Shortest Path Speed Up Techniques – Lower Bounds and Applications

Von der Fakultät Informatik, Elektrotechnik und
Informationstechnik der Universität Stuttgart zur Erlangung
der Würde eines Doktors der Naturwissenschaften
(Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Jochen Eisner

aus Wolgast

Hauptberichter:	Prof. Dr. Stefan Funke
Mitberichter:	Prof. Dr. Joachim Giesen

Tag der mündlichen Prüfung: 28.11.2013

Institut für Formale Methoden der Informatik
Universität Stuttgart

2013

Contents

I. Preliminaries 5

1. Fundamentals 7
   1.1. Graphs and Paths 7
   1.2. Road Networks 8
      1.2.1. Distance Metrics 9
   1.3. Computing Shortest Paths 10
      1.3.1. Dijkstra's Algorithm 12

2. Implementation 15
   2.1. Static Graphs 15
   2.2. Miscellaneous Data Structures 17
      2.2.1. Fast Dijkstra 17

II. Transit Node Constructions Revisited 23

1. Motivation 25
   1.1. Related Work 27
   1.2. Outline 28

2. Integer Linear Programming 31

3. Transit Node Computation as Hitting Set Problem 35
3.1. Notions of “long” & Algorithmic Details 36
3.2. Comparison to Existing Transit Node Constructions 38
3.3. Preprocessing 39
3.3.1. Primal Algorithm 39
3.3.2. Dual Algorithm 41
3.3.3. Set Computation 41
3.3.4. Computation of Access Nodes 42
3.4. Query 44

4. Experimental Results 47
4.1. Implementation 47
4.1.1. Instance-based Approximation Guarantees 48
4.1.2. Set Generation 48
4.1.3. Memory Management 50
4.2. Results 50
4.2.1. Comparison with Previous Constructions . 51
4.2.2. Query timing 53

5. Discussion and Extensions 57

III. Applications of speedup Techniques: Path Prediction 59

1. Motivation 61
1.1. Related Work 63
1.2. Outline 64

2. Basic Concepts 65
2.1. Reach 65
2.2. Quality Metrics 70

3. Path Prediction Strategies 73
3.1. Offline Strategies 74
3.2. Online Strategies 75

4. Experimental Results 77
4.1. Quality 77
4.2. Cost 83

5. Discussion and Extensions 85

IV. Applications of speedup Techniques: Sequenced Route Queries 87

1. Motivation 89
1.1. Related Work 91
1.2. Our Contribution 96

2. Basic Concepts 97
2.1. Contraction Hierarchies 97
2.1.1. Implementation Details 100

3. Speeding-Up Sequenced Route Queries 105
3.1. Iterative Doubling 105
3.2. CH enhanced Iterative Layer Search 107
3.3. Arbitrary Order Routes 108

4. Experimental Results 111
4.1. Query Locality 112
4.2. Influence of the Order of POIs 113
4.3. Influence of the Number of POI types 114
4.3.1. Arbitrary Order Routes 115

5. Conclusion and Future Work **121**

V. Epilogue **123**

Conclusion **125**

Zusammenfassung **127**

Bibliography **128**

Preface

Our Contribution

The widespread proliferation of portable hand held computing devices in the form of mobile phones, rivaling 6 year old desktop computers in raw computational power and outclassing them in terms of communication interfaces and interoperability, gave rise to a plethora of new geospatial applications and services. One of the many roles a modern mobile phone can provide, is the complete substitution of printed maps with added functionality as navigation aid, for self localization, or – with more semantic back-end information – complex routing queries all around the world.
With the computer-based compilation of the majority of the worlds' road networks, which are freely available to everyone in the form of OpenStreetMap, vast geospatial databases are to our disposal today. One of the most fundamental questions in such a network is to compute a shortest (or quickest) path between two designated points. This problem was solved by Dijkstra in 1959 in a provably optimal way, but his algorithm, although very elegant and simple in design, does not scale well on continent sized road graphs. Therefore a multitude of alternative approaches for the single-source-single-target shortest path problem and more complicated flavors were devised in the last decade. The motivation for highly efficient algorithms in this field is twofold. On the one hand they enable a real time user experience on a mobile phone, even for complex tasks, on the

other hand these approaches scale well when a server back-end is employed, whose task is to server a large number of mobile agents. If used in combination these techniques enable us to compute different shortest path related problems in the order of ten milliseconds on the complete road network of Europe. This constitutes a speedup of more than three magnitudes compared to Dijkstra's approach.
This work consists of three main parts, each addressing an exemplary problem in the field of geospatial application which are outlined in the following sections.

Transit Node Constructions Revisited

In the first part we reconsider the concept of *transit nodes* as introduced by Bast et al. [BFM06]. Transit nodes are an offline speed up technique which enables very fast point to point shortest path distance computations and are based on a precomputed point to point distance table of a small subsets of nodes – the transit nodes.
For the first time we construct instance based lower bounds on the size of transit node sets by interpreting a LP formulation of the problem and its dual and, as a side product, we achieve considerably smaller access node sets which directly improve the query time for non-local queries.
We devise an algorithm to construct transit node sets for this theoretic framework and verify their properties with a practical implementation.
This result was also published as "Transit Nodes - Lower Bounds and Refined Construction" at the *14th Meeting on Algorithm Engineering and Experiments (ALENEX)* in 2012 [EF12b].

Applications of speedup Techniques: Path Prediction

In the second chapter of this work we work on the *path prediction problem* – given the path a mobile user has moved along in a road network up to the current moment, predict where the user will move along in the near future. In contrast to known solutions to this problem we will compute our prediction not only based on the geometry of the known path (using extrapolation) or directional information implied by the underlying road network but make explicit use of the structure of the space of shortest paths in the network. Our proposed path prediction algorithm is equally simple but yields extremely accurate predictions at a very low computational cost.
This work was published as path of the paper "Algorithms for Matching and Predicting Trajectories" in the *13th Meeting on Algorithm Engineering and Experiments (ALENEX)* at 2011 [EFH^{+}11].

Applications of speedup Techniques: Sequenced Route Queries

The third and final chapter considers the problem of a *sequenced route query* – the problem of planning an optimal route (quickest or shortest) that visits facilities of the respective type (for example a gas station or a super market) on the way home. The proposed solution, based on the combination of a distance sensitive doubling technique and contraction hierarchies, is orders of magnitudes faster than either a naive approach or previous results. In addition it produces the answers in an instant for realistic queries without compromising guaranteed optimality. With such fast query times, this type of route query becomes feasible even on mobile devices or for high-throughput web-based route planners.
This work is published as "Sequenced Route Queries: Getting Things Done on the Way Back Home" in the *Proceedings of the 20th Inter-*

national Conference on Advances in Geographic Information Systems (SIGSPATIAL) in 2012 [EF12a].

Part I.

Preliminaries

1. Fundamentals

In this chapter we lay the foundation for this work with the introduction of the fundamental definitions and relevant algorithms in the application area of shortest path computation in road networks.

1.1. Graphs and Paths

A graph G is a tuple of a finite set of nodes (or vertices) V and edges E as Cartesian power over V, connecting these nodes. We denote the cardinality of these two sets as $|V| = n$ and $|E| = m$. The concept of graphs is fairly general so we define more specialized graph types with regard to our applications. Each edge shall connect exactly two nodes, so $E \subseteq V \times V$. Let there be an edge e which connects the nodes a and b. If we are interested in the specific orientation of this connection we denote the edges as tuples, so $e = (a, b)$ and call G a directed or digraph. In this case we call a the source and b the target of e. Otherwise the edges are just sets itself and we denote them as $e = \{a, b\}$. We refer to the in-degree $\text{inDeg}(v)$ of an node v as the number of edges with v as target, so $\text{inDeg}(v) = |\{e \in E : e = (s, v), s \in V\}|$. The out-degree of a node is defined accordingly. The reverse graph $\overline{G}(V, \overline{E})$ of a digraph G shares the exact same nodes as G but interchanges all source and targets, so $\overline{E} = \{(a, b) : (b, a) \in E\}$. We call a graph simple if there are no two edges between the same two nodes and also no self loops of the form $e = (v, v)$ with $v \in V$.

A path $p = (e_1, e_2, \ldots, e_k)$ in G is a sequence of sequentially connected edges $e_{1\leq i\leq k} = (s_i, t_i)$ with $t_i = s_{i+1}$ for all $1 \leq i \leq k$. It is called simple if all the nodes induced by p are unique. If $s_1 = t_k$ the path is called a cycle. A graph without any cycles is called acyclic. Note that such a path p can also be unambiguously defined by nodes instead of edges if the graph is simple.
For a subset $V' \subseteq V$ we call the graph $G' = (V', E')$ induced subgraph of $G = (V, E)$ if $E' = \{e \in E : e = (a, b), a \in V' \wedge b \in V'\}$, so G' inherits all the edges of G that are connecting valid nodes in V'.
We can also extend $G = (V, E, \gamma)$ by an cost function $\gamma : E \rightarrow \mathbb{R}$ which assigns a weight (length) to every edge. The overall length of a path $p = (e_1, \ldots, e_k)$ is defined as the sum of its edge weights $\sum_{e_{1\leq i\leq k}} \gamma(e_i)$. We denote this length as $d_\gamma(p)$ or in short $d(p)$.

1.2. Road Networks

Directed graphs offer a natural way to model road networks with the representation of intersections as nodes and road segments as directed edges. It is also common to diverge from the pure structural representation of the network itself in favour of additional spatial information. Considering this agenda edges are often partitioned in chains of degree two nodes and nodes are augmented by data of their absolute spatial position. Moreover the weight function is chosen to convey the modelled road segment type.
The graph representations of road networks exhibit some interesting properties which will be heavily exploited in the more advanced algorithms later on.
Road networks are generally very sparse with $m \in \mathcal{O}(n)$ furthermore the maximum degree of any node is low. They are nearly planar and due to the underlying objective of a road network they

reveal a distinct hierarchy with small residential networks at the bottom and highways spanning entire continents at the top. This characteristic is one of the reasons that shortest paths in road networks are most often unique once they exceed a certain length.

1.2.1. Distance Metrics

In the previous section we discussed the transformation of spatial and structural features of road networks into digraphs which leaves us with the choice of an appropriate weight function.
Two viable possibilities come to mind. The first one is a straight mapping of euclidean edge length to weight. This approach conserves the spatial properties of the graph but equalizes all the different road types into one common class. On the other hand it will enable us to compute absolute euclidean path lengths. Although euclidean length conveys meaningful information, more often we are not interested in the shortest but in the fastest s-t-path with minimal travel time. In this regard the weight function should also convey the type of the road segment in addition to its length as the average travel speed on a highway is much higher compared to passing along a small rural driveway.
This observation further places emphasis on the hierarchical nature of the network as city connecting expressways are channeling travel time optimal routes of certain length which is their exact point in the first place.

With the OpenStreetMap database [OSM] as our main source for real world road networks, we use the meta information for each road segment to define its maximum travel speed and therefore the time needed to traverse it. OSM provides a fine grained categoriza-

tion[1] for the typically encountered pathway types. Figure 1.1 lists the types we considered as passable by car and the top speeds and therefore travel time we assigned to them.
The OSM guidelines try to map these road types to the local road types in each country in a consistent way, such that continent wide road networks are internally consistent in this regard. For Germany "motorway" would denote an Autobahn, "primary" a Bundestraße and finally "secondary" a Landstraße.

road type	speed limit in km/h	road type	speed limit in km/h
living street	30	secondary link	80
motorway	130	service	30
motorway link	130	tertiary	70
primary	120	tertiary link	70
primary link	120	trunk	130
residential	45	trunk link	130
road	50	turning circle	50
secondary	80	unclassified	50

Figure 1.1.: A selection of the different road segment types in the OSM metadata as described in and their assigned top speed. These top speeds allow us to deduce a travel time for each segment.

1.3. Computing Shortest Paths

Shortest path computation is the most fundamental building block for both path prediction and transit node computation. The formal

[1] http://wiki.openstreetmap.org/wiki/Map_Features#Highway

definition of the classic shortest path problem can be formulated as follows: Given a graph $G = (V, E, \gamma)$ and two nodes $s, t \in V$, compute the path $p = (s, \ldots, t)$ starting in s and ending in t which minimizes $d(p)$. We will refer to this path as $\pi(s, t)$.
As both start and endpoint of p are fixed and we are only interested in exactly one optimal path this problem is also called single-source single-target shortest path problem. A straightforward extension is the omission of t as fixed source which results in the problem formulation of given a node $s \in V$, compute the shortest paths $\pi(s, \ldots, v)$ for all $v \in V$. Without even fixing the source we get the problem of conducting an all pairs shortest path computation, or more formal: find the shortest paths $\pi(v, \ldots, w)$ for all pairs $v, w \in V$.

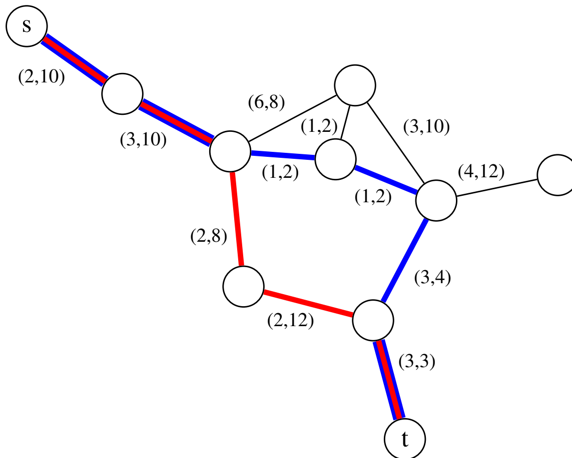


Figure 1.2.: A graph $G = (V, E, \gamma)$ with two different distance metrics (γ-functions). The (a, b) weight vector for every edge denotes the respective assigned weight of each edge. While the red path is a shortest s-t-path under γ_1, γ_2 implies a different, blue shortest path.

In general we assume that the shortest path between two nodes is unique. This is not so much a necessary constraint for the correctness of the following algorithms but often simplifies the reasoning about the ground truth of our models.

1.3.1. Dijkstra's Algorithm

Dijkstra's algorithm [Dij59] is designed to answer single-source shortest path queries in graphs with non-negative edge weights. Given a graph $G = (V, E)$ and a source node $s \in V$ it returns a shortest path tree rooted in s as union of all shortest paths $\pi(s, \cdot)$ in G.
In order to accomplish this task the algorithm maintains two pieces of information for every node v in V which are a distance value as length of a possibly non-optimal s-v-path and the predecessor of v in this path. The algorithm partitions the set of nodes V in three subsets $V = A \cup B \cup C$. In set A are all the nodes where the distance and predecessor label are final and the algorithm is certain about the correctness of these values. These nodes are called settled nodes. In the set B are nodes with adjacent neighbors in A where the algorithm has assigned a distance and predecessor label but the resulting paths are not yet confirmed to be optimal. So B could be imagined as small zone around A which has to be extended once A grows. Finally in set C are nodes the algorithm has not yet discovered.
The high level idea is now to start with $A = \{s\}$, as we know that $\pi(s, s) = \emptyset$ with length 0 is optimal. After creating B accordingly the algorithm moves node after node from B to A once it has verified the the correctness of its specific labels. As A contains more and more nodes new nodes are discovered on its border and with the assignment of temporary labels to this border nodes they are migrated from set C to B. The algorithm finishes once the set B runs

empty and it is unable to explore new nodes around A. Afterwards we have nodes in A with some finite distance and a predecessor implicitly realizing this distance and possibly also nodes in C with a distance label of infinity. The nodes in C are exactly the nodes where no path $p(s, \ldots, v \in C)$ exists and therefore no shortest path can be found.
Now the main essence of Dijkstra's algorithm is the order in which nodes are explored and in this regard the ability to argue about the correctness of a certain distance label such that the changeover from nodes in B to A is justified.
To this regard the algorithm employs a priority queue Q which holds all nodes in B sorted by their distance labels. Q is initialized as only containing s with distance 0. The following series of operations is repeated until Q runs empty: Extract the node c with minimum distance from Q. Let c have a certain distance $\text{dist}(c)$ and a predecessor $\text{pred}(c)$. Consider all outgoing edges of c and for each of these edges $e = (c, v_i) \in E$ compare $\text{dist}(v_i)$ with $\text{dist}(c) + \gamma(e)$.If it holds that $\text{dist}(v_i) > \text{dist}(c) + \gamma(e)$ update $\text{dist}(v_i)$ to the later value, set $\text{pred}(v_i)$ to c and add v_i with its new distance label into Q. This operation is also called edge relaxation. After the inspection of all those edges the algorithm extracts the next node from Q. Note that the extraction of the node with minimal distance corresponds to the action of settling this node and implicitly adding it to A. If an edge $e = (c, v_i)$ is relaxed because $\text{dist}(v_i) = \infty$ the node v_i is explored the first time and implicitly moved from C to B.
As long as all edge weights are non-negative Dijkstra's algorithm observes the following properties: Each node $c \in V$ is either extracted exactly once from Q with its final distance and predecessor label, or never added into Q iff there is no s-c-path. For every extracted node c its outgoing edges $e = (c, \cdot) \in E$ are considered exactly once, each of them may or may not result in an

relaxation. Therefore the the runtime of Dijkstra's algorithm is $\mathcal{O}(n \cdot \mathcal{T}_{\text{extractMin}} + m \cdot \mathcal{T}_{\text{decreaseKey}})$ and strongly dependent on the particular implementation of Q. Employing a Fibonacci heap which supports the extractMin operation in amortized $\mathcal{O}(\log n)$ and the decreaseKey operation in amortized $\mathcal{O}(1)$ time, Dijkstra's algorithm therefore observes an overall runtime of $\mathcal{O}(m + n \log n)$.

If we are only interested in a single shortest s-t-path it may seem wasteful to compute the complete single-source shortest path problem. To address this concern two modifications may be employed. The first one is to simply stop the algorithm once t was settled. We know that both distance and predecessor label of t are correct at this point so we can reconstruct $\pi(s,t)$. The second one is to start two Dijkstra computations in a bidirectional search. The first one DJ_1 on G, starting in s and the second one DJ_2 on the reverse graph $\overline{G}$, starting in t. If we interleave their computation in such a way that the succession of settled nodes distances is an increasing sequence, i.e. the one DJ with the smaller potential distance to settle is allowed to do so, there will be a point in time when a node $c \in v$ got settled by both computations. For this node both $\pi(s,c)$ and $\pi(c,t)$ are known and therefore $L = \text{Len}(\pi(s,c)) + \text{Len}(\pi(c,t))$ is an upper bound for the length of $\pi(s,t)$. Also all nodes $v_i \in V$ without any label from DJ_1 are more distant than c from s and the same holds for the nodes without a label from DJ_2 and their distance to t. Once we know L we can prune the search space of both DJ computations as we do not need to explore any nodes without a label from the other Dijkstra computation if both of them have settled all nodes up to distance R and $2R > D$. Also every node settled by both Dijkstra computations may tighten the bound L. Overall this results roughly in a speed-up of factor two over the unidirectional version.

2. Implementation

In this section we describe the most important design decisions for the basic algorithms described so far.

