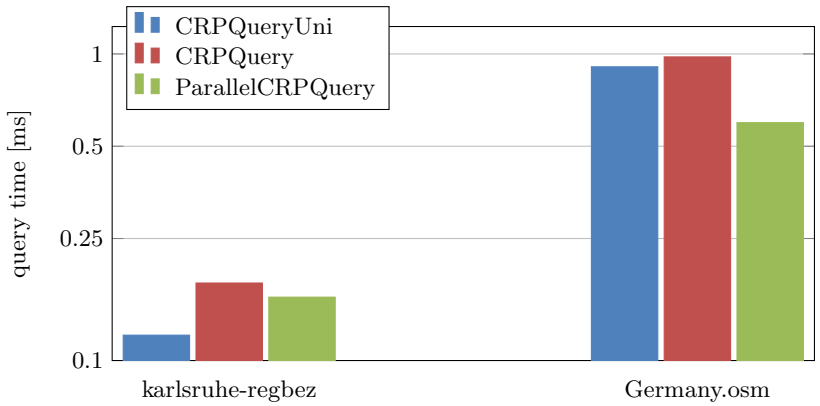

Fig. 4: Average query times of our algorithms with distance metric on benchmark instances.

Graph	**Customization**		**Path Unpacking**
	sequential [s]	parallel[s]	[ms]
karlsruhe-regbez	0.53	0.31	0.73
germany	16.08	7.19	4.18

Table 2: Average runtimes for sequential and parallel customization and unpacking the paths.

Path Unpacking. Since the shortcuts of a *CRP* query have to be unpacked to obtain a path ready to display on a map, we also measured the time it took to unpack the paths. The average times for unpacking the paths of 100000 random s, t-pairs can be found in Table 2. We see that the average path unpacking takes 0.73 ms on *karlsruhe-regbez* and 4.18 ms on *germany*. Since the unpacking is independent within different cells, the algorithm would definitely benefit from a parallelization in the future.

Quality. We informally tested some specific queries with known turn restrictions and checked that our algorithms handled each restriction correctly. While this check is by no

means a proof of correctness, it is at least a strong indicator that everything works as expected.

Regarding the computed distances and times, we got results which were comparable to the ones obtained by a professional routing engine like Google Maps [5].

5 Conclusion

We implemented *CRP* by Delling et al. [2] and extensively evaluated the performance of the routing engine. Changing the metrics of a large road network is indeed possible within seconds, which makes *CRP* very interesting for real-world applications. It turned out that our bidirectional algorithm does not pay off on small graphs but seems to close the gap on larger ones. For *germany*, our parallel version of the bidirectional algorithm (*ParallelCRPQuery*) significantly outperforms the sequential one. We therefore expect that on larger road networks like Europe, our bidirectional versions will both outperform the sequential one. Regarding the path unpacking, we see that this step consumes significantly more time than the actual query. Our current implementation is sequential but should be easily convertible into a parallel algorithm in the future.

References

1. Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F Werneck. Route planning in transportation networks. *arXiv preprint arXiv:1504.05140*, 2015.
2. Daniel Delling, Andrew V Goldberg, Thomas Pajor, and Renato F Werneck. Customizable route planning in road networks. *Transportation Science*, 2015.
3. Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.
4. Kurt Mehlhorn and Peter Sanders. *Algorithms and data structures: The basic toolbox*. Springer Science & Business Media, 2008.
5. Peter Sanders and Christian Schulz. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *LNCS*, pages 164–175. Springer, 2013.

[5] https://maps.google.com/