2.1. Static Graphs

Our graph implementation is optimized for static directed graphs and based on the idea of an offset array. The graph holds two global arrays, each holding all edges. The first of these arrays is sorted by source whereas the second one is sorted by target. A third global array holds all the nodes. Each node has indexes into the in- and out-edge array to the first end behind the last of its incident edges. This data structure is called 'adjacency array' representation in the literature [MS08]. The approach results in constant sized nodes and edges and guarantees that all incident edges of a given node are in one consecutive block of memory. The decision to hold each edge twice and sorting them this way was made with regard to Dijkstra like algorithms which greatly benefit from the resulting locality as they heavily rely on enumerating the incident edges for possible edge relaxations. The meta data added to both edges and nodes will be based on the necessitates of the concrete application. Edges are expanded by both euclidean length and travel time. Fig. 2.1 shows the basic interfaces for both nodes and edges.
The precomputation step of a contraction hierarchy requires to add edges to this graph structure. In this case we will manage a tempo-

rary set of these additional edges and rebuild the complete structure once integration becomes necessary in a batched fashion. This approach avoids the linear time penalty for adding nodes or edges into this structure without actually supporting constant time edge set operations by means of a more complex data structure.
This is the only instance in our use case where we will have to 'dynamically' add edges to our road graph.

```
template <typename metaNodeType>
struct node {
  uint ID;
  uint beginOutEdge, endOutEdge;
  uint beginInEdge, endInEdge;
  metaNodeType metaData;
};
```

```
template <typename metaEdgeType>
struct edge {
  uint ID;
  uint source, target;
  metaEdgeType metaData;
};
```

Figure 2.1.: Basic node and edge structure.

Figure 2.1 lists the bare bone structure of nodes and edges in our static graph structure. We prefer unsigned int offsets into the edge and node arrays over pointers based on the fact that 2^{32} addressable features are sufficient. The complete network of Europe consists of about 241M edges, if one counts all types of path ways. This way we get away with 4 bytes per offset, compared to 8 for a pointer.
Overall a node requires 5×4 bytes for the graph structure and typ-

ically 2×8 bytes more for the longitude and latitude of each node, totaling 36 bytes. Each edge requires 3×4 bytes for the graph structure and 4 more byte per weight of which we will typically hold two, totaling 20 bytes per edge.

2.2. Miscellaneous Data Structures

2.2.1. Fast Dijkstra

In practice the two most critical aspects for a fast Dijkstra computation are the layout of the outgoing edges to be enumerated for each settled node and the performance of the priority queue. As described in section 2.1 we took great care in the optimization of the edge layout and size to ensure cache locality. Regarding the later point there exists research for very complicated multistage priority queues which are finely tuned to the cache hierarchy of the specific platform [San99]. In our work we decided to elude this engineering heavy problem and use the standard libstdc++-v3 priority queue instead. This queue is backed by a fairly well tuned implementation of a single heap in form of a balanced binary tree in an array.
For our specific use case the standard queue is consistently 30% slower compared to the one provided by [San99], but does not require parameter tuning for element size or cache sizes. The libstdc++-v3 also provides several drop in replacement implementations[1] for the standard heap based queue which are all slower in our use case according to our measurements.
The main quirk of the standard priority queue interface is the absence of an `increase-key` operation (the std::priority_queue is a max-queue) which is required in the relaxation step of Dijkstra's

[1] `http://gcc.gnu.org/onlinedocs/libstdc++/ext/pb_ds/pq_design.html`

algorithm. As listed in figure 2.2 and 2.4 we deal with this shortcoming in the following way: We do not decrease any distances and instead insert the to node to be relaxed as a second copy with its new distance into the priority queue. Therefore some nodes will be multiple times in the queue, whereas only the instance with the lowest distance is the ”real“ one. To cope with this problem we introduce an additional check each time we `pop` the top element from the queue, we check its distance label from the priority queue against the value saved in the distance array for this particular node. If both these value match, the current top element is the ”real“ copy (although we do not know if there are copies with larger distances in the queue at this point) and proceed as normal. If the distance is larger than the value in the distance array, we do hold a copy and just drop it. The number of copies of each node is limited by the in-degree of a node. Each time we settle one of the nodes’ neighbors we could decide to relax the node and set the respective neighbor as predecessor. In contrast - based on our measurements, the space and time overhead induced by copies in the priority queue is lower than 10% on our typical road network instances at any given time.

```
1 struct Length {
2   uint operator()(const simpleEdge& E) const {
3     return E.metaData.length[0]; }
4     typedef uint costType;
5   }
6 };
```

Figure 2.2.: Example for an Cost class returning an weight for an edge of the correct type. This is completely inlined by the compiler.

```
struct dijkstra::pqType {
  uint distance;
  uint node;
  bool operator<(const pqType& o) const {
    return distance>=o.distance;
  }; // the std::priority_queue<T> is a maxQueue
};

// some members of the dijkstra class
const graphType& Graph;
std::array<uint, N> Distances;
std::array<uint, N> Predecessors;
std::priority_queue<pqType> PQ;
```

Figure 2.3.: Basic version of the one-to-all or one-to-one shortest path computation of our implementation of Dijkstra's algorithm. Part I – Continued Below in Fig 2.4.

Our real implementation of Dijkstra's algorithm is more complicated as compared to the version in figure 2.4 out of the necessity for higher flexibility. We also need to to calculate one-to-many and many-to-many shortest paths, as well as the ability to add additional break criteria as number of `pop` operations or distance from the source. We archive this flexibility by rigorous management of the `dijkstra` class state.

One final important performance trade of, which we managed by two separate implementations of the `dijkstra` class itself, is the question whether one holds the complete distance, predecessor and target etc. arrays for very fast access and is willing to pay the space and time needed to initialize them, or alternatively implements them as dynamic sets and maps. The later version requires minimal space or word on initialization but pays extra for every lookup.

In practice we are able to perform about 5.3M `pop` operations per

second for the array version and 1.15M `pop` operations per second for the map version on a Intel Core i5-2500K at 3.3Ghz and one thread. These numbers are relatively unaffected by the size of the road network.

```
14 template <typename graphType, class Cost>
15 void dijkstra::compute( const uint source,
16                         const uint target =
                               dijkstra::noTarget) {
17   PQ.push(pqType({0, source}));
18   while (!PQ.empty()) {
19     const pqType currTop = PQ.top();
20     PQ.pop();
21     const uint currNodeId = currTop.node;
22     const uint currDistance = currTop.distance;
23     const typename graphType::nodeType& currNode =
         graph.getNodes()[currNodeId];
24     if (currNodeId == target)
25       return;
26     if (Distances[currNodeId] == currDistance) {
27       for (uint e = currNode.beginOutEdges;
28                 e < currNode.endOutEdges; e++) {
29         const typename graphType::edgeType& currEdge =
             Graph.getOutEdges()[e];
30         const nodeId currTargetId = currEdge.target;
31         const uint newDist = currDistance +
             Cost()(currEdge);
32
33         if (Distances[currTargetId] > newDist) {
34           Distances[currTargetId] = newDist;
35           Predecessors[currTargetId] = e;
36           PQ.push(pqType({newDist, currTargetId}));
37 } } } } }
```

Figure 2.4.: Basic version of the one-to-all or one-to-one shortest path computation of our implementation of Dijkstra's algorithm. Part II – Continuation from fig. 2.3. The cleanup and setup of the member variables `Distances`, `Predecessors` and `PQ` is done externally in other member functions of the `dijkstra` class. Note the special handling of duplicate PQ elements in line 29. The `Cost` class returns an `uint` weight for a given edge (by returning one of the values of the edges' `metaEdgeType` extension).

Part II.

Transit Node Constructions Revisited

1. Motivation

Dijkstra's algorithm is still the baseline when it comes to computing the shortest path distance in a graph with non-negative edge weights. For a given source node A and a target node B it computes the shortest path in time $\mathcal{O}(n \log n + m)$ which seems best possible in the comparison model, *if no preprocessing on the graph is allowed*. With preprocessing the experienced query times can be drastically improved. On one hand there are techniques that allow for a pruning of the Dijkstra search such as A^* (here some potential function $\phi : V \to \mathbb{R}$ is precomputed (or chosen 'on the fly') to modify edge costs), ArcFlags [KMS06, GKW06] (edges are tagged if they are relevant for a shortest path to some target region), Reach [Gut04] (edges/nodes are classified according to their importance and whether they can appear in the middle of a long shortest path), and many more. The latest of these techniques allow for query times in the order of *milliseconds* in contrast to plain Dijkstra which takes in the order of seconds on a moderate size road network like the US or Europe with about 20 million nodes and 50 million edges. A completely different approach was introduced by Bast et al. in [BFM06]. There, one considers the set P of all "long" shortest paths (at this point we leave the term "long" imprecise on purpose, but think of "longer than 90km") in the graph $G(V, E, \gamma)$. Each $\pi \in P$ can be represented as the sequence or set of its nodes. For sake of a simpler exposition we focus on the case of an undirected graph, mentioning important differences for the directed case along

the way.
Preprocessing: In a preprocessing step we want to compute a set of *transit nodes* $T \subset V$ such that $\forall \pi \in P$ we have $\pi \cap T \neq \emptyset$. What is this set T good for, what other properties of T are desired?
Let us start with the following observation which seems natural when thinking about it for a while:

> If you are travelling "far" – let's say further than 90km – you will leave your local neighborhood on one of *few* arterial routes. Nearly all road networks exhibit a hierarchy of more and more "important" routes which are meant to channel large amounts of traffic. You would use them as feeder road to a highway or ring-road if your target isn't in the imminent vicinity. With Stuttgart and its periphery as example you would use one of only four Bundesstraßen (10,285,14 or 27) and ultimately one of two Autobahnen (8 or 81) to reach *any* other large city in Germany – regardless of the target's geographic location.

In reality, a handful of such routes suffice. So, looking at the paths in P starting at some specific vertex v, on the *first few kilometers* (leaving the local neighborhood) they all share one of let's say 5 common prefixes. If we could make sure that all "long" paths leaving from v *are hit by* T *in the local neighborhood* of v, we can define as the set of *access nodes* AN_v of v the first node from T on each of these prefixes, so typically we expect $|AN_v|$ to be a small constant. In the preprocessing stage – apart from determining T – we compute and store

- for each pair $(v, w) \in T \times T$ the distance between v and w
- for each $v \in V$ the distances to all $p \in AN_v$

Query: For a given distance query from s to t which are sufficiently far apart — also called *non-local* –, we can determine their exact distance by evaluating $\forall v \in AN_s, \forall w \in AN_t$ the expression $d(s, v) + d(v, w) + d(w, t)$ and taking the minimum. All terms of this expression are known after the preprocessing stage and there are essentially $|AN_s| \times |AN_t|$ expressions to evaluate. If s and t are not "far apart" – or *local* –, we resort to some other strategy such as contraction or highway hierarchies [GSSD08, SS05], which will be very fast due to source and target being nearby.
Of course, this whole scheme relies on the hope that one can construct a small enough set T (essential of size $\mathcal{O}(\sqrt{|V|})$ such that storing the $|T| \times |T|$ distance table does not require humongous additional space) and still cover most shortest paths. The query time for "far away" queries depends on the actual sizes of the access node sets.

1.1. Related Work

In [BFM09] an algorithm was presented which showed that it is possible to construct a rather small set T such that storing the $|T| \times |T|$-sized distance table requires essentially $O(|V|)$ space. At the same time T can cover a very large fraction of all shortest paths, and the access node sets were also rather small. More concretely, for the road network of California consisting of 1.6 million nodes and 3.9 million undirected edges, in [BFM09] a transit node set T was constructed with $|T| = 15087$ covering more than 97.1% of all shortest paths in the network and the average size of an AN_v was around 9, so a typical query required around 81 lookups in the precomputed data structure[1]. Transit node sets of the same order of magnitude

[1] They also reported similar numbers for the larger US road network.

were reported for a refined construction by Sanders and Schultes [SS09].

Up to this point, it was not clear, though, whether the same covering rate was possible with – let's say 700 transit nodes, or maybe even just 300 transit nodes. In [ADF$^+$11] and [AFGW10] Goldberg et al. propose the notion of *highway dimension* to obtain theoretical explanations for the great success of transit nodes and other acceleration schemes. They are more concerned with *upper bounds*, though, while our work is concerned with *lower bounds*.

1.2. Outline

In this chapter we derive *instance based lower bounds* on the size of the transit node sets that can be constructed for several notions of "far". This is achieved by modelling the problem of computing a transit node set T as a *hitting set problem* for which we set up an (integer) linear programming formulation as well as its relaxation and dual. By a dual fitting argument we can show that a simple greedy algorithm in practice achieves very small approximation ratios implying that it is not possible to compute transit node sets considerably smaller than e.g. in [BFM09] or [SS09].
As a nice side effect, our algorithm produces access node sets which are considerably smaller (≤ 4 on the average) resulting in query times that are one magnitude faster than the results reported in [BFM09]. Still, we do not consider our technique to be of practical importance, though, due to the humongous preprocessing time and space, but rather as a computational proof and insight that the transit node construction schemes developed so far are close to optimal for the considered network instances. As a small partial result we also show a simple filter for deciding locality of shortest path

queries based on the triangle inequality – this might be of actual use in practical implementations of any transit node scheme.

2. Integer Linear Programming

The classic hitting set problem is defined as follows: Given a universe U and a family $\mathcal{H}$ of subsets from U, the goal is to choose the smallest subset $T \subset U$ such that $\forall H \in \mathcal{H} : H \cap T \neq \emptyset$. Its formulation as the following primal integer linear program (ILP) is rather straightforward:

$$\text{min:} \sum_{u \in U} x_u \tag{2.1a}$$

$$\text{s.t.:} \sum_{u \in H} x_u \geq 1, \qquad \forall H \in \mathcal{H} \tag{2.1b}$$

$$x_u \in \{0, 1\}, \qquad \forall u \in U \tag{2.1c}$$

where we have a variable x_u for each element from U indicating its presence in T. There is a constraint for each $H \in \mathcal{H}$ which demands that at least one of the elements in H is in T. In the linear programming (LP) relaxation the integrality constraint on the x_u is replaced by $x_u \geq 0$. In our concrete setting, we have $U = V$, i.e. the universe consists of all vertices of the road network and $\mathcal{H}$ at this point consists of all "long" paths $\pi \in P$. *Remark: Later we will argue that $\mathcal{H}$ should not consist of all "long" paths but prefixes thereof, partly because we want to make use of our "local neighborhood" observation.*

The dual of this LP formulation is a *fractional packing problem*.

Here we have a (the same) family $\mathcal{H}$ of subsets from the universe U and aim at selecting as many fractionally disjoint sets from $\mathcal{H}$ as possible. This can be formulated as the following dual LP:

$$\begin{aligned} \text{max:} \quad & \sum_{H \in \mathcal{H}} y_H & & \text{(2.2a)} \\ \text{s.t.:} \quad & \sum_{u \in H} y_H \leq 1, & \forall u \in U \quad & \text{(2.2b)} \\ & y_H \geq 0, & \forall H \in \mathcal{H} \quad & \text{(2.2c)} \end{aligned}$$

where y_H indicates to what (fractional) degree set $H \in \mathcal{H}$ is chosen. Replacing $y_H \geq 0$ by the integrality constraint $y_H \in \{0, 1\}$ we obtain an integral set packing problem where the goal is simply to choose as many disjoint sets as possible.

Clearly, the objective function value of the optimal *fractional* solution to the primal LP is a *lower bound* to the optimal *integral* solution to the primal ILP. Analogously, the objective function value of the optimal *fractional* solution to the dual LP is an *upper bound* to the optimal *integral* solution to the dual ILP. By strong duality, the objective function values of the optimal fractional solutions to the primal and dual LP are the same.
Of what use is this formalism for our concrete problem of computing transit nodes? The approach in [BFM09] or the one proposed in the following computes a *feasible integral* solution to the primal problem formulation (typically not optimal, though). Let us assume that this solution has objective function value z_{prim}. If we exhibit a possibly fractional but feasible solution to the dual problem with objective function value z_{dual} with $z_{prim}/z_{dual} \leq \alpha$ for some $\alpha \geq 1$, we know that z_{prim} is at most an α-factor above the optimal integral solution to the primal problem, since the optimal integral solution to the primal problem is (in terms of the objective function value)

sandwiched between any feasible primal integral solution and any dual feasible solution.

As the hitting set problem is essentially the same as the *set cover* problem, it seems very unlikely that any polynomial-time algorithm can compute an integral solution which in general is always a $o(\log n)$ factor away from the optimal integral solution [Vaz01]. It is important to note, though, that for concrete problem instances it might be well possible to derive dual feasible solutions which are only a small constant factor away from the primal solution and hence proving close-to-optimality of the primal integral solution.

In the remainder of this chapter we will develop a greedy algorithm for the primal problem formulation and a strategy for obtaining a feasible dual solution which for the problem instances of transit node computation was always only a small *constant* factor away from the primal greedy solution in terms of the objective function value hence proving the close-to-optimality of the respective solution from our algorithm. This correlation is visualized in figure 2.1. The instance based approximation guarantee is the ratio between the greedy integral solution size of the set packing problem compared to the dual greedy integral solution size of the hitting set instance.

As described above it would be very desirable to compute the optimal fractional solution of either the primal or dual formulation of a given instance to obtain a tighter approximation bound. We will discuss in section 4.1 that the resulting set cover instances, obtained by 'moderately' sized graphs are very large. It will be infeasible to compute either the exact integral solutions or the exact fractional solution for our setting in general.

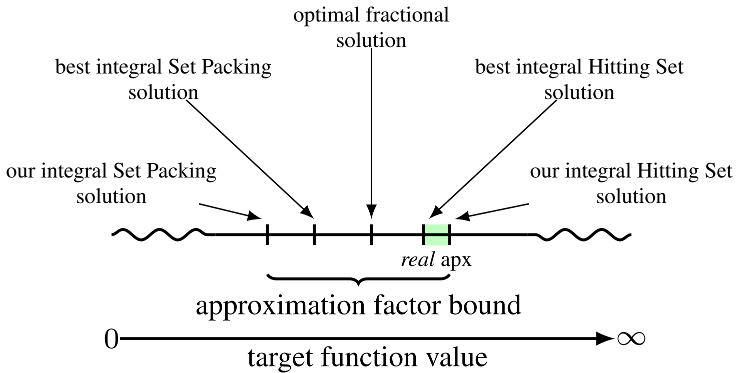


Figure 2.1.: Connection between primal and dual LP solutions. By the strong duality theorem the objective function values of the fractional set packing and hitting set problems are identical for an optimal solution. Around this common value is an integer gap for both integral solutions. Finally the solutions of our greedy algorithms are approximations for the optimal integer solutions. The instance based approximation bound is the distance between both greedy solutions.

3. Transit Node Computation as Hitting Set Problem

The previous section should have made clear how the problem of computing transit nodes fits into the LP framework of hitting set and set packing. So, for some notion of "long" we could first compute the set of all "long" shortest paths and consider them as a family of subsets from V. Of course this seems hardly feasible for networks with several million nodes since this family of subsets would contain in the order of $|V| \cdot |V| \approx 10^{12}$ many sets, each of which might be pretty large, too. Here, the specific characteristics of our hitting set problem comes as a rescue. When we consider some "long" path π, then clearly all prefixes of this path which are also "long" are contained in $\mathcal{H}$ – since they are also shortest paths. Each of them has to be hit by our hitting set, too, so we can actually restrict $\mathcal{H}$ to the set of *minimal* "long" shortest paths, i.e., all "long" shortest paths for which no prefix is also "long". Only this observation reduces the size of $\mathcal{H}$ and each individual $H \in \mathcal{H}$ such that it gets treatable in practice.

If we want to make use of our observation about traveling far and local neighborhoods, we furthermore want to enforce that each minimal long path π emanating from v is hit locally, i.e. "close" to v. We can achieve this by further truncating π to be an even shorter prefix, so at the end, our sets H to be hit by our hitting set algorithm are prefixes of minimal shortest "long“ paths. This constraint might look artificial but has already been implicitly been imposed

by [BFM09] since there the local access nodes aka relevant transit nodes were forced to be nearby by construction. So the problem we consider for the rest of the chapter is following:

> *For some notion of 'long' and 'nearby', compute a set of transit nodes $T \subset U$ which hits all 'long' paths 'nearby' its starting point.*

The framework of LP duality will allow us to come up with guarantees about the quality of solutions to this problem.

3.1. Notions of "long" & Algorithmic Details

There are different ways to define what a "long" path is or when two nodes A and B are "far" apart. The following are natural choices:

1. a shortest path from A to B is "long" if the *Euclidean distance along the path* from A to B is more than some constant D
2. a shortest path from A to B is "long" if the *Graph distance (e.g. travel time)* along the path from A to B is more than some constant D
3. a shortest path from A to B is "long" if the Dijkstra rank of B w.r.t. A *and* the Dijkstra rank[1] of A w.r.t. B in the reversed graph is more than some constant D

Let $m : V \times V \rightarrow \mathbb{R}_0^+$ be the distance function which determines what a "long" shortest path is. The choice of m determines how the

[1]The Dijkstra rank of a node v w.r.t. some other node s is k if in a Dijkstra computation starting at s, v is pulled as k-th node from the priority queue.

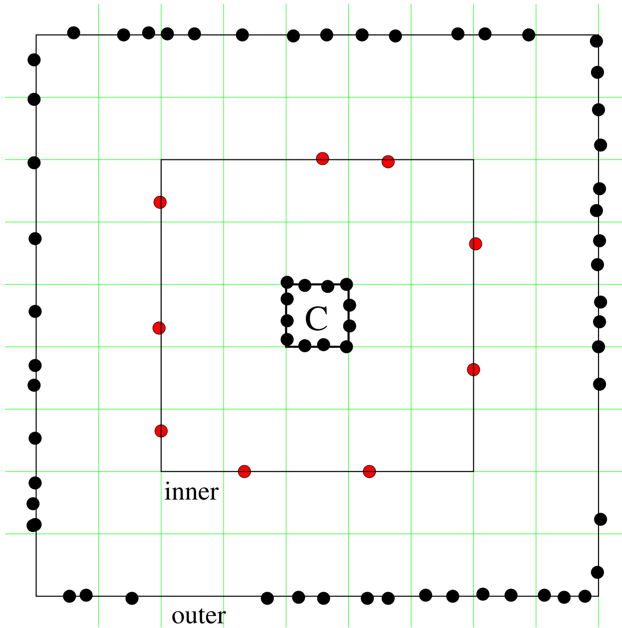


Figure 3.1.: Transit Node Construction by Bast et al. [BFM09]. The graph is partitioned into gridcells, all shortest paths starting in C and crossing the outer perimeter are covered by transit nodes.

family $\mathcal{H}$ is constructed for preprocessing how to decide at query time whether a A-B-query can be answered using the transit node scheme.

3.2. Comparison to Existing Transit Node Constructions

In the original paper on Transit Node routing by Bast et al [BFM09], the following transit node construction was used, see Figure 3.1:

1. a grid of – let's say 128×128 is put over the network

2. for each boundary node of a grid cell C, a Dijkstra is started until the nodes of the outer boundary (as in Figure 3.1) are settled

3. for each node in $v \in C$, its set of access nodes is determined by the crossing points of all shortest paths to outer boundary nodes from the boundary nodes of C

The shortest path from A to B is "long" if there are at least 4 grid-cells between A and B vertically or horizontally. This is a simplified version of the first notion of "long" via Euclidean distance. In this case for example the function $m(A, B)$ could return the the number of grid-cells in between.

In [SS09], a second incarnation of *Transit Node Routing* was presented. Here, the authors use as transit node set the (core of) a certain level in their Highway Hierarchy (HH). Highway Hierarchies are some sort of formal classification of a road network based on Dijkstra ranks. Essentially, their approach uses our third notion of "long". The idea behind Highway Hierarchies is very similar to the concept of Reach in section 2.1. For a given graph $G = (V, E)$ the authors define the concept of a 'highway network' $G_1 = (V_1, E_1)$ as the set of edges $(u, v) \in E$ that are part of a shortest s-t-path $\pi = (s, \ldots, u, v, \ldots, t)$ with the property that neither v is in the

neighborhood of s nor u is in the neighborhood of t. The 'neighborhood' of a node n is defined as the first H nodes a Dijkstra computation would settle in an one-to-all shortest path computation rooted in n. V_1 is accordingly the maximal subset of V s.t. G_1 is connected. From the highway network the authors define a 'contracted highway network' G'_1. G'_1 is obtained from G_1 by partitioning G_1 in its '2-core', which is the maximal vertex induced subgraph with minimum degree two, and 'attached trees', which are rooted in nodes of the 2-core but otherwise consist of degree 1 nodes. The contraction part consists of the replacement of all degree two chains in the 2-core with single edges, thus G'_1 consists of the contracted 2-core of G_1 and all the trees from G_1 which were not in the 2-core. Finally the 'highway hierarchy' is the the union of all the contracted highway networks created by iterating the aforementioned process. So G'_2 is computed by the exact same process as described for G'_1 but with G'_1 as base graph.

The second notion of "long" has not been used that frequently, probably because it inherits both main disadvantages of the other two notions: namely, checking for locality – that is answering whether the s, t shortest path is 'long enough' to be covered by transit nodes at all – appears similarly difficult as for the third notion and the adaptivity to varying network densities is similarly bad as for the first notion. These issues are discussed in more detail in the following.

3.3. Preprocessing

3.3.1. Primal Algorithm

The primal algorithm follows a simple greedy strategy which in each iteration adds one node to the hitting set – the one covering

most so far "unhit" sets from $\mathcal{H}$. More formally, let T_i be the hitting set after the i-th iteration, $\mathcal{H}_i := \{H \in \mathcal{H} : H \cap T_i = \emptyset\}$, then we choose in the $(i+1)$st iteration the $u \in U - T_i$ which maximizes $|\{H \in \mathcal{H}_i : H \cap \{u\} \neq \emptyset\}|$. We iterate until $\mathcal{H}_i = \emptyset$. This simple algorithm computes a feasible integral solution to the primal problem and achieves an approximation guarantee of $\mathcal{O}(\log |V|)$ since there is a generic way [Vaz01] to construct a dual feasible solution which is at most a $\mathcal{O}(\log n)$ factor away. In the following we will sketch a simple dual algorithm which in practice yields dual solutions which are very close to the primal solution of the greedy algorithm in terms of the objective function.

Algorithm 3.3.1: Primal Greedy Algorithm for the computation of a hitting set.

```
1  HittingSet(U, H)        ▷ U ... universe (= V), H ... family of sets over U
2  |  i ← 0
3  |  H_0 ← H
4  |  T_0 ← ∅                                  ▷ set of selected nodes from U
5  |  while H_i ≠ ∅ do
6  |  |  choose u ∈ {U − T_i} s.t.
   |  |  |{H ∈ H_i : H ∩ {u} ≠ ∅}| is maximized
7  |  |  i ← i + 1
8  |  |  T_i ← T_{i−1} ∪ u
9  |  |  H_i ← {H ∈ H : H ∩ T_i = ∅}
10 |  end
11 |  return T_i
```

3.3.2. Dual Algorithm

The dual algorithm follows a similarly simple greedy strategy. Our algorithm proceeds in iterations, picking it each iteration the set with the smallest weight and which does not overlap the previously picked sets. In this context the weight of a set H is defined as the number of sets in $\mathcal{H}$ which have a non-empty overlap with H. More formally let $\mathcal{P}_i$ be the set of (pairwise disjoint) sets picked after the i-th iteration $\mathcal{H}_i := \{H \in \mathcal{H} : H \cap H' = \emptyset \; \forall H' \in \mathcal{P}_i\}$, then we pick in the $i+1$st iteration a set from $\mathcal{H}_i$ of minimum weight.

At this point we have set up our basic framework for computing transit node sets with a guaranteed a posteriori approximation quality. In the remainder of the chapter we will fill in the (important) details, in particular: discuss several notions of "long" including their relation to existing transit node computation schemes and evaluate our algorithms in terms of approximation guarantee (of the transit node sets), query times (size access node sets) and efficacy (queries that can be answered).

3.3.3. Set Computation

Once the notion of "long" is fixed, the actual preprocessing step to set up the primal and dual LPs (2.1), (2.2) is quite generic: For each node A we run Dijkstra until $\forall B$ in the priority queue we have $m(A, B) \geq D$. The result is a shortest path tree, rooted at A, consisting of all nodes with distance smaller than D according to our chosen notion for "long". For each node B still present in the priority queue we trace back the shortest path to A, generating a set which is a prefix of the path from A to B consisting of the nodes closer than $L := \alpha D$ to A for some $\alpha < 1$ to make use of the travelling far and local neighborhoods observation. We expect the

Algorithm 3.3.2: Dual Greedy Algorithm for the computation of a set packing.

```
1  SetPacking(U, H)        ▷ U ... universe (= V), H ... family of sets over U
2    i ← 0
3    H_0 ← H
4    P_0 ← ∅                                ▷ set of selected sets from H
5    while H_i ≠ ∅ do
6      choose u ∈ H_i s.t. |{H ∈ H_i : H ∩ u ≠ ∅}| is minimal
7      i ← i + 1
8      P_i ← P_{i−1} ∪ u
9      H_i ← {H ∈ H : H ∩ H′ = ∅ ∀H′ ∈ P_i}
10   end
11   return P_i
```

resulting sets to be the same for many B's and ending up with few sets per node A. Several tricks like restricting the sets to nodes of degree larger 2 (since any degree 2 transit node can be replaced by the next larger degree node) can be employed to make this approach more efficient in regard to smaller sets and therefore less space consumption. See Figure 3.2 for a depiction of the situation.

3.3.4. Computation of Access Nodes

Once we have computed our set T of transit nodes, it remains to compute for each $v_i \in V$ its access nodes $\text{AN}_i \subseteq T$ (intuitively these are the first transit nodes on the few routes leaving the local neighborhood). This is a smaller hitting set problem $\text{HS}_i = (H_i^j, (\bigcup_j H_i^j) \cap T)$, where we need to hit all sets H_i^j, but are only

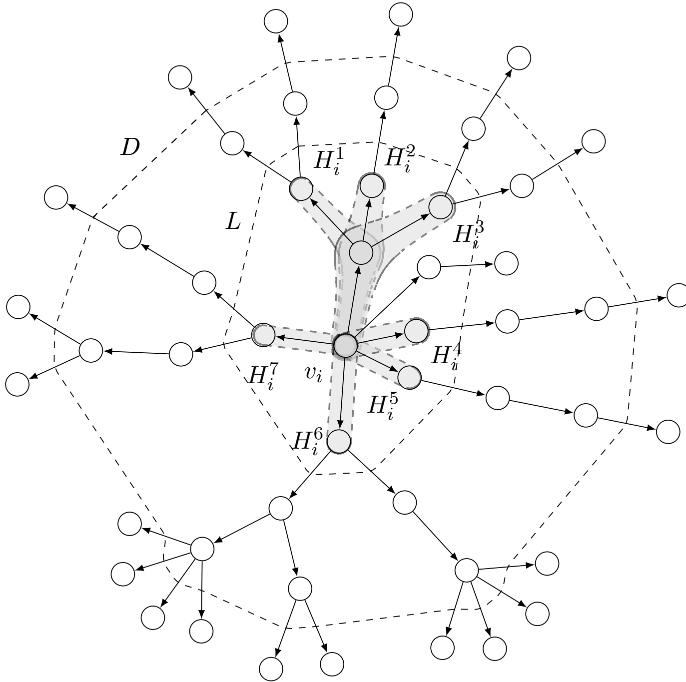


Figure 3.2.: A set is created for each distinct shortest path which crosses L and D. Here we create 7 sets.

allowed to choose Elements of T. Even if these sub problems HS_i can be quite large, we are able to solve them optimally in an efficient way, employing the special structure of the sets H_i^j. Because these sets were constructed by the shortest path tree rooted in v_i and were sub sampled by throwing away all nodes which were not chosen to be in T, we can sort their elements in increasing distance to v_i and starting with the closest mark them to be in the solution set AN_i in a greedy manner. It is easy to see that this yields the optimal solution

to HS_i as we are traversing the shortest path tree in increasing distance. This approach is similar to the one described in [SS09]. The number of access nodes compared is considerably decreased compared to the approach e.g. in [BFM09] and directly influences the query times as we will see later on.

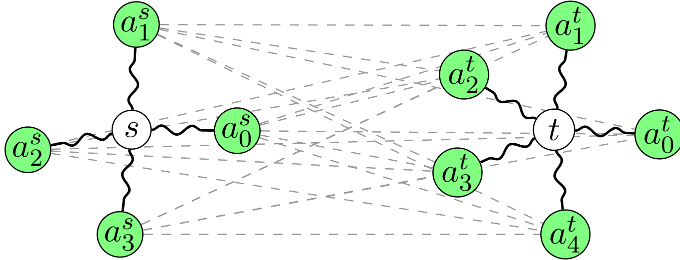
3.4. Query

For a query, we need to decide, whether vertices A and B are far apart in the distance notion m, so we aim at computing a *lower bound* for $m(A, B)$ and checking this lower bound against the parameter D.
If m is the Euclidean metric, a straightforward strategy is to use the beeline as a lower bound: $m(A, B) \geq |AB|$.
By taking into account the maximum speed in a road network, we could also turn this lower bound into a lower bound in case m is the graph distance metric. Unfortunately this does not provide good bounds as $m(A, B)$ and $|AB|$ are only loosely coupled for small transit node sets where the access nodes of a node are relatively distant, compared to the overall shortest path length. The following yields a better bound (and is also applicable to the Euclidean case). The crucial observation is that using the triangle inequality the following statement holds for each pair of access nodes: $m(AN_A^i, AN_B^j) \leq m(AN_A^i, A) + m(A, B) + m(B, AN_B^j)$. So

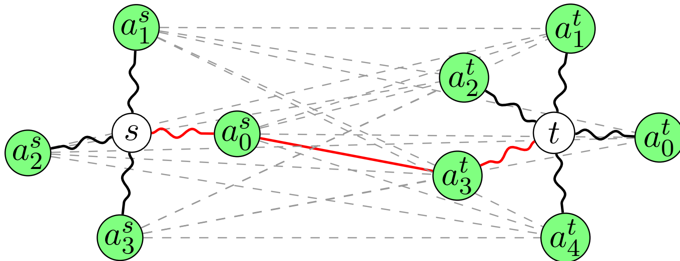
$$m(A, B) \geq \max_{i,j} \Big(m(AN_A^i, AN_B^j) - m(AN_A^i, A) - m(B, AN_B^j) \Big) . \tag{3.1}$$

Note that for directed graphs m is not necessarily symmetric, so while we use $m(AN_A^i, A)$ and $m(B, AN_B^j)$ for deriving the lower bound on $m(A, B)$, we use $m(A, AN_A^j)$ and $m(AN_B^j, B)$ for com-

putation of the distance between A and B itself. Experiments show that this lower bound is very close to the true value of $m(A, B)$.
The case that $m(x, y)$ is the Dijkstra rank of y w.r.t. x, things get more complicated, as this distance function does not satisfy the triangle inequality at all. We solve this by deriving individual constants D_v for each node v which specifies that all paths outgoing from v and longer than D_v (measured according Euclidean or graph distance) are hit by a transit node. D_v can be computed by a local Dijkstra computation during the set generation (similar to the method in [SS09]). Note that we need to compute both D_v (for the outgoing paths of the source nodes) as well as $\widetilde{D}_v$ (for incoming paths of the target nodes) in the reversed graph if we have directed edges. We then employ the same mechanism for deriving the lower bounds on the (Euclidean or graph) distance between A and B and compare this value to $\max(D_A, \widetilde{D}_B)$.
One might wonder, why we use the third notion of "long" (Dijkstra rank) at all. One important advantage of Dijkstra rank based methods is their adaptivity to varying network densities (which can also be observed when comparing Highway hierarchies vs. e.g. edge reach, [Gut04, GKW06]). In particular, using the Dijkstra rank allows to keep the number of targets t for any source node s that are *not* far away constant throughout the network. The first two notions of "long" typically result in more "non-far" targets in urban areas and less in the countryside.



(a) Computing $d(s,t)$ requires fetching their respective access nodes $A_s = \{a_0^s, \ldots, a_3^s\}$ and $A_t = \{a_0^t, \ldots, a_4^t\}$.



(b) Here the minimal distance is $d(s,t) = d(s, a_0^s) + d(a_0^s, a_3^t) + d(a_3^t, t)$.

Figure 3.3.: The basic shortest path query process once all access node sets are precomputed. If we found s and t to be far enough apart, we fetch their respective access node sets as shown in figure (a) and check all possible pairs of access node combinations. From the 4×5 possible paths we return the one realizing the smallest distance. As shown in figure (b) we return the distance for combination (s, a_0^s),(a_3^t, t) in this case.

4. Experimental Results

4.1. Implementation

We have implemented our primal and dual greedy algorithms in C++ and computed transit node sets with lower bounds for different road networks. In the following presentation we will focus on the road network of California (CA), which was also evaluated in [BFM09] and is available at the DIMACS[1] challenge website. It has 1.613.303 nodes and 3.946.702 edges. We used two versions of this graph, one bears the euclidean distances as edge costs, the other bears travel times. For some tests we also used the US network with 24.266.702 nodes and 58.098.086 edges.
Our C++ code was evaluated (unless stated otherwise) on a 24 core machine with 2 AMD Opteron 6172 cpus at 2.1GHz and 96GB of ram. Timings are in terms of CPU time, so 5 minutes on two cores are accounted as 10 minutes.

The main challenge in the implementation is the handling of the sets. In contrast to the other transit node construction schemes, they have to be explicitly built and stored for derivation of the lower bounds. To give you an idea, on the road network of California, for the parameter setting which results in a 98.59% rate of non-local queries, we had to generate 11 million sets with a 423 million nodes in total. For the US road network, a similar setting required 158

[1] http://www.dis.uniroma1.it/challenge9/

million sets with 27 billion nodes in total, pushing our server with 96GB of RAM to its limits.
Even for very small networks the (I)LP formulation becomes very large, so employing standard linear programming techniques is infeasible. Neither glpk[2] nor lp_solve[3] where able to find feasible solutions for the LP case on instances consisting of only $5 \cdot 10^5$ sets. Such a set of constraints is the result of a road network with about 10^4 nodes. Primal-dual techniques were also tried and reasonable fast but consistently non-competitive with respect quality of the outcome. With Gurobi[4] we also used a state of the art commercial LP solver, which is free for academic use. Gurobi is able to provide some very tight approximation bounds for instances up to 10^7 sets quickly (about 2% gap after a few minutes) but is not able to solve these instances exactly.

4.1.1. Instance-based Approximation Guarantees

The main result of this work is the computational proof that for the road networks encountered in practice, the transit node construction schemes that have been developed so far are essentially optimal up to a small constant factor. To that end, we have used different notions of "long" in our experimental evaluation; we computed a set of transit nodes (*primal TN size*), a feasible dual solution (*dual TN size*), their ratio and the average number of access nodes.

4.1.2. Set Generation

We implemented two different approaches for the set generation step. The first one is feasible if the amount of generated set data

[2]GNU Linear Programming Kit
[3]http://lpsolve.sourceforge.net/5.5/
[4]http://www.gurobi.com

is in the order of the available memory - in our case 96GB. In this case we assemble all the sets, generated by the Dijkstra shortest path tree in each node, in one huge set cover / set packing instance. The only optimization is to sort each set to ensure its representation is unique, sort all the sets to delete possible multiple occurrences of a set and save away which sets were found for each shortest path tree. We do this as external memory algorithms which maps all the sets into the virtual memory and uses multiple passes to sort the data and discard duplicates.
The second case is that the resulting set data is larger than the main memory by a factor of more than 3, in which case the sorting and especially solving of one giant set cover instance becomes infeasible. In this case we split up the graph in several subgraphs of equal size, generate the resulting instances separately and take care that we find a solution for each of the instances which is consistent with each other.
The splitting in our original publication was done by a simple $n \times m$ grid which divided the graph in $n \times m$ subgraphs and optimization instances. A better approach is to use METIS[5] which employs a sophisticated multilevel recursive heuristic to create subgraphs of similar size with minimal capacity between each other. Our goal is to chose the number of subgraphs such that each sub-instance fits in the fist category.
To create a valid overall solution from the sub-problems it suffices to keep track of the selected nodes in in both formulations and mark all the corresponding sets as hit in our current sub-problem in the primal version or as forbidden in the dual formulation.

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

4.1.3. Memory Management

Due to our choice of a 64bit Linux system we chose to 'outsource' the management of the large set data to the kernel. We use `mmap` to map the large set data files direct into our memory space. In this regard we only have to be able to de-/reserialize the sets, which are kept as simple vectors. In general `mmap` is the fastest way to access large files and read/write access is completely cached as long as the file size plus resident memory usage of the algorithm 'itself' is less than the main memory of the system. Above this limit the performance degrades as more and more cache misses occur. Each `mmap` cache miss results in a disk access, so the performance drastically degrades beyond a certain point.

4.2. Results

For example, the third row of the *Dijkstra rank on travel metric* block in Table 4.1 means that we consider a graph which has travel times as edge costs, the shortest path between some s and t is “long” if the Dijkstra rank of s w.r.t. t is $\geq$ 16000 and vice versa. All “long” paths emanating from some vertex v must be hit by transit nodes of Dijkstra rank $\leq$ 4000 w.r.t. v. Our primal greedy algorithm computed a transit node set of size 8996, the dual feasible solution and hence lower bound is 4886, that is, we are at most a factor of 1.85 of the optimum size for this notion of “long”. The average number of access nodes for a node was 3.67. The other rows are to be interpreted in the same manner. For any notion of “long”, the proven approximation ratio was always below 2.2.

	D	L	primal TN size	dual TN size	APX	avg. TSnodes
Dijkstra rank on euclidean metric	4000	1000	29896	15876	1.88	4.45261
	8000	2000	18299	9028	2.02	4.84460
	16000	4000	10888	5017	2.17	5.21792
Dijkstra rank on traveltime metric	4000	1000	27400	15464	1.77	3.66852
	8000	2000	16153	8822	1.83	3.67069
	16000	4000	8996	4866	1.85	3.63877
traveltime	30000u	15000u	36031	25537	1.41	3.25567
	60000u	30000u	12788	7819	1.64	3.36096
	90000u	45000u	6619	3806	1.74	3.42776
euclidean path length	15000m	7500m	32866	21074	1.56	4.87995
	30000m	15000m	12894	6836	1.89	5.43762
	45000m	22500m	7360	3492	2.10	5.59396

Table 4.1.: CA Primal and dual objective function values, approximation ratios, avg. number of access nodes for different notions of "long". D/L are the upper/lower distance bounds for the respective search spaces. 'u' is our chosen traveltime metric unit, 'm' are meters.

	D	L	primal TN size	dual TN size	APX	avg. TSnodes
Dijkstra rank on euclidean metric	96000	24000	37344	21836	1.71	3.40624
	128000	32000	27843	16064	1.73	3.43464

Table 4.2.: Same as table 4.1 for US.

4.2.1. Comparison with Previous Constructions

How does this relate with the results, for example in [BFM09]? To that end we first have to determine, what a concrete notion of "long" means in terms of the fraction of paths that are indeed "long" according to this notion. In Table 4.3 we see for example that for the scheme just explained, for 98.06% of all queries, the distance lower

bound equation (3.1) proves that we can employ the transit node scheme. In fact, the result of the transit node scheme was correct for 99.90% of the queries but we could not prove it; similar effects were also reported in [SS09].

What results are most comparable to the results in [BFM09]? According to Table 5 of [BFM09], 15087 transit nodes were necessary for a 128×128 grid on California (which equals a success rate of 97.16% – this is essentially determined by the grid dimensions) and the travel time metric on the edges. 21230 transit nodes were necessary for the euclidean metric. In terms of the construction scheme, for the travel time metric our most similar results are the (non-DijkstraRank-based) *traveltime* constructions with D value between 30000units and 60000 units. Unfortunately we cannot match exactly the success rate of [BFM09] but we would estimate that for a D value of around 40000 we would get upper and lower bounds of around 20000 and 12000. In terms of efficiency, our DijkstraRank-based construction for the travel time metric is far superior. With fewer transit nodes (8996) we obtain a considerably higher success rate (98.06%). Similar results can be observed for the euclidean metric.
Looking at the US road network in Tables 4.2 and 4.4, we see for example that for the euclidean metric we can provably hit 98.63% of all paths using 27843 transit nodes (at most a factor 1.73 above the lower bound). In [SS09], Table 3, the first layer of their transit node scheme for euclidean distance contains 15399 transit nodes and provably decides 91.2% of all queries; the average number of access nodes is 17 compared to ≈ 3.4 in our case.

Measuring the preprocessing time is difficult as our implementation is tuned for very large instances and highly dependent on the amount

cpu model	query time in μs
AMD Phenom 9850, 2.5GHz	2.25
AMD Opteron 6172, 2.1GHz	1.20
Intel i3-2310M, 2.1GHz	0.79

Figure 4.1.: Average query times for the shortest path distance computation on different processors. These timings were averaged over 10^9 random "long" queries.

of I/O. In order to provide an estimate we measured the usertime for 30000/15000m path length case on the CA graph where the set generation step took 73min and resulted in $8.5 \cdot 10^6$ sets consisting of $226 \cdot 10^6$ elements overall. The greedy computation of the primal solution took 16s, the dual one 18s. Finally the computation of the all pair shortest path distances took $5m30s$. The set computation step scales linearly with the number of nodes in the graph as with the choice of the upper and lower bound parameter and is the dominating part of the precomputation. The greedy primal/dual computation on the US instances is mainly I/O bound as the generated sets have to be retrieved from disk. The main primal/dual computation running time is in the order of several minutes compared to the I/O which requires up to one hour.

4.2.2. Query timing

As described in Section 3.4, the s, t distance calculation at this point is reduced to some lookups of precomputed values. To be more specific we need to compute minimum value of $d(s, v) + d(v, w) + d(w, t)$ over all $v \in AN_s$ and all $w \in AN_t$. This results in $|AN_s| \times |AN_t|$ lookups for $d(s, \cdot)$ and $d(\cdot, t)$ in their respective access node arrays and also in the same amount of lookups in the

	D	L	correct distance	lower bound 'far'
Dijkstra rank on euclidean metric	4000	1000	99.98%	99.62%
	8000	2000	99.94%	99.27%
	16000	4000	99.87%	98.59%
Dijkstra rank on traveltime metric	4000	1000	99.97%	99.96%
	8000	2000	99.94%	98.97%
	16000	4000	99.90%	98.06%
traveltime	30000u	15000u	99.99%	98.77%
	60000u	30000u	99.69%	94.35%
	90000u	45000u	99.19%	92.25%
euclidean path length	15000m	7500m	99.90%	99.29%
	30000m	15000m	99.58%	97.57%
	45000m	22500m	99.31%	96.67%

Table 4.3.: CA *Correct distance* is the measured success rate where applying the transit node scheme results in the correct shortest path distance. *lower bound* is the fraction of nodes where we could actually prove correctness of the transit node scheme using equation 3.1. D/L are the upper/lower distance bounds for the respective search spaces.

	D	L	correct distance	lower bound 'far'
Dijkstra rank on euclidean metric	96000	24000	99.998%	98.63%
	128000	32000	99.97%	98.57%

Table 4.4.: Same as table 4.3 for US.

all-pairs shortest path table of the access nodes for the respective value of $d(v, w)$. Hence the number of access nodes is the key factor which determines the query time.
In [BFM09], a (admittedly not very sophisticated) strategy for determining access nodes resulted in 9 (16) access nodes per node on

the average in the travel time (euclidean) metric graph. The last column in Table 4.1 shows the avg. number of access nodes. We obtain around 3.6 (5.2) access nodes on the average for the travel time (euclidean) metric. This results in roughly 13 (27) lookups instead of 81 (256) which, of course, is also reflected in the actual query times.
The time needed for these lookups is dominated by the access of the all-pairs shortest path array as the access pattern of the values $d(v, w)$ will most often result in cache misses. In contrast the distance values of $d(s, \cdot)$ and $d(\cdot, t)$ are hold in one contiguous small array for s and t respectively. So it is not surprising that the real worlds query timings are highly dependent on the cache hierarchy of the chosen processor, the timings in Table 4.1 were measured for the CA graph with an average of 3.62 access nodes per node. In any case, the results are considerably faster than the times reported in [BFM09]; here for the travel time metric, average query times were around 8.9μsecs on a 2.4 GHz Opteron processor, even though comparison of timings on different machines are doubtful. The size of the access node sets (and hence the number of lookups) is a much better, architecture-independent indicator. The access node sizes in [SS09] were better (6.1 on the US for travel time, 17 on the US for euclidean) but still above the numbers we have experienced in our current construction scheme. Note that we sampled “long” queries exclusively, as we don’t use any fallback mechanism if the chosen path cannot be predicted by our transit node framework in this setting.

In more recent work [ALS13] the authors review different transit node contraction methodologies and also present a new approach where the TN candidates are the k most important nodes of the contraction hierarchy. The authors analysis main emphasis is query

wall time to which regard they optimize for memory access, locality and cache misses. In all these categories the TN approach seems to be an excellent contender as the CH computation and query engineering approaches can easily be extended to the TN setting. As the resulting transit node set size is a freely chosen parameter, the authors just fix k to be comparable to other TN construction approaches and report 8.5 and 7 access nodes per node for the resulting TN sets on a DIMACS map of Western Europe with 18m nodes and 22.4m edges. With transit node sets of 10k, 24k and 28k nodes the authors report the fraction of local queries as 0.58%, 0.17% and 0.14%.

5. Discussion and Extensions

In this chapter we have presented computational proof that the known transit node constructions like [BFM09] or [SS09] produce transit node sets which are close to optimal in size. Such *lower bounds* were not known before; we have proven computationally that for the problem instances commonly considered, it is *not* possible to construct transit node sets of – let's say – 1/20th the size. We do not recommend our preprocessing scheme for practical purposes as it is very time and space consuming, but should rather be seen as a computational proof that the known schemes are not that bad after all. In future work we will try to optimize our implementation such that approximation guarantees can be derived for even the largest networks like the US or the whole of Europe, even though we do not expect substantially different results there.

Some efficacy is lost in the case of splitting the problem in sub-instances. It would be interesting to see whether the usage of a solid state disc results in faster `mmap` accesses even under severe memory pressure. This would simplify our implementation significantly and allows straightforward reasoning about the correctness of our solutions. The limiting factor is the random read access on the sets by the greedy solver and the resulting random disc reads with their latency penalty.

Part III.

Applications of speedup Techniques: Path Prediction

1. Motivation

Imagine a setting in which an autonomous agent is traveling on a road network and there is a need to synchronize the agents' exact location in the network with an external observer. As to illustrated in Figure 1.1 the observer is informed about the starting point of the agent and the route it took so far, but has no knowledge about the agents intended destination or the actual route the agent will use on its journey to get there. Since travel is bothersome and costs time as well as resources, both agent an observer agree on using shortest (or quickest) paths. If this is too much of a constraint the agent will at least travel on piecewise shortest paths.
The need of the observer to be informed about the agents' position at all times requires communication between both parties. On the other hand our model assumes that such kind of communication may be slow or expensive and should be avoided at any cost. So our optimization goal is to ensure a consistent world view for both of them with a minimal amount of synchronization events where the agent has to inform the observer about its changed course.
In order to render such kind of updates possible both of them will have to follow the same model about the agents' intentions which empowers them to predict the agents' movement at each and every intersection on its path. The agent will then communicate to the observer every time it violates their agreed upon model and both will then draw the appropriate conclusions from this update. This framework is also known as *dead reckoning protocol*.

This problem can be formalized as follows: Given a road network $G = (V, E, \gamma)$ and a path $\pi = (v_0, v_1, \ldots, v_i)$ defined by nodes in G we want to predict how π continues in G, that is we'd like to compute $\pi' = (v_0, v_1, \ldots, v_i, v_{i+1}, \ldots, v_k)$. For measuring the performance of our proposed models or strategies we will choose random paths $\mathcal{R} = (v_0, v_1, \ldots, v_k)$ and provide the partial Path $\mathcal{R}' = (v_0, v_1, \ldots, v_i)$ for all $0 \leq i \leq k$ as input for the respective prediction of v_{i+1}. A prediction error will result in a mismatch of the node v_{i+1} in $\mathcal{R}$ and the proposed v'_{i+1} by the predictor. We will always start with the assumption that the agent starting in v_0 moves in such a way that π is shortest path or at least exhibits piecewise shortest path property where the agent does add some detours to its destination.

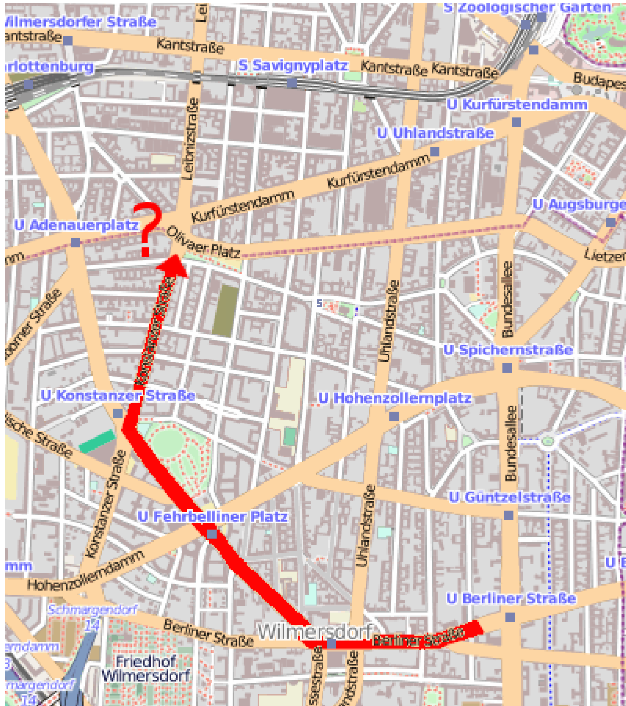


Figure 1.1.: The Path Prediction Problem: We are given the route a mobile user has travelled up to now and want to predict its route in the near future.

1.1. Related Work

In [LNR02] the authors note that in general road intersections are designed with the intention to admit unhindered travel on lanes with higher density travel. To that end the more important road is straight and only changing to the smaller road requires a right or left turn or

the use of an exit lane. So they propose to inspect the bearing of the road segment (v_{i-1}, v_i) and predict the node v'_{i+1} which realizes the smallest change of absolute bearing if the agent would take the segment (v_i, v'_{i+1}).
Intuitively this assumption makes sense for long segments of non-urban travel where highways or expressways most often realize the shortest path but breaks down completely in dense urban road networks with large numbers of equidistant similar alternative routes.
We will use this strategy as baseline and categorize it later on in section 3. In addition we will also adapt two of the authors prediction quality metrics, as will be discussed in section 2.2.

1.2. Outline

In the following sections we will first present the concept of *reach* which represents an important graph metric and will help us to quantify the importance of an edge in our setting. We will present an approximation algorithm for the computation of lower bounds for edge reach values. This enables us to compute this metric even on continent sized road networks.
We will discuss the different quality metrics we devised to compare our prediction strategies.
Finally we will outline the prediction strategies themselves and motivate the decision to consider two different settings in the form of online and offline strategies and their applications.

2. Basic Concepts

2.1. Reach

For our more advanced strategies in section 3 we will need the concept of *reach* which was first formalized by [Gut04]. Intuitively reach is a quantity which assigns an importance to each node $v_i \in V$ in a graph $G = (V, E, \gamma)$. To be more precise the reach of v_i considers all shortest paths in G of the form $\pi = (v_0, \ldots, v_i, \ldots, v_k)$, containing v_i and reflects the largest value of $\min_\pi(d((v_0, \ldots, v_i)), d((v_i, \ldots, v_k)))$. That is the reach of a node v_i is large iff there exists a shortest path π, containing v_i near its midpoint, thus far away from its source and target. The reach of v_i is small if there exists no such long path or v_i is near the end of all such paths. This is the case if v_i is close to a dead end for example.
The authors suggest that this value also conveys a metric for the importance of each node. A high reach value implies that there are many shortest paths in the network which contain the respective node so this node would be highly frequented by traffic in the network. This concept is of importance for our prediction models later on. Under this assumptions it comes natural to predict a node with high reach as most probable successor.

From the definition of reach we can directly induce an algorithm to compute all reach values for a given graph $G = (V, E, \gamma)$. For each node $v_i \in V$ we have to generate all shortest paths with v_i as source.

For each of these paths we follow its course updating the reach for every visited node. For every node on a given path we keep track of the distance to v_i and to the path's target. If the minimum of these two distances is larger than the old reach value we update the respective node. The pseudo code for this computation is given in algorithm 2.1.1.
The simplicity of this algorithm comes at a cost. In particular its runtime. Under the reasonable assumption to compute a single one to all shortest path trees with Dijkstra's algorithm in $\mathcal{O}(n \cdot \log n + m)$ time and to traverse it, such that we can update all the reach values in $\mathcal{O}(n)$ for this tree, the complete algorithm will take $n \cdot (\mathcal{O}(n \cdot \log n + m) + \mathcal{O}(n)) = \mathcal{O}(n^2 \log n + n \cdot m)$ time. The algorithm actually computes the all pair shortest paths in G which is infeasible even for moderately sized road networks.

Algorithm 2.1.1: Naive Reach computation.

```
1  NaiveReachComputation(G = (V, E, γ))
2  foreach v_i ∈ V do
3      reach(v_i) ← 0
4  foreach v_i ∈ V do
5      foreach shortest path π starting in v_i do
6          s ← v_i
7          t ← target of π
8          foreach v_k ∈ π do
9              reach(v_k) ← max(reach(v_k),
10                                 min(d((s, ..., v_i)),
11                                     d((v_i, ..., t))))
12 return reach(v_0, ..., v_N)
```

In order to cope with the problem of excessive runtime we relax our previous demand for exact reach values for all the nodes and compute lower bounds instead. If we cannot argue about the exact reach or implied importance of a node we may at least certify that a node is more important than a threshold. To this extend [Gut04] developed an approximation algorithm of which an modified and simplified version is described below.

The most important difference to the authors original work is that we compute the reach for edges instead of nodes. This transfer is straight forward as the reach of an edge can be naturally defined as the smaller one of its two incident node reach values. For a given shortest path $\pi(s,t) = (s, v_1, \dots, v_h, t)$, when we compute the distances from s, we call a node v_i Δ-far iff v is settled and $d(v_1, v) \geq \Delta$. For given values $0 \leq \alpha \leq 0.5$, Δ we say an edge $e = (p, q)$ is (α, Δ)-important iff (p, q) lies on a shortest path from some node s to some other node t with $d(s,t) \geq \Delta$ and $d(s,q), d(p,t) \geq \alpha\Delta$. We now want to compute all (α, Δ)-important edges and show that this metric is actually an approximation for the true reach value of each edge. You may also refer to figure 2.1 or 2.2 for some examples of this concept.

Algorithm 2.1.2 computes an shortest path tree with an radius slightly larger than Δ around every node in the Graph and uses the resulting shortest paths as witnesses for the (α, Δ)-importance of all edges which lie $\alpha\Delta$ far away from both ends of such an path. It can be shown that this is correct as long as $\alpha < 0.5$ holds as in this case the algorithm always finds a long enough prefix of an (s,t) shortest path which witnesses the importance of a given edge.
This sub procedure is now used to iteratively grow larger and larger shortest path trees for every node in the graph, very much as in the

Algorithm 2.1.2: Edge Lift helper Function.

```
1 LiftEdges(G = (V, E, γ), α, Δ)
2 foreach v_i ∈ V do
3 |   grow a shortest path tree rooted in v_i until the
|   predecessors of all active nodes are Δ-far
4 |   foreach edge (p, q) on a shortest path from v_i to tome
|   Δ-far node z do
5 |   |   if d(v_i, q) ≥ Δα ∧ d(p, z) ≥ Δα then
6 |   |   |   mark edge (p, q)
7 foreach e_i ∈ E do
8 |   mark e_i as (α, Δ)-important if it was marked above at
|   least once
```

naive version but with the notable improvement that we can prune away edges which are not important enough without affecting the correctness of the result.

Algorithm 2.1.3: Compute edge levels.

```
1 ComputeEdgeLevels(G = (V, E, γ))
2 H_0 ← G for i = 1 to log_2(M/δ) do
3 |   LiftEdges(H_{i−1}, α, Δβ^{i−1})
4 |   H_i ← subgraph of H_{i−1} induced by all lifted edges
|   from previous call
```

Algorithm 2.1.3 now uses the former one to compute all $(\alpha, \Delta\beta^{i-1})$-important edges in round i. It can be shown that it suffices to only consider edges which were lifted in the last round to compute the next one. This has the profound effect that although the radii in

each round grow larger and larger we are able to mask out most of the edges of whom we already know that they are not part of an longer shortest path. We chose $\beta = 2$ and $\alpha = 0.25$ for our reach computations so each lower bound is actually a 2APX of the real reach value.

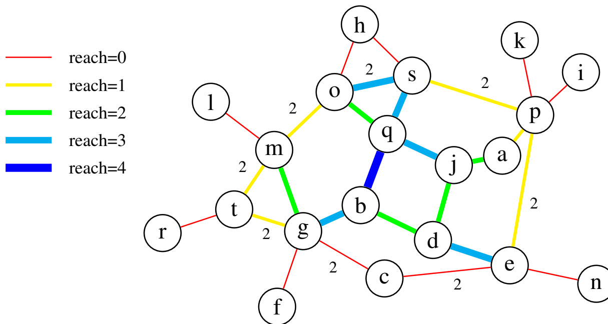


Figure 2.1.: A small example Graph with the respective reach values for each edge. The edges on the outer "ring" have a weight of 2, all other edges have a weight of 1. The edge (b, q) has the highest reach, which is realized by the path (r, t, g, b, q, s, p, i).

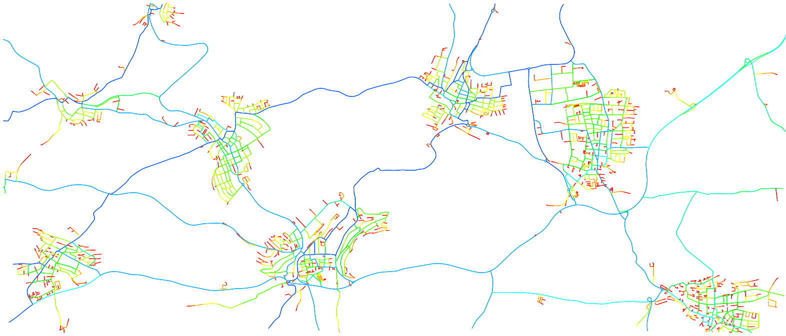


Figure 2.2.: A real example with edge reach coded as color. The color grades are separated by a factor two, that is red corresponds to a reach value of $[0, 100]$m, orange to the interval $(100, 200]$m and so on. The reach was computed with our methodology - as an edge property.

2.2. Quality Metrics

As described in section 1 with a piecewise shortest path $\pi = (v_0, v_1, \ldots, v_k)$ and a given prefix $\pi' = (v_0, v_1, \ldots, v_i)$ a prediction error occurs every time our model predicts a v'_{i+1} which is not the actual predecessor of v_i in π. As it turns out, the probability for these mismatches and therefore the error rate of any model is not only dependent on the power of the model itself but even more in the relative position on π, that is the ratio of i to k. This is natural, since near the end of a path, the final destination could be essentially around 'any' corner. On the other hand, very short prefixes do not convey much information. To this extend all our measurements do have in common that a large number of random paths π, which may obey further limitations, are drawn with the intention of evaluating each strategy on an 'average' path in G.

The *expected distance between failure* metric uses paths of the same length and maps the quantized relative position on a path in percent ($\frac{i}{k} \cdot 100$) to the expected distance of the next prediction error along with its standard deviation. So if you know your relative position on a concrete path and use a specific prediction strategy, this metric will tell you how long a correctly predicted path fragment you can expect at that position. A superior strategy naturally exhibits larger distances, in fact the optimum value is the remaining path length. Therefore this value converges to zero at the end of a path.
The *absolute number of errors* metric uses the same quantized relative path position and maps the absolute number of prediction errors which occurred at this relative path point. A superior strategy will not only exhibit a smaller overall sum of this values but different strategies also result in different profiles along the path.
Note that all strategies implicitly assume the ground truth to be a path of infinite length, so the error rate in the second half is likely to be higher. π diverges drastically from its infinite modes as one get closer to the real destination. As mentioned before, though, it is natural that it is hard/impossible to make any educated guess about the future path when being close to the final destination as this could be around any corner.

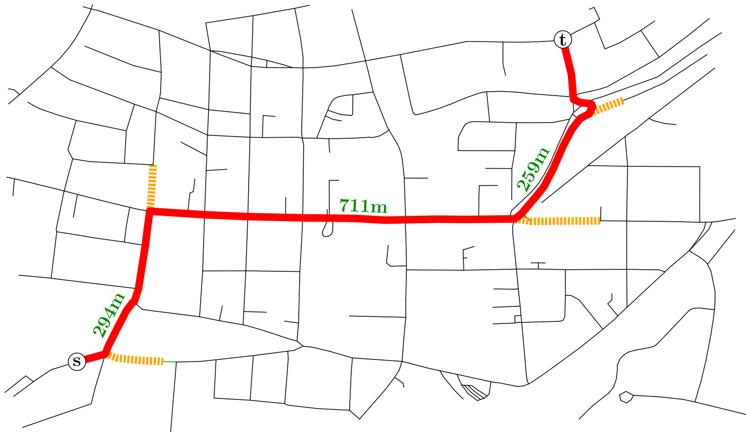


Figure 2.3.: An example shortest path from s to t with possible mispredictions in orange. The expected distance between failure metric would average over the length of the three middle segments between prediction errors. They have a length of 294, 711 and 259 meter in this case. The absolute number of error metric would just count the four prediction errors.

3. Path Prediction Strategies

We categorize our proposed strategies into two main classes, according to the limitations they have to obey at query time.
Offline strategies may employ extensive precomputation but the stored precomputation data is restricted by a linear space bound with regard to the size of the graph. At query time, an offline strategy is allowed to spend time proportional to the maximum degree of the graph. In particular, offline strategies are not allowed to start complex graph explorations.
Online strategies on the other hand have to obey the same precomputation and storage limitations as offline strategies but are allowed to perform extensive computations at query time, in particular (partial) Dijkstra computations are allowed. We make this distinction to differentiate between algorithms that can potentially be employed on the simplest mobile devices and such that require at least some computing power at query time. For the description of the employed strategies assume that we choose a path $\pi = (v_0, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, v_k)$ with the intention of predicting v'_{i+1}, so a prediction strategy is given $\pi' = (v_0, \ldots, v_{i-1}, v_i)$ as input (an offline strategy is given a constant-sized suffix thereof). Observe that the prediction of degree 2 nodes is trivial in this context. Nodes of degree 2 are therefore ignored.
At this point the task of predicting a path trajectory breaks down to the question of estimating the one outgoing edge of v_i we think of being the most likely one to continue the path.

3.1. Offline Strategies

In [LNR02], a very straightforward strategy – which we refer to as *baseline* strategy (OF_b) – was proposed. When coming from vertex v_{i-1} and being at v_i we pick as next edge/vertex the outgoing edge from v_i, $\overrightarrow{v_i v'_{i+1}}$ with minimal change of direction compared to $\overrightarrow{v_{i-1} v_i}$. Clearly, this is an offline strategy according to our categorization. The intuition behind this strategy is that shortest/quickest paths tend to be rather straight.
A slightly more involved strategy which we refer to as *simple Dijkstra* (OF_sD) works as follows: In a precomputation step we compute a full Dijkstra from each vertex v in the network and remember for each outgoing edge of v how large a subtree (in terms of # of nodes) is hanging below this edge. Clearly this is very time consuming but requires only linear space. At query time, when being at node v_i we choose the outgoing edge *not* leading to v_{i-1} which bears the largest subtree in the shortest path tree from v. Conceptually this is close to choosing the edge where most likely a target chosen uniformly at random in its shortest path subtree resides.
To reduce the enormous computational cost (even though this can be easily parallelized and computed in a few days on a small cluster also for large networks like the US or the whole of Europe), we can slightly modify the precomputation step and start an unidirectional reach-[Gut04]-based Dijkstra at every $v \in V$ and order the outgoing edges of v according to the *longest* path discovered during these searches. This reduces the precomputation time by orders of magnitudes without really affecting the quality of the prediction as we will see.
As we employed a reach-based search, we refer to this strategy as *simple reach based Dijkstra* (OF_rbD). As this strategy turned out to be essentially equal to OF_sD, we have omitted the latter in our

experiments. For the precomputation time, we are in the range of a few milliseconds per vertex (on a standard PC, not including the precomputation time for the reach or HH information itself), so even large networks can be preprocessed in a few hours.
Having introduced the reach concept, another obvious strategy is to simply return the edge with highest reach (see Section 2.1) which does not lead back to v_{i-1}. This we call the *reach based* strategy (`OF_rb`). Intuitively, high reach means important edge whereas low reach means unimportant edge.

3.2. Online Strategies

Under the assumption that the ground truth path π is a shortest path from s to some random t, the arguably optimal strategy constructs a shortest path tree (by Dijkstra) starting in s. At any point in time, the known path segment $v_0, \ldots, v_i$ is part of this shortest path tree, and as t is randomly chosen, picking the outgoing edge of v_i which contains the most nodes in its subtree is the optimal prediction strategy. We call this the *full Dijkstra* (`ON_fD`) strategy. Apart from being quite demanding computationally at query time (a full Dijkstra computation), this strategy breaks down if the path π is not a shortest path but consists of *piecewise* shortest paths only – think of stopping by the bakery, the grocery, and the butcher on your way home from work.
To cope with this problem, we need to detect when the path $(v_0, \ldots, v_i)$ leaves the shortest path tree rooted at $s = v_0$. If this happens at v_i, we start a Dijkstra[1] at v_i exhibiting the longest suffix of $(v_0, \ldots, v_i)$ that is (in reverse order) a subpath in the shortest path tree rooted

[1] If the graph has asymmetric edge costs, we run this Dijkstra on the reversed graph.

at v_i. Let $(v_k, v_{k+1}, \ldots, v_i)$ be this suffix (note that we can abort Dijkstra if we have settled v_{k-1} in a different subtree of the shortest path tree). We then start a (full) Dijkstra in v_k and use this for prediction. We call this strategy *lazy full Dijkstra* (`ON_lfD`). Due to space restrictions and the still somewhat high computational demand at query time, we will not report results on that but on the following variant thereof.
Similar to our heuristic offline strategy we simply replace the full Dijkstra computation by a reach-based Dijkstra computation which prunes out the edges with increasing distance from the source. So as long as we move on a path of the shortest-path tree of the reach-based Dijkstra started in v_0, we always pick the edge leading to the 'furthest' node. If we leave the shortest path-tree at v_i we start a backward reach-based Dijkstra starting in v_i to exhibit a suffix $(v_k, \ldots, v_i)$ as before and use a reach-based Dijkstra rooted at v_k for the further prediction. The next time we leave this shortest path tree, we again start a search for such a suffix. This strategy – which we call *lazy reach-based Dijkstra* (`ON_lrbD`) – allows both for adaptivity in case of piecewise shortest paths as well as good running times due to the heavy pruning of edges in the Dijkstra computation.

4. Experimental Results

Our two test graphs are based on OpenStreetMap[1] [OSM] data which was stripped of all features impassable by car. For the two resulting graphs with 321k and 18.57M vertices respectively, we computed the reach of all edges based on a travel time metric as proposed in [Gut04]. From the extensive meta data of the OSM dataset we picked the street type to derive a speeds for individual edges. The larger graph [GER] represents a street map of Germany, the smaller one [MV] of its federal state Mecklenburg-Vorpommern. Due to the fact that part of the OSM data is generated by GPS plots, each road segment is composed of a larger amount of degree two nodes. The average chain length is $\approx$10.7 segments on [MV] and $\approx$10.2 on [GER] which corresponds to an average length of less than one kilometer. As direct result $\approx 80\%$ of both graphs nodes are of degree 2.

4.1. Quality

For the *expected distance between failure* metric the random shortest paths π were sampled with their length being limited to the interval of $100\,\mathrm{km} \pm 2.5\%$ in the case of [MV] and $500\,\mathrm{km} \pm 2.5\%$ in the case [GER]. For the other metric a lower bound of 150 nodes resulted in minimal path lengths of $\approx 15\,\mathrm{km}$, being only limited upwards by the respective graphs diameter of $\approx 331\,\mathrm{km}$ for [MV] and

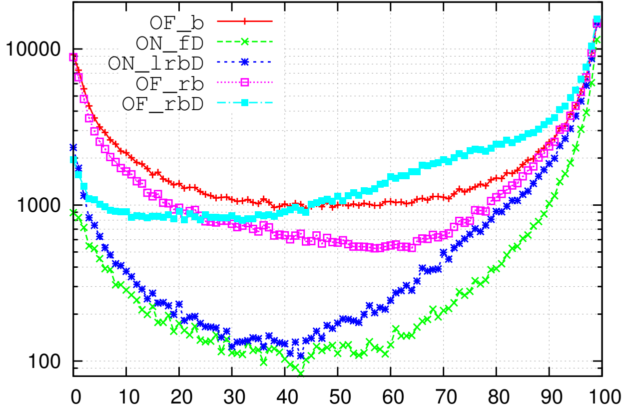
[1] http://download.geofabrik.de/osm/

$\approx$1044 km for [GER].
In Figure 4.1 the *absolute number of errors* metric was employed for 25k random shortest paths on [MV] and 10k random shortest paths on [GER]. Common to all strategies – be it on- or offline – is the fact that close to the start and the end, the prediction quality is pretty bad, which was to be expected. Furthermore – no surprise! – online strategies are far more accurate than offline strategies. Amongst the offline strategies, the baseline predictor (`OF_b`)fares worst on the first half, leading to a total prediction failure rate of 6.25% on [GER], that is, on 6.25% of all nodes on the path *with degree larger than 3* a prediction error has occurred. Employing the reach-based Dijkstra strategy (`OF_rbD`) we obtain much better results at the beginning of the paths but getting worse towards the end, resulting in a 5.98% failure rate overall. The purely reach based strategy `OF_rb` uniformly exhibits better performance than the baseline strategy and except for the beginning also than `OF_rbD`, its total failure rate is 4.03%. It is natural to combine the purely reach-based strategy with the reach-based Dijkstra strategy for the best offline prediction rate. Unfortunately, it is not so clear a priori when one gets better than the other; in case of [MV] this point was at around 22% of the path length, in [GER] it was around 15%. With some tuning we were able to find mixing parameters such that the prediction failure rate of such a mixed offline strategy became 3.21%.

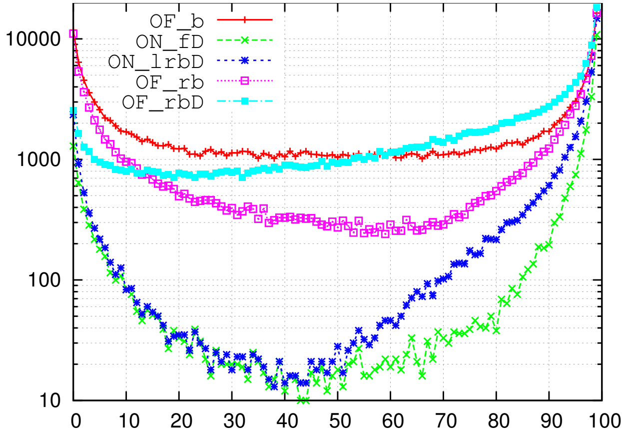
Our online prediction strategies perform much better. Given that we are working with single shortest paths only, the *full Dijkstra* strategy (`ON_fD`) yields an almost perfect result with a total failure rate of 0.95% only. The more practical *lazy reach-based Dijkstra* strategy is not much worse with a total failure rate of 1.59%. In general, the results for [MV] and [GER] are comparable.

Finally we consider the *expected distance between failure* metric, which is arguably the most intuitive one, in Figure 4.2. For each strategy we have depicted the expected length of a correctly predicted path chunk when being at a certain relative position, comparing our two online strategies with the baseline strategy and the simple reach-based Dijkstra offline strategy. It is clear that our online strategies are far superior to the baseline strategy. In this metric the choice of the underlying road network makes a bigger difference. Both online strategies are gaining a significant amount of accuracy on the [GER] graph, to the point that the average distance to the next prediction error is always more than $\approx 80\%$ of the remaining path length, compared to $\approx 60\%$ on the [MV] graph. In contrast to that, the *baseline* strategy's average prediction distance is always below $50\,\mathrm{km}$ or 10% on the [GER] graph compared to $20\,\mathrm{km}$ or 20% on the [MV] graph. The other reach-based offline strategy performs slightly better but is still way below our online strategies.

In real-world scenarios, the trajectories of mobile users are often not exact shortest s-t-paths but they tend to be composed of several shortest-path segments (from work to the bakery, then to the butcher, then to the florist before driving home). In Figure 4.3 we present results for such problem instances constructing a series of $50{+}90{+}30{+}100{+}70\,\mathrm{km}$ shortest path pieces resulting in an overall length of $340\,\mathrm{km}$. Both *lazy reach-based Dijkstra* approaches and the *baseline* approach are producing characteristic profiles along each shortest path piece being limited by the distance of the actual piece ending, while again the latter approach is far inferior in absolute performance. The *full Dijkstra* approach on the contrary completely fails to produce reliable predictions behind the first shortest path piece as the available shortest path information at v_0 is meaningless from there on.

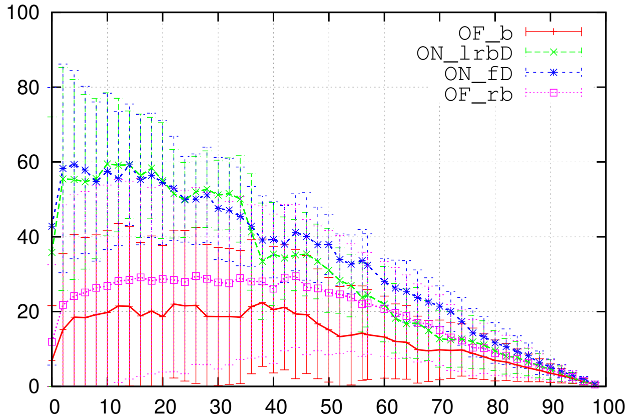


(a) Absolute amount of prediction errors by the respective relative position on the path for MV, sampled on 25K paths

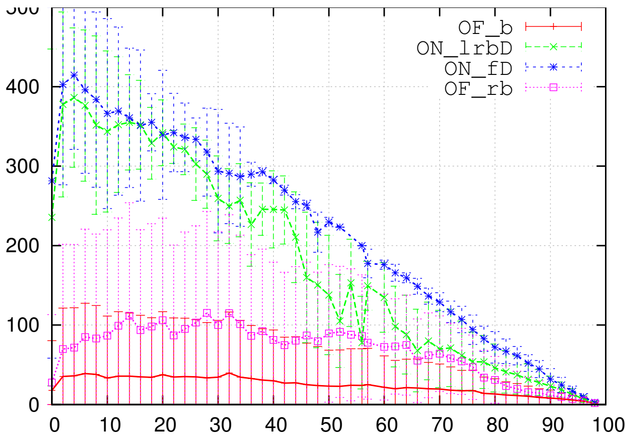


(b) Absolute amount of prediction errors by the respective relative position on the path for GER, sampled on 10K paths

Figure 4.1.: ***Absolute number of errors*** **metric. For fig. (a) 25k random paths on [MV] graph were sampled. Accordingly 10k paths on the [GER] graph for fig. (b).**



(a) 25k sampled paths of 100km length on [MV]



(b) 10k sampled paths of 500km length on [GER]

Figure 4.2.: The *expected distance between failure* metric is limited by the remaining path length. The *full Dijkstra* strategy shows nearly optimal results.

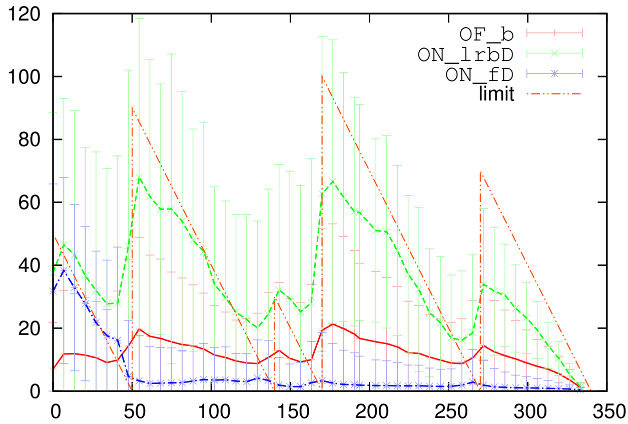


Figure 4.3.: ***Expected distance between failure* metric for series of piecewise shortest paths with a length of $50 + 90 + 30 + 100 + 70 = 340\,\mathrm{km}$. The “limit” line plots the remaining length of the actual shortest path.**

4.2. Cost

Let us briefly review the cost for predicting trajectories at runtime; we focus on the number of Dijkstra operations as this is determining the real-time applicability of our algorithms. We count the total number of Dijkstra operations that were performed while running our prediction strategies on a shortest path of 500km length.

strategy	# Dijkstra polls
`all Offline strategies`	0
`ON_fD`	$\approx 18.5 \times 10^6$
`ON_lrbD`	$\approx 6.32 \times 10^5$

Table 4.1.: Computational cost per path for different strategies in terms of Dijkstra polls on the [GER] graph at prediction time. Offline strategies only spend constant time and no Dijkstra poll at runtime, our most competitive online strategy is `ON_lrbD`. Average over 10k random shortest paths of length $\approx$ 500km.

The good performance of `ON_lrbD` in Table 4.1 is due to two reasons: first, a single reach-based Dijkstra visits only a small fraction of the nodes compared to a full Dijkstra, furthermore, only few prediction requests trigger a reach-based Dijkstra computation. More concretely, 99.25% of all prediction requests in our experiment can be answered using the already existent shortest path tree. Only for the remaining 0.75%, a reach-based Dijkstra has to be started, resulting in the total number of polls as shown in Table 4.1.

Strategy `ON_lrbD` in total for the *whole path* uses less than one tenth of the Dijkstra polls that are necessary for a full Dijkstra on the network (which equals the strategy `ON_fD` which is much worse

for only piecewise-shortest paths, though). Again, if we assume the car to drive at about 100 km/h (so the trip takes $\approx$ 5 hours), a single core can perform predictions for more than ten thousand vehicles at the same time. Hence with a not too expensive compute server with let's say 100 cores, we can easily predict trajectories for one Million vehicles in real-time.

5. Discussion and Extensions

One immediate application for path prediction methods is in the context of dead-reckoning protocols. Here a mobile user is to periodically send its position to a server. When the required bandwidth for transmission is an issue, predicting the motion of a user both on the client/user side as well as the server side can result in considerable savings by only transmitting deviations from the prediction as was shown in [LNR02]. With our improved prediction method, these savings can be even further increased. Navigation systems, in particular the ones built-in by car manufacturers, have become 'always-on' devices, so even if no target has been given by the user and no route planning takes place, they acquire the current position. Accurate path prediction routines allow the device to call attention to points of interest that lie on the predicted route. Examples are gas stations, restaurants, shopping malls, parking lots but also areas of difficult road conditions or traffic jams.

The combination of Map Matching, which was also part of the original publication in [EFH+11], and Path Prediction also makes a lot of sense, in particular when the mobile device is not equipped with GPS: for a given sequence of location measurements our map matching routine identifies an initial path, which is then continued via path prediction routines. As map matching under very imprecise location information tends to be inaccurate for the first few and the

last few measurements, one would discard those parts and employ our path prediction routines on the remaining path fragment. This could be the basis for a rudimentary navigation system that works only based on GSM location data.

So far we have only considered the problem of predicting the path. A natural extension is to compute a mapping Time$\rightarrow \mathbb{R}^2$, that is, predict at what time one expects the mobile user to be at what position. This has to take into account both existing speed-limits on the road segments of the predicted route as well as the observed driving speed within the known trajectory. This extension seems to be natural as the protocol of 'synchronization points' could be used to synchronize a model of travel speed – in addition to position – in the exact same fashion as discussed so far.

Part IV.

Applications of speedup Techniques: Sequenced Route Queries

1. Motivation

When heading back home from work there are often things to do on the way like grocery shopping, getting cash from an ATM, refueling at a gas station, or dropping off a parcel at a post-office. We consider the problem of planning an optimal route (quickest or shortest) that visits facilities of the respective type on the way home. The proposed solution based on the combination of a distance sensitive doubling technique and contraction hierarchies is orders of magnitudes faster than either a naive approach or previous results and produces the answers *in an instant* for realistic queries without compromising guaranteed optimality. With such fast query times, this type of route query becomes feasible even on mobile devices or for high-throughput web-based route planners.

We are given a graph $G(V, E, \gamma)$ with edge costs $\gamma :\rightarrow \mathbb{R}$ and a collection $\mathcal{C} = \{C_1, C_2, \ldots, C_k\}$ of facilities with $C_i \subset V$. For example, G could be the road network of Germany, w the travel times on the road segments, C_1 the locations of all gas stations in the network, C_2 the locations of all ATMs, etc.
A *query* is specified by a source s and a target t as well as a sequence of facility classes $(p_1, p_2, \ldots, p_l)$. We are interested in finding the shortest path from s to t in G visiting a facility in C_{p_1} followed by a facility in C_{p_2} ..., followed by a facility in C_{p_3}. This type of query is referred to as *sequenced route query* in the literature. Answering such a query allows us to find for example the fastest route

home from work visiting an ATM, a gas station and a post-office. Note that we primarily consider the variant where the order in which the facilities have to be visited is *fixed*. Dropping the restriction on the order essentially turns this problem (for non-constant l) into the NP-hard travelling salesperson problem. On the other hand, in most practical scenarios, l is rather small, and as our query procedure for fixed order turns out to be very efficient, a brute force exploration of all possible orders is actually possible.

For the remainder of this chapter we consider the following setting: the graph G is the road network of Germany derived from data of the OpenStreetMap (OSM) project [OSM] consisting of about 15 Million nodes and 30 Million edges. Edge costs are travel times calculated based on the Euclidean distance and the road type. The facilities $\mathcal{C}$ are also derived from OSM data, for example we extracted 15.9k gas stations, 18.5k bakeries and 21.4k grocery stores.

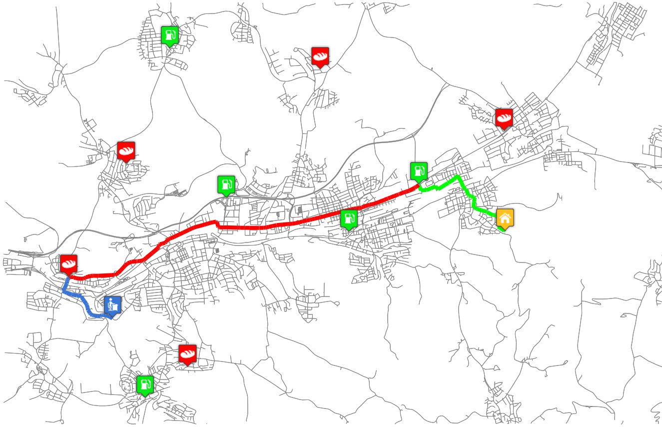


Figure 1.1.: Finding the way home - a shortest route from the office (blue) frequenting a bakery (red) and a gas station (green) to our home (yellow).

1.1. Related Work

Sequenced route queries have appeared in several contexts in the literature. In [SKS05][1], the authors consider sequenced route queries in Euclidean space and describe an approach called the EDJ algorithm which creates for a sequenced route query $(s, t, p_1, \dots, p_l)$ a directed, acyclic layered graph consisting of $l+2$ layers $0, 1, \dots, l+1$. Layer 0 and $l+1$ consist only of the source and the target respectively. The nodes of layer i correspond to all facilities of type p_i. Between layers i and $i+1$, we have a complete bipartite graph,

[1] In fact, the authors address a slightly different version of the problem in a sense that they do not require the trip to end in a target t. Instead, their query consists only of a source s and the sequence $(p_1, p_2, \dots, p_l)$ of facilities to visit. But the two variants are closely related and the authors also mention the variant considered in our work.

where the (directed) edge from node $v^{(i)}$ in layer i to node $w^{(i+1)}$ in layer 1 has cost of and corresponds to the shortest path from v to w in G (in [SKS05] this is simply the Euclidean distance). In Figure 1.2 and 1.3 we see such a layered graph for a query (s, t, p_1, p_2, p_3) (the nodes in layer 1 could correspond to locations of ATMs, layer 2 nodes to gas stations, and layer 3 nodes to grocery stores). Once this layered graph has been constructed, running Dijkstra from s or even simpler, relaxing the edges from top to bottom yields the desired optimal route. In practice however, the construction of such a layered graph is prohibitive, both in terms of running time as well as of space consumption. Remember that we are dealing with thousands of facilities in one single class. So in [SKS05] the authors propose a new algorithm – LORD – which avoids the explicit construction of the complete layered graph by an adaptive thresholding technique. LORD is refined to R-LORD using a range query data structure for nearest neighbor queries to more efficiently prune the search space. The case where where the underlying space is not the Euclidean space but a road network is discussed briefly and no experimental results are reported in the paper – probably because computing the (now shortest path) distances between nodes of consecutive layers is very costly, even though the pruning by (R-)LORD reduces the number of such costly computations.

Further results in the context of euclidean nearest neighbor exploration with spatial databases can be found in [HS99, JKPT03]. In [CKSZ11], the authors consider explicitly the case where the underlying space is a road network and solve sequenced route queries and even generalizations thereof where precedence constraints between the facilities are given (for example, we might have to visit an ATM *before* going to a restaurant if we are short on cash, but it does not matter whether we drop the letter before or after the ATM or restau-

rant visit in a mailbox). They propose several heuristics which do not guarantee optimality, though. The reported experimental results also suggest that their approach is only practicable for rather small road networks – their showcase road network (California) has only about 21k nodes and their reported query times are already above $1/100$th of a second for this small network.
Sequenced route queries (though not under this name) are instrumented for example in [LZZ$^+$09] to solve the so-called *map matching* problem. Here, the goal is to match (possibly imprecise) location measurements by GPS to actual routes in a road network. By computing the shortest path visiting the nodes contained in the respective measurement circles in the order in which the measurements were taken, faithful routes could be determined.

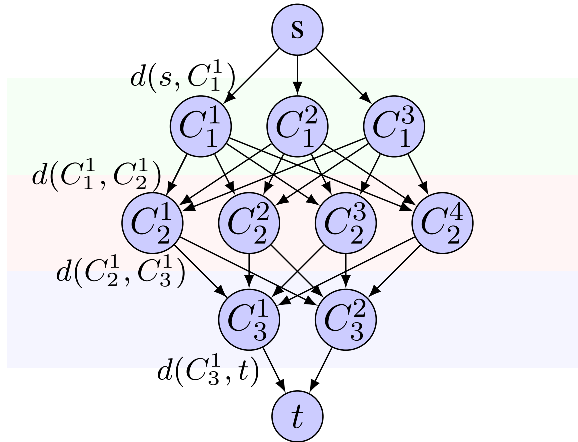


Figure 1.2.: EDJ layer approach: The enhanced Dijkstra based approach builds an explicit layered graph to cover all inter layer distances. Each edge (C_i^l, C_{i+1}^k) represents the shortest path from C_i^l to C_{i+1}^k in the underlying road network and is weighted with $d(C_i^l, C_{i+1}^k)$. A single Dijkstra computation from s recovers the optimal $s - t$ route.

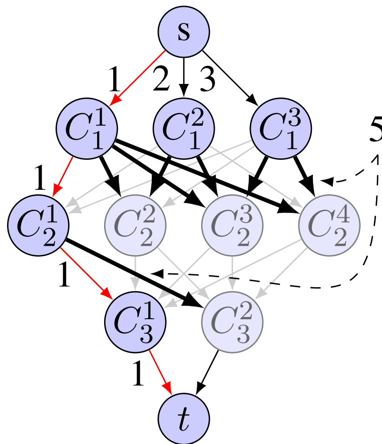


Figure 1.3.: Upper bounds: As soon as one feasible s t route is discovered POIs with larger distance to s can be ignored. In this example no layer need to settle nodes with an s distance larger than 5 or propagate their distances.

1.2. Our Contribution

We propose two speed-up techniques for answering sequenced route queries. The first is based on a general preprocessing technique for ordinary shortest path queries called *contraction hierarchy* [GSSD08] which can be extended to deal with sequenced route queries. The second technique makes use of the fact that likely most sequenced route queries are more of a local kind (doing things on the way back home from work rather than on a cross-country trip), and results in a certain distance sensitivity. Our algorithms – in contrast to [CKSZ11] *always* compute the optimum solution and do so faster by orders of magnitudes being able to deal with network sizes that could not be processed before. Our fast query times for sequenced route queries also allow us to answer queries without fixed order as long as the number of facilities to be visited remains moderate (as seems to be the case in many real-world scenarios).

2. Basic Concepts

2.1. Contraction Hierarchies

As the computation of s-t shortest paths is of essential importance for our work, we gain much by employing time efficient algorithms to this regard. Although Dijkstra's algorithm is very fast from a theoretical standpoint, a single s-t query on a Germany sized road network will take in the order of several seconds with a fairly optimized implementation. This can be attributed to two facts: first of all, for a single s-t query Dijkstra's algorithm will compute all s-v shortest paths for $d(\pi(s, v)) < d(\pi(s, t))$ although we might not be interested in this information at all. And more important, Dijkstra's algorithm is an online computation. Although we can consider all road networks as static for our applications the algorithm does not employ any a priori knowledge to gain advantage of this fact.
So we are more interested in an algorithm which may perform offline precomputation but runs in sublinear time at query time.
Contraction hierarchies provide the means to archive this goal. Introduced by [GSSD08] a contraction hierarchy (CH) is two-part algorithm consisting of a precomputation step to compute the eponymous hierarchy by the elementary operation of node contractions and a modification of Dijkstra's algorithm which exploits this newly gained structure at query time. In the following we will describe both steps in more detail.

The hierarchy of the nodes for a graph $G = (V, E, \gamma)$ is a partial order $\mathcal{H}$ over the the set of nodes V which conveys a concept of importance for each of them. If the node $v \in V$ is above an other node $w \in V$ in our hierarchy we denote this as $v \succ_{\mathcal{H}} w$. The the concrete meaning of “importance” and therefore the hierarchy as objective criteria will only become clear later on with the modified query time algorithm but the underlying idea is that an unimportant node could be removed from the graph without impairing ‘too many’ shortest paths. For now we just assign a unique value to each node and contract them ascending in the order of their importance.
The operation of contraction of a given node $v \in V$ in the case of directed edges is defined as follows: Consider all pairs of distinct incident nodes of v. Let u, w such a pair of nodes with $e_1 = (u, v), e_2 = (v, w) \in E$. Check if $\pi(u, w) = (e_1, e_2)$, that is check if v is visited by the shortest path from u to w. If this is the case add an shortcut (u, w) to E and set its length to the combined length of e_1 and e_2. Otherwise if $\pi(u, w) \neq (e_1, e_2)$ we call this path a witness path and do not add the additional edge. In the case of undirected edges, the same procedure is simplified by the fact that we do not need to differentiate between in- and out edges. Repeat this witness search for all u, w pairs, potentially add some shortcuts and delete v from G afterwards.
After we contracted all nodes in G we end up with an empty graph, so the last step is to take the original graph and augment it with all the shortcuts we found to be necessary during the contraction phase. After this precomputation step we end up with a graph $G_{\mathcal{H}} = (V, E \cup \text{Shortcuts})$.
To answer a shortest path query we now can exploit the fact that each s-t shortest path in $G_{\mathcal{H}}$ breaks down in an upward part and a downward part. Let $(v_1, \ldots, v_k, \ldots, v_m)$ with $v_1 = s$ and $v_m = t$ be the sequence of nodes visited by $\pi_{G_{\mathcal{H}}}(s, t)$ and without loss of

generality v_k shall be the linking node between the upward and downward part. Then it holds that for $i \in (1, k] : v_i \succ_{\mathcal{H}} v_{i-1}$ and for $i \in (k, m] : v_i \prec_{\mathcal{H}} v_{i-1}$.
This is now used by the query time algorithm, which consists of a bidirectional Dijkstra search, starting in s and t as before with the crucial modification that the both of them only consider adjacent nodes with strictly increasing importance according to $\mathcal{H}$. As with the bidirectional version before they will meet up in several nodes of which one will realize the shortest overall distance. This is the exact node we called v_k. Consider Figure 2.2 for a visualization of this process.

The described search space for each possible node $v \in V$ is called the upward-/downward graph induced by the node, depending on v's role as source or target and will be denoted as $G^{\uparrow}_{\mathcal{H}}(v)$ and $G^{\downarrow}_{\mathcal{H}}(v)$. The sizes of these search trees are highly dependent on the choice of $\mathcal{H}$ during the preprocessing step and the main contributor to the query runtime. It was proven in [Mil12] that both the calculation of an optimal $\mathcal{H}$ is APX-hard and the tightest bound on the number the number of necessary shortcuts is $\mathcal{O}(nh \log D)$ with D, h being the diameter and highway dimension of the graph. On the contrary a road network is a very special kind of graph class where we will be able to compute hierarchies with $\mathcal{O}(n)$ edges in general. To this regard there is ongoing research to improve the heuristics from [GSSD08].
On a country sized road network we will be able to gain a more than two orders of magnitude faster runtime compared to a reasonably tuned bidirectional Dijkstra computation.

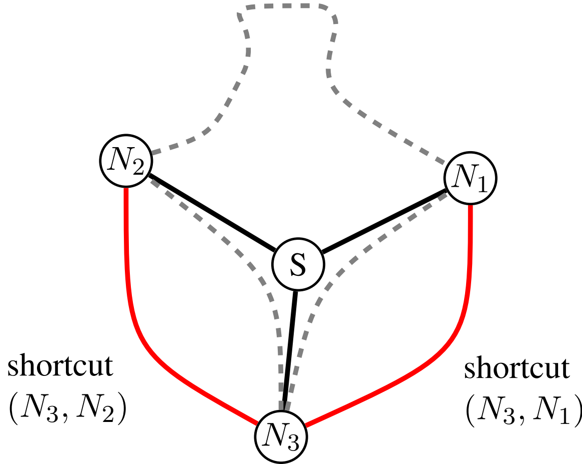


Figure 2.1.: During the contraction of node S we computed the shortest paths $\pi_1(N_1, N_2)$, $\pi_2(N_1, N_3)$ and $\pi_3(N_2, N_3)$ and found the later two to be of the form $\pi_1 = (N_1, S, N_3)$ and $\pi_3 = (N_2, S, N_3)$, while the first one does not contain S. From this knowledge we add shortcuts (N_1, N_3) and (N_2, N_3) with a weight corresponding to the length of π_1 and π_3.

2.1.1. Implementation Details

As mentioned in section 2.1 the main contributing factor of the real world usability and performance of a contraction hierarchy is the number of added shortcuts during the preprocessing phase. The employed heuristics to compute a good contraction order have changed quite significantly compared to the original proposals in [GSSD08]. In the more recent literature the priority function which dictates the contraction order is an on-line heuristic. For example [DGNW11] defines it as $2\,\mathrm{ED}(u) + \mathrm{CN}(u) + \mathrm{H}(u) + 5\,\mathrm{L}(u)$ for a node $u \in V$,

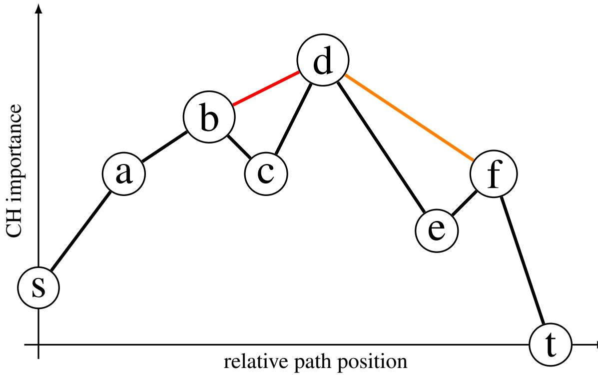


Figure 2.2.: For the computation of $\pi(s,t)$ we explored $G^{\uparrow}_{\mathcal{H}}(s)$ and $G^{\downarrow}_{\mathcal{H}}(t)$ and found d to be a common node of high rank. (s,a,b,d,f,t) will yield the correct shortest path after unpacking because we inserted the shortcuts (b,d) and (d,f) exactly as needed during the contraction phase. Note that e was contracted before its two neighbours d and f, and so was c respectively.

where ED is the 'edge difference' as difference between shortcuts added and edges deleted if u is contracted at this point, CN is the number of previously contracted neighbors, H is the number of edges which would have to be short-cutted and finally L is the 'level' of u which conveys a concept of height of the contraction so far. $\mathrm{L}(u)$ is defined as $\mathrm{L}(v)+1$, where v is the highest-level neighbor of u in the augmented graph at this point in the contraction.
Our heuristic is very similar to the one discussed so far and can be described as round based greedy contraction. Let $G^{i}_{\mathcal{H}}$ be the augmented graph where the first i nodes are already contracted, so $G^{0}_{\mathcal{H}} = G$. Each round operates on $G^{k}_{\mathcal{H}}$ for some k, where the first k nodes where contracted in former rounds and consists of the fol-

lowing steps for round R:

(a) Compute a 'simple metric' $\mathrm{SM^k}(u) = \mathrm{inDeg^k}(u) \cdot \mathrm{outDeg^k}(u)$ for all nodes in $G^k_{\mathcal{H}}$ and sort the nodes ascending by this metric.
(b) Consider the nodes in this order and compute a maximum independent set $\mathrm{IS^k}$.
(c) Compute the average weight of $\mathrm{IS^k}$ as $\mathrm{AVG}(\mathrm{IS^k}) = |\,\mathrm{IS^k}\,|^{-1} \cdot \sum_{u \in \mathrm{IS^k}} \mathrm{SM^k}(u)$ and omit any nodes with $\mathrm{SM^k}(u) > \mathrm{AVG}(\mathrm{IS^k}) \cdot 1.1$.
(d) Compute a 'complex metric' for all remaining nodes, which consists of contracting the node itself and counting the number of necessary shortcuts $\mathrm{SC^k}(u)$. This gives the edge difference for each of these nodes, to be more specific $\mathrm{CM^k}(u) = 6 \cdot \mathrm{SC^k}(u) - 2 \cdot (\mathrm{inDeg^k}(u) + \mathrm{outDeg^k}(u))$.
(e) Again order the nodes in ascending order ascending by $\mathrm{CM^k}(u)$ and throw away all nodes with $\mathrm{CM^k}(u)$ larger than 1.1 times the average weight of all remaining nodes.
(f) Finally contract all the surviving nodes in arbitrary order to get the graph $G^{k+l}_{\mathcal{H}}$ if we were able to contract l nodes. Set $\mathrm{L}(u) = R$

The crux of the matter is to employ the independent set which separates all contraction candidates. This enables us to compute step (d) completely in parallel as none of the possible contractions interfere with each other. As for the notation of the level $\mathrm{L}(v)$ of a node, all the contracted nodes in round R can be set to $\mathrm{L}(v) = R$, so the 'height' of the hierarchy is equal to the number of required rounds. In the first few rounds the algorithm will be able to find large independent sets and accordingly contract many nodes of low degree. As the contraction process progresses this average degree increases as the 'important' nodes remain in a dense relatively small subgraph. In general we will not be able to contract all the nodes as in the last few rounds a small 'core' of important nodes in a clique

like graph remains. To contract this core we would need to add an unreasonable large number of shortcuts, so we stop after contracting 99.5% of G and assign a height of infinity to all remaining nodes. On the downside we will have to modify the query time search to not only consider neighbors with strictly higher, but also equal importance. That is, we will have to search through all the remaining 0.5% of nodes for each shortest path query.
Unfortunately the 'quality' of the resulting hierarchy can not be measured by a single metric. It is desirable to get away with a small amount of rounds, the least possible amount of added shortcuts and a preferably small size of the average upward graph $G^{\uparrow}_{\mathcal{H}}(s)$. All of these optimization goals are in conflict with each other. A smaller number of rounds comes at the cost of more edges, but even with more edges the average upward graph can be smaller.

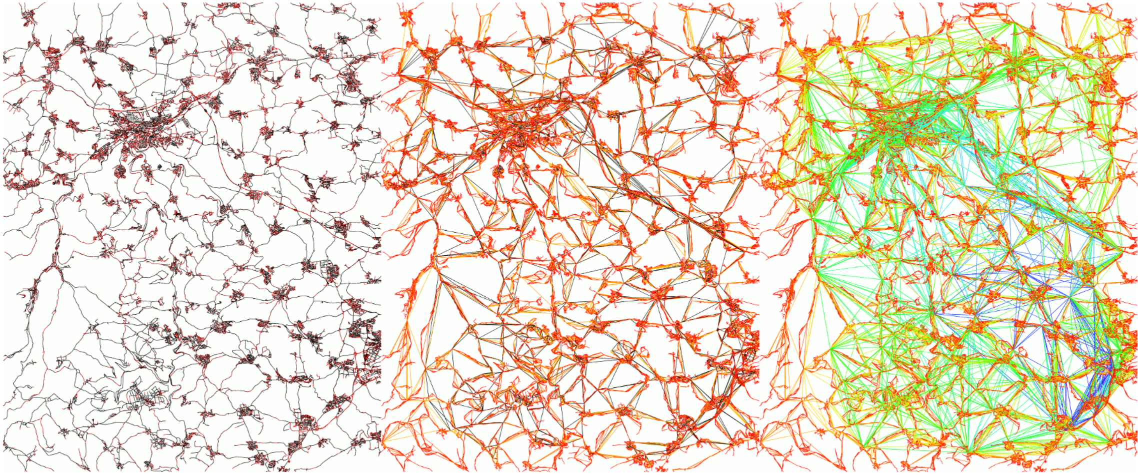


Figure 2.3.: Visualization of the computation process of a contraction hierarchy. On the left side there is a single layer of red shortcuts spanning over the black base graph. In the middle we added about 20 layers of shortcuts, where higher layers are orange. On the right is the completed CH with higher layers being yellow, green and finally blue. As expected, the upper layers of the CH are a dense cluster of a relatively small number of important nodes.

3. Speeding-Up Sequenced Route Queries

In this section we develop our two main tools for speeding-up sequenced route queries. While both techniques can be employed independently, the combination of both yields the best speed-up compared to the naive EDJ approach. The first speed-up technique – *iterative* doubling – works well, if the actual result path is relatively short – probably the most frequent type of query result in practice –, avoiding the exploration of facilities that are far away from source and target. The second technique – contraction hierarchies (CH) – has been developed in the context of fast point-to-point shortest path queries [GSSD08]. We extend CH in a natural way to speed-up the computation of inter-layer distances. This technique applies equally well for local and non-local queries. Both speed-up techniques do not compromise optimality of the result.

3.1. Iterative Doubling

Let us first modify the EDJ algorithm such that no explicit construction of the layered graph is necessary:

a) Run Dijkstra from s to compute distances d_0 to all nodes C_1
b) Run a *single* Dijkstra starting at all nodes in C_1 where each node $v \in C_1$ has initial distance value $d_0(v)$ until all nodes in C_2 are settled. This computes shortest path distances d_1 from s via at

least one node in C_1.

c) Run a *single* Dijkstra starting at all nodes in C_2 where each node $v \in C_2$ has initial distance value $d_1(v)$ until all nodes in C_3 are settled. This computes shortest path distances d_2 from s via at least one node in C_1 and one node in C_2.

d) ...

e) Run a *single* Dijkstra starting at all nodes in C_l where each node $v \in C_l$ has initial distance value $d_{l-1}(v)$ until the target t is settled. This actually computes the shortest path from s via at least one node in C_1, at least one node in C_2, ..., at least one node in C_l to t.

Note that the $(i+1)$'st Dijkstra computation ($i = 1, \ldots, l$) starts with a preinitialized priority queue containing all nodes of C_i with initial distance as determined by the i-th Dijkstra computation. Clearly, the running time of this approach is essentially that of performing l Dijkstra runs on the graph — which is already a considerable improvement to EDJ which essentially required $\sum |C_i|$ many Dijkstra computations to compute the weights of all inter-layer edges.

There is still an obvious source of inefficiency here. We expect realistic sequenced route queries to be mostly local (typical commuter distances are 40km to 60km at most which translates to 60 to 90 minutes). It seems very inefficient to explore facilities that are hundreds of kilometers (and hours of driving) away — but that is exactly what the above approach does. What can we do about it?

Let us assume for now that we know the length (duration) of the optimal path from s to t visiting facilities in the given order; let that length be D. We could stop each (!) Dijkstra computation above once we reach distance D and still guarantee that we find the optimal path since no subpath of the optimal path can have length more

than D. Note that in case the optimal path is rather short – let's say it takes 70 to 100 minutes – this will drastically reduce the search space of every single Dijkstra.

Unfortunately we do not know the optimal route's exact length D a priori, this is where the iterative doubling part comes into play. We start with some estimation/lower bound D' for D which can be pretty small (let's say 10 minutes). We use the above sequence of computations except for one important difference: we abort each Dijkstra run once we have settled all nodes at distance at most D'. Two things can happen: a) the computation does not reach t – so our estimation D' was too small, we double D' and repeat. b) the computation does reach t – so we have a valid path from s to t visiting facilities on the way in the right order on a path of distance D''. It is not hard to see that this solution is optimal.

3.2. CH enhanced Iterative Layer Search

As described in detail in section 2.1, we can make use of the CH to answer a simple $s - t$ shortest path query by performing two interleaved Dijkstra computations, one starting in s, the other starting in t. The former one only considers edges in $G_s^{\uparrow}$, the latter only edges in $G_t^{\downarrow}$. When both Dijkstra computations settle a node $v \in (G_s^{\uparrow} \cap G_t^{\downarrow}), d(s,v) + d(v,t)$ is an upper bound for $d(s,t)$ and the shortest path is realized by $\mathrm{argmin}_{v \in (G_s^{\uparrow} \cap G_t^{\downarrow})}(d(s,v) + d(v,t))$. This method can be extended to one to many shortest path computations where the task is to find all shortest paths from a node $s \in V$ to a set of nodes $T \subset V$. The conceptually easiest method is to mark all edges in the downward graph for each $t \in T$ and use and Dijkstra computation from s which considers all edges in $G_s^{\uparrow}$ and all marked edges.

For our concrete problem of speeding-up an inter-layer Dijkstra, we can extend the same methodology even further by the following preprocessing step. For each facility/POI class we construct the *downward graph for this facility class* by taking the union of the downward graphs of all nodes in that facility class. These downward graphs can be represented by a one bit marker for each edge and facility/POI class indicating whether the edge belongs to the respective downward graph. Then, during query processing, the ordinary inter-layer Dijkstra is replaced by a Dijkstra operating on the union of the upward graphs of the settled nodes of the current facility class and the downward graph for the next facility class. This speed-up technique does rely on locality of the queries but exhibits a considerable speed-up in all cases.

3.3. Arbitrary Order Routes

Our speedup techniques put more advanced types of route queries within reach. Let us reconsider our original notation of sequenced queries with a start and endpoint $(s, t) \in V^2$ and its POI type vector $c = (p_1, p_2, \ldots, p_n) \in \mathcal{C}^n$. As also discussed in [CKSZ11], a very natural extension is to soften the total order requirement of the different POI types in c. Maybe we insist on visiting a POI of type c_1 before one of type c_2 but do not really care about the order of the other elements in c. Under the reasonable assumption that $|c|$ is typically quite small – let's say $|c| < 5$ – the speedup obtained by our new techniques allows us to to check all $(|c|)!$ possible orderings. For partial orderings it is a mere task of enumerating the subset of all compatible permutations.
The computational costs of this approach will typically be less than

the combined cost of testing all possible permutations of c independently because each length $d(s,t)_{\pi(c)}$ of a shortest route for a given permutation $\pi(c)$ of c is an upper bound for the minimum of all these routes and can therefore be carried over to the subsequent route computations. Another potential for a drastic reduction of computational cost is that we are free to choose the order in which we test these permutations. If we enumerate them in ascending lexicographic order we preserve the longest possible prefix between consecutive permutations. This enables us to reuse all layers which are managing the distances for the sets C_i in this prefix, in an one to one fashion. Employing this strategy we do only need to generate layers for the actually changed suffix of c and propagate our already known distances into the "joint"-layer.

4. Experimental Results

Experiments were performed on the road network of Germany with $15.015.877$ nodes and $30.760.517$ edges extracted from OpenStreetMap [OSM], it has a diameter of about $1100\,\mathrm{km}$. The CH preprocessing step resulted in an augmented graph with $61.630.345$ edges and a preprocessing time of about 15 minutes on a single core of an Intel i5-2500k. As performance metrics we use time as well as priority queue pops (which is equivalent to the number of settled nodes) as a more robust and platform independent indicator.
All timings are averaged over 250 random queries and performed on a single core of the same i5-2500K.

We measured four algorithms: the *naive* algorithm which is essentially the first version of the iterative doubling approach (without the doubling) as explained in Section 3.1, the *iterative doubling* approach on its own, the *CH-naive* approach (without iterative doubling), and the combined version *CH-iterative* which employs both the iterative doubling as well as the CH. We refrained from benchmarking the naive EDJ approach, since there is no hope to obtain any acceptable running times as was already mentioned in [SKS05].

type	count	type	count
parking	65282	pub	13611
restaurant	53073	post office	4778
post box	40857	atm	4201
supermarket	21411	taxi	2571
bakery	18527	university	821
bank	18425	car sharing	605
pharmacy	16156	airport	79
gas station	15902	gambling hall	70
vending machine	14177		

Table 4.1.: Different POI types and their frequencies out of the OSM Dataset for Germany.

4.1. Query Locality

We first consider a fixed query type with $c =$ (atm, gas station, post box, supermarket) and examine the influence of locality on these queries, where we call a query *local* if the $s - t$ distance in the underlying road network is between 40 and $50\,\mathrm{km}$. A *nonlocal* query shall be between 400 and $500\,\mathrm{km}$ long. As local queries typically result in short resulting routes and low upper bounds we expect to see significant speedups of the iterative doubling approach.

In Table 4.2 we measured 250 random $s - t$ queries for all four approaches to examine the influence of locality. For the short queries the iterative aspect results in a speedup of factor 30 while the CH gains a speedup factor of 340. The iterative CH approach needs 43ms on average for these queries, compared to 23449ms for the EDJ version. The non-local queries are significantly harder to solve, the iterative gain over the full searches shrinks down to a factor of 1.22 an even the CH version only results in a speedup of about 17 and average query times of 1259ms for the iterative CH approach.

query type	naive	iterative
local popcnt	$61.49 \cdot 10^6$ ($14.30 \cdot 10^5$)	$19.60 \cdot 10^5$ ($11.15 \cdot 10^5$)
non-local popcnt	$71.36 \cdot 10^6$ ($23.44 \cdot 10^5$)	$57.67 \cdot 10^6$ ($10.69 \cdot 10^6$)
local time	23449 (272)	650 (398)
non-local time	26855 (1043)	21365 (4206)
	CH-naive	CH-iterative
local popcnt	$17.82 \cdot 10^5$ (6605)	59189 (38725)
non-local popcnt	$20.12 \cdot 10^5$ (58069)	$16.37 \cdot 10^5$ ($31.27 \cdot 10^4$)
local time	1404 (14)	43 (28)
non-local time	1565 (52)	1259 (249)

Table 4.2.: PQ pop counts and timings in ms for local ($40-50\,\mathrm{km}$) and non-local ($400-500\,\mathrm{km}$) route queries for $c =$ (atm, gas station, post box, supermarket), averaged over 250 random queries. The values in brackets are the standard deviation.

The timings for the naive non-CH version matches, as expected, the runtime of 5 independent one to all Dijkstra computations in this road network.

4.2. Influence of the Order of POIs

Another interesting question is whether the order of the POIs in c is important for the performance of our algorithm in particular when choosing POI types with significant size differences.
We choose c as (atm, post box, bank) with set sizes of 40857, 4201 and 18425 (see Table 4.1) and compare all possible 6 permutations for the CH naive and iterative approach.
In Table 4.3 we measured the resulting runtime for all these permutations for local queries. As expected the runtime depends on the

c vector	CH naive	CH iterative
atm, post box, bank	1031 (16)	33 (22)
atm, bank, post box	898 (17)	31 (21)
post box, atm, bank	1097 (19)	34 (23)
post box, bank, atm	1152 (23)	34 (22)
bank, atm, post box	957 (18)	32 (21)
bank, post box, atm	1143 (20)	35 (23)

Table 4.3.: Influence of different sized POI sets and their permutation in the c vector. All values are runtime in ms, the values in brackets are the standard deviation.

order in c but surprisingly the ratio between the slowest and fastest route type is only about 1.28 where the slowest route is the one with ascending POI set sizes while the fastest is the one with descending sizes accordingly.

4.3. Influence of the Number of POI types

In Table 4.4 we measured the increase in runtime for growing c vectors. Based on the results in section 4.2 we chose the worst case for the different sizes of c i.e. starting with the largest set of "parking" for $|c| = 1$ followed by "parking" and "restaurant" in this order, and so on. Apparently adding more POI types to the end of c adds a constant amount of additional computational cost which somewhat scales with the size of the added POI type. Note that adding restaurants for $|c| = 2$ results in a 18ms increase to query timings whereas the banks set for $|c| = 6$ has roughly one third the size and results in a 10ms increase.

c vector size	CH naive	CH iterative
1	491 (9)	19 (13)
2	1177 (28)	37 (23)
3	1769 (33)	52 (33)
4	2279 (24)	65 (42)
5	2636 (40)	75 (48)
6	2981 (52)	85 (55)

Table 4.4.: Influence of different sized POI sets and their permutation in the c vector. All values are runtime in ms, the values in brackets are the standard deviation.

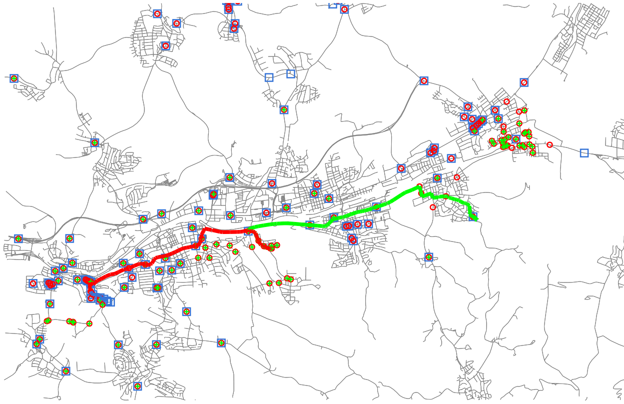


Figure 4.1.: The search space of a sequenced route query with 2 different POIs sets, resulting in 3 path segments. The background nodes are colored by the responding layers. This is the incremental CH version.

4.3.1. Arbitrary Order Routes

As described in section 3.3 we also used our CH approach in a setting of permuted layer orderings with two different query types.

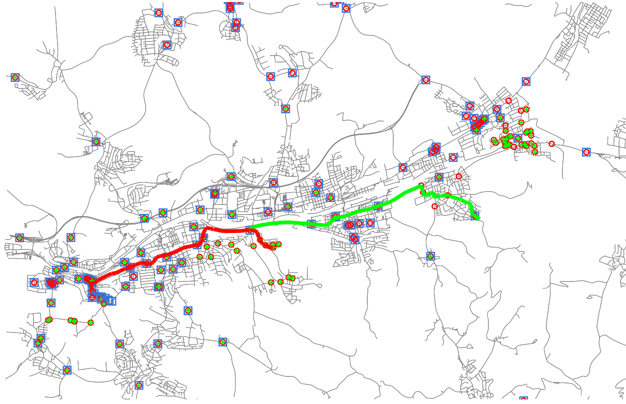


Figure 4.2.: The search space of a sequenced route query with 2 different POIs sets, resulting in 3 path segments. The background nodes are colored by the responding layers. This is the non-incremental CH version. Note that some nodes are visited by more layers than before.

First we investigated the impact of the type vector length c on the number of required Dijkstra queue pops with abundant POI types, namely with $c_3 =$ (“parking“, “restaurant“, “gas station“), $c_4 = c_3+$“bakery” and $c_5 = c_4+$“supermarket“. Then we swapped the last to POIs of c_5 with the two rarest one ”airport“ and ”gambling hall“ yielding c'_4 and c'_5 with the intend to generate very long shortest paths due to the sparseness of those two types. In both cases we generated 500 random local $s-t$ pairs with $d(s,t) < 50\,\mathrm{km}$. While all solutions for the $c_{\{3,4,5\}}$ cases are well below $55\,\mathrm{km}$ the routes for $c'_{\{4,5\}}$ partially passed $200\,\mathrm{km}$ in length. The pop counts can be compared to the local pop counts in Table 4.2.

From table 4.5 we can gain two insights, first that holding the layers in the common prefix does not result in a reduced amount of re-

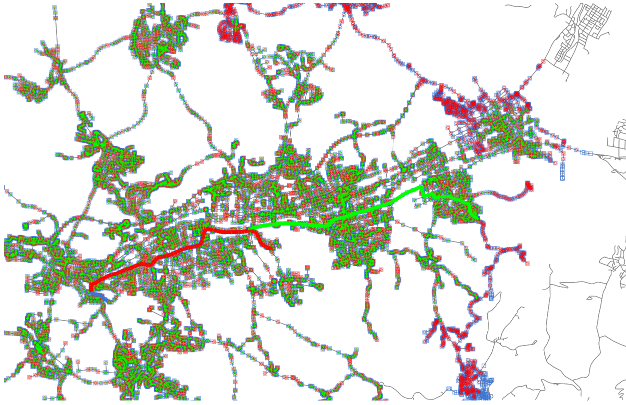


Figure 4.3.: The search space of a sequenced route query with 2 different POIs sets, resulting in 3 path segments. The background nodes are colored by the responding layers. This is the incremental non-CH version where each layer has to explore all nodes with a distance lower than the one of the optimal path.

POI	pop count CH	pop count non-CH
c_3	$14.68 \cdot 10^4$ $(14.79 \cdot 10^4)$	$34.50 \cdot 10^5$ $(33.35 \cdot 10^5)$
c_4	$52.62 \cdot 10^4$ $(54.29 \cdot 10^4)$	$12.88 \cdot 10^6$ $(12.51 \cdot 10^6)$
c_5	$24.61 \cdot 10^5$ $(26.10 \cdot 10^5)$	$61.65 \cdot 10^6$ $(60.92 \cdot 10^6)$
c'_4	$20.97 \cdot 10^5$ $(20.92 \cdot 10^5)$	$57.99 \cdot 10^6$ $(57.25 \cdot 10^6)$
c'_5	$16.39 \cdot 10^6$ $(12.28 \cdot 10^6)$	$53.41 \cdot 10^7$ $(42.34 \cdot 10^7)$

Table 4.5.: Pop counts for arbitrary order routes. The different POI vectors are as described in section 4.3.1. Averaged over 500 **random** $s - t$ **queries with** $d(s,t) < 50Nitkm$ **(only 100 queries for** c'_5 **non-CH).**

quired work. This is due to the fact that for most of the queries the

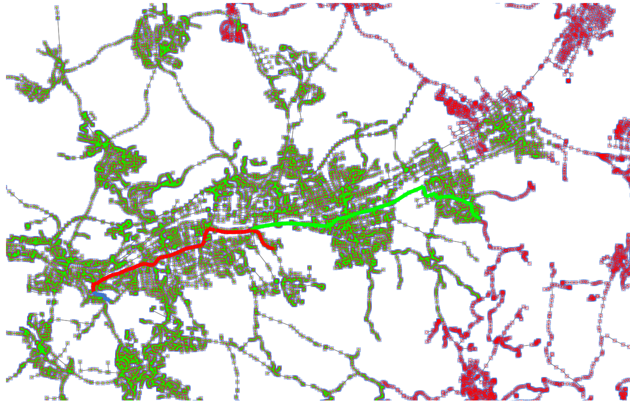


Figure 4.4.: The search space of a sequenced route query with 2 different POIs sets, resulting in 3 path segments. The background nodes are colored by the responding layers. This is the non-incremental non-CH version, here the graph is so small compared to the query length that all layers explore the whole graph.

algorithm finds an nearly optimal upper path length bound during the first two or three possible permutations. All following permutations need to explore this radius in the "non-prefix" layers but fail to reach the target by a few kilometres. And more important that the runtime is very sensitive regarding the sparseness of the used POI types. Both c_4' and c_5' queries are expensive due to their final length. The high standard derivation of all values are the result of some very expensive queries which only gain optimality in the last few of the tested permutations.
Nevertheless the Dijkstra implementation on our test machine is able to process about 2.9 million pops per second, so all "typical" $c_{\{3,4,5\}}$ queries can be answered well beyond one second. Also for nearly all the queries we observed, the gap between a random cho-

sen permutation of c and the optimal one is typically < 1.5, this becomes particularly evident for the c' vectors where the optimal route needs to visit the nearest airport and then just ”collects“ the more common POIs on the way with some small detours.

Consider figures 4.1, 4.2, 4.3 and 4.4 at the end of this chapter as comparison of the different search spaces for a the same query of two POIs (so $|c| = 2$) of the four proposed techniques. Figure 4.1 and 4.2 offer a vastly smaller amounts of visited nodes due to the usage of CH. Figure 4.3 and 4.4 show the effects if the incremental search, although it only cuts off a small part of the graph due to the relatively long query compared to the graphs diameter in this case.

5. Conclusion and Future Work

In this chapter we have considered the problem of answering sequenced route queries and developed two very efficient speed-up techniques that allow for the *exact* computation in few milliseconds for realistic queries involving common tasks/points of interest. The focus of this work has been the case where the order in which the points of interests are to be visited is fixed. The very fast query times for fixed order queries allows for a straightforward treatment of unordered or only partially ordered queries by simply enumerating all possible orderings. Another interesting topic for future research is the transition from fixed edge costs to parameterized ones. This extension seems natural under the assumption that the ”cost“ of an edge could be the required travel time or the battery consumption necessary to cross this road segment.

Part V.

Epilogue

Conclusion

In this thesis we have considered three challenges in the context of shortest path computation and their applications for routing techniques in large scale road networks.

In the first chapter we proposed a theoretical framework, based on integer linear programming, to proove lower bounds on the optimal size of transit node sets, which are computed in a certain way. We used an intuitive definition for the property of being a ”long“ shortest path and derived an efficient algorithm to generate all prefixes of such paths. We then showed how to construct a set cover and a set packing instance from this data – which are solved by a very fast greedy approximation algorithm. The solutions of these two instances gave us an approximation bound for the road network instance and notation of ”long“ in question.
With this framework we devised several experiments to compare our results to other construction techniques. While the resulting transit node sets where of similar size on all tested graphs, our sets resulted in a much smaller number of relevant access nodes per node. This corresponds to fewer necessary lookups per shortest path distance query and ultimately in faster lookups.

Path prediction is an application of several graph routing speed up techniques and the main topic of the second chapter. We exploited the remarkable hierarchical property of real world road networks to-

gether with their static nature to derive a highly efficient path prediction protocol. With the concept of reach – an edge property which measures the length of the longest shortest path the edge is part of – and some other ideas, as counting these shortest paths, we devised several offline prediction strategies. In comparison to their online counterparts they can be completely computed in advance and are very cheap to use at query-time. In addition they result in a much higher prediction quality as they are able to take advantage from knowledge about more complex structural graph properties.
We compared these offline strategies with several other online flavors and devised some mixed strategies which improve path prediction by a tremendous amount under all quality metrics we measured.

The final chapter is about a second application of yet an other speed up technique. We used a precomputed contraction hierarchy and proposed a carefully constructed layer based iterative search algorithm to answer sequenced route queries. The algorithm cuts down the required search space by such a large margin, that we were able to answer sequenced route queries with up to 6 segments in the order of several tens of milliseconds. At the same time it retains optimality in terms of resulting path length and an approximation guarantee in the number of explored point of interest nodes in the graph. We used meta data from the OpenStreetMap project to test our approach on large road networks.

Zusammenfassung

In dieser Arbeit bearbeiten wir drei Problemstellungen aus dem Bereich der Routenplanung und deren Anwendung auf großen Straßennetzwerken.

Im ersten Kapitel haben wie eine Methode vorgestellt Transitknotenmengen auf eine bestimmte Art und Weise zu berechnen, die es uns ermöglicht instanzbasiert Aussagen über die Approximationsgüte dieser Mengen zu treffen. Wir haben dafür eine Definition von "langen" kürzesten Wegen gewählt, welche entweder direkt auf deren Länge oder aber dem Dijkstrarank ihrer Knoten basiert, und einen Algorithmus entwickelt, der Präfixe aller dieser Pfade effizient berechnet. Mit diesen Präfixen haben wir ein Mengenüberdeckungsproblem und das duale Mengenpackungsproblem definiert, mit denen wir aus den Resultaten der ganzzahligen linearen Optimierung eine untere Schranke für die optimale Größe einer optimalen Transitknotenmenge herleiten können.
Wir haben diesen Ansatz mit anderen Konstruktionsverfahren verglichen, um zu erfahren ob diese bei gleicher Definition von "lang" nicht wesentlich kleiner gewählt werden könnten. Dies ist nicht der Fall. Allerdings konnten wir zeigen, dass die mit unserer Methode generierten Transitknotenmengen in wesentlich weniger Transitknoten per Knoten im Graph resultieren und damit direkt schnellere kürzeste Pfad Anfragen erlauben.

Im zweiten Kapitel ging es um Pfadvorhersage als Anwendung von

schnellen Routenplanungsalgorithmen. Da Straßengraphen eine stark ausgeprägte hierarchische Struktur aufweisen, konnten wir dies nutzen um gefahrene kürzeste Wege ab einer gewissen Länge mit sehr hoher Präzision vorherzusagen. Wir haben dafür die ”Reach“ Metrik von Kanten benutzt, die für jede Kante die Länge des längsten küzesten Pfades angibt, welcher durch diese Kante geht. Zusammen mit anderen komplexen Metriken wie der Anzahl an kürzesten Wegen welche in einem Knoten starten, haben wir mehrere komplett auf Vorberechnung basierende Vorhersagestrategien entwickelt. Diese haben gegenüber ihren online Gegenstücken nicht nur den Vorteil bei der eigentlichen Anfrage weniger Rechenschritte zu benötigen, als auch wesentlich weniger Vorhersagefehler zu erzeugen.
Wir haben mehrere online sowie offline Strategien praktisch verglichen und eine optimale Kombination vorgestellt, welche die Anzahl von Vorhersagefehlern minimiert.

Das letzte Kapitel ist eine Anwendung von ”Contraction Hierarchies“. CHs ermöglichen sehr schnell kürzeste Wege zu berechnen. Dies nutzen wir um eine Modifizierte Variante des Problems des Handlungsreisenden zu lösen, bei der zwar die Reihenfolge der Städte fixiert ist aber für jede Stadt sehr viele Alternativen zur Verfügung stehen. Wir haben dafür einen Suchalgorithmus entwickelt, der für je zwei aufeinander folgende Zielmengen einen Suchraum bildet und sich dann iterativ in größer werdenden Abständen vom Startpunkt durch diese Suchräume arbeitet. Dies erlaubt uns Optimalität bei der Länge des Weges beizubehalten und nie mehr als einen konstanten Faktor an möglichen Stadtknoten zu viel zu explorieren.
Wir zeigen, dass unsere Implementierung schnell genug ist, um jede realistische Anfrage in wenigen Millisekunden optimal zu beantworten.

Bibliography

[ADF$^+$11] Ittai Abraham, Daniel Delling, Amos Fiat, Andrew V. Goldberg, and Renato Fonseca F. Werneck. Vc-dimension and shortest path algorithms. In *ICALP (1)*, pages 690–699, 2011.

[AFGW10] Ittai Abraham, Amos Fiat, Andrew V. Goldberg, and Renato Fonseca F. Werneck. Highway dimension, shortest paths, and provably efficient algorithms. In *SODA*, pages 782–793, 2010.

[ALS13] Julian Arz, Dennis Luxen, and Peter Sanders. Transit node routing reconsidered. In *SEA*, pages 55–66, 2013.

[BFM06] Holger Bast, Stefan Funke, and Domagoj Matijevic. Transit ultrafast shortest-path queries with linear-time preprocessing. In *In 9th DIMACS Implementation Challenge [1]*, 2006.

[BFM09] Holger Bast, Stefan Funke, and Domagoj Matijevic. *Ultrafast shortest-path queries via transit nodes*, volume 74 of *DIMACS Series on Disrecte Mathematics and Theoretical Computer Science*, pages 175–192. AMS, Providence, RI, 2009.

[CKSZ11] Haiquan Chen, Wei-Shinn Ku, Min-Te Sun, and Roger Zimmermann. The partial sequenced route query with

traveling rules in road networks. *GeoInformatica*, 15(3):541–569, 2011.

[DGNW11] Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. Phast: Hardware-accelerated shortest path trees. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 921–931, Washington, DC, USA, 2011. IEEE Computer Society.

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

[EF12a] Jochen Eisner and Stefan Funke. Sequenced route queries: getting things done on the way back home. In *Proceedings of the 20th International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '12, pages 502–505, New York, NY, USA, 2012. ACM.

[EF12b] Jochen Eisner and Stefan Funke. Transit nodes - lower bounds and refined construction. In *14th Meeting on Algorithm Engineering and Experiments (ALENEX)*, pages 141–149, 2012.

[EFH+11] Jochen Eisner, Stefan Funke, Andre Herbst, Andreas Spillner, and Sabine Storandt. Algorithms for matching and predicting trajectories. In *13th Meeting on Algorithm Engineering and Experiments (ALENEX)*, pages 84–95, 2011.

[GKW06] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. Reach for A*: Efficient point-to-point shortest path algorithms. In *IN WORKSHOP ON ALGORITHM ENGINEERING & EXPERIMENTS*, pages 129–143, 2006.

[GSSD08] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: faster and simpler hierarchical routing in road networks. In *Proceedings of the 7th international conference on Experimental algorithms*, WEA'08, pages 319–333, Berlin, Heidelberg, 2008. Springer-Verlag.

[Gut04] Ronald J. Gutman. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In *ALENEX/ANALC*, pages 100–111, 2004.

[HS99] Gísli R. Hjaltason and Hanan Samet. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2):265–318, June 1999.

[JKPT03] Christian S. Jensen, Jan Kolářvr, Torben Bach Pedersen, and Igor Timko. Nearest neighbor queries in road networks. In *Proceedings of the 11th ACM international symposium on Advances in geographic information systems*, GIS '03, pages 1–8, New York, NY, USA, 2003. ACM.

[KMS06] Ekkehard Khler, Rolf H. Mhring, and Heiko Schilling. Fast point-to-point shortest path computations with arc-flags. In *IN: 9TH DIMACS IMPLEMENTATION CHALLENGE [29]*, 2006.

[LNR02] Alexander Leonhardi, Christian Nicu, and Kurt Rothermel. A map-based dead-reckoning protocol for updating location information. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, pages 15–, Washington, DC, USA, 2002. IEEE Computer Society.

[LZZ+09] Y. Lou, C. Zhang, Y. Zheng, X. Xie, W. Wang, and Y. Huang. Map-matching for low-sampling-rate GPS trajectories. In *Proc. ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (ACM SIGSPATIAL GIS)*, pages 544–545. ACM, 2009.

[Mil12] Nikola Milosavljevi. On optimal preprocessing for contraction hierarchies. In *The 5th ACM SIGSPATIAL International Workshop on Computational Transportation Science*, IWCTS'12, 2012.

[MS08] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, Berlin, 2008.

[OSM] OpenStreetMap. `http://www.openstreetmap.com`.

[San99] Peter Sanders. Fast priority queues for cached memory. *ACM Journal of Experimental Algorithmics*, 5:312–327, 1999.

[SKS05] Mehdi Sharifzadeh, Mohammad Kolahdouzan, and Cyrus Shahabi. The optimal sequenced route query. Technical report, VLDB Journal, 2005.

[SS05] Peter Sanders and Dominik Schultes. Highway hierarchies hasten exact shortest path queries. In *13th European Symposium on Algorithms (ESA'05)*, pages 568–579, 2005.

[SS09] Peter Sanders and Dominik Schultes. *Robust, Almost Constant Time Shortest-Path Queries via Transit Nodes*, volume 74 of *DIMACS Series on Discrecte Mathematics and Theoretical Computer Science*, pages 193–218. AMS, Providence, RI, 2009.

[Vaz01] V.V. Vazirani. *Approximation algorithms*. Springer, 2001.