Fast Exact Shortest Path and Distance Queries on Road Networks with Parametrized Costs*

Julian Dibbelt
Karlsruhe Institute of Technology (KIT)
dibbelt@kit.edu

Ben Strasser
Karlsruhe Institute of Technology (KIT)
strasser@kit.edu

Dorothea Wagner
Karlsruhe Institute of Technology (KIT)
dorothea.wagner@kit.edu

ABSTRACT

We study a scenario for route planning in road networks, where the objective to be optimized may change between every shortest path query. Since this invalidates many of the known speedup techniques for road networks that are based on preprocessing of shortest path structures, we investigate optimizations exploiting topological structures. We experimentally evaluate our technique on a large set of real-world road networks of various data sources. With lightweight preprocessing our technique answers long-distance queries across continental networks significantly faster than previous approaches towards the same problem formulation.

Categories and Subject Descriptors

G.2.2 [**Discrete Mathematics**]: Graph Theory—*graph algorithms, network problems*; G.2.3 [**Discrete Mathematics**]: Applications

General Terms

Algorithms, Experimentation, Measurement, Performance

Keywords

route planning, road networks, shortest paths

1. INTRODUCTION

Road networks of large geographic regions easily consist of hundreds of millions of nodes, and collaborative spatial data collection efforts, such as OpenStreetMap (OSM), have seen growths by two orders of magnitude over the last years. On such large networks, Dijkstra's classical shortest path algorithm [9] incurs substantial running times of several seconds even on modern computer hardware. Since this is too slow for many applications, the past decade has seen numerous research (by both theoretical and applied communities) into techniques that accelerate shortest path queries by applying offline preprocessing [1, 16]. If the graph metric is fixed or changed rarely, these techniques offer very fast queries.

If instead costs change for every query, these techniques cease to provide benefit over Dijkstra's algorithm, due to their relatively expensive preprocessing. Yet, in practice, even *the same user* might prefer a quickest route in the morning but a safe and fuel-efficient route back home. This scenario is considered in Personalized Route Planning (PRP) [10]. Here, every arc in the road graph is associated with a vector c of several non-negative numeric costs, e. g., travel time, distance, speed, emissions, and energy consumption. The input of a query, besides source and target node, consists of a cost vector w with non-negative entries. In the search, every arc is associated with the scalar product of w and c. The output consists of the shortest path with respect to this weighted sum of costs. Moreover, the PRP problem can be extended [8] to vehicle restrictions, such as height and weight limits, or user preferences such as avoidance of highways.

In this work, we improve on the performance of [10] by following a preprocessing approach that naturally exploits the structure of road networks. See the full paper [8] for a more complete exposition.

2. PRELIMINARIES

We denote by $G = (V, A)$ a *directed graph* with *node set* V and *arc set* $A \subseteq V \times V$. An *undirected graph* is denoted by $G = (V, E)$ where E is the *edge set*. For road networks, a node corresponds to a position on the earth's surface and an arc to a road segment between two positions. In particular, *not* every node models a road intersection. For most arcs (u, v) there is a *back-arc* (v, u). However, there are notable exceptions such as one-way streets or highways, which are modeled as two separate one-way streets. We consider multi-cost graphs, where each arc is associated with several costs, such as travel time or distance. Denote by k the number of costs. Formally, we have a function $c : A \to \mathbb{R}^k_{\geq 0}$. An *st-path* between a source node s and a target node t, is a sequence $su \dots vt$ of pairwise adjacent nodes. A graph is called *biconnected* if, after removing any node $v \in V$, the remaining graph $G \setminus \{v\}$ is still connected. A *biconnected component* (BCC) is a subgraph of G that is biconnected. An independent set I is a subset of V such that no two nodes $u, v \in I$ are adjacent, i. e., no edge $\{u, v\} \in E$ exists.

3. TOPOCORE

We propose a core-based speedup technique. In the preprocessing phase, a *core graph* $G_C = (V_C, A_C)$ is computed.

*Partially supported by DFG grant WA654/16-2 and Google Focused Research Award

SIGSPATIAL'15, November 03 - 06, 2015, Bellevue, WA, USA

http://dx.doi.org/10.1145/2820783.2820856

Conceptually, this core graph is a coarsened subgraph containing all major roads, to which the query is restricted to after initial local searches. This decreases query times because G_C is smaller than G.

Formally, the nodes V_C of G_C are a subset of V and called *core nodes*. The arcs of the core are defined as following: For every loop-free path $v_1 v_2 \dots v_k$ for which only the endpoints v_1 and v_k are in V_C and all intermediate nodes are in $V \backslash V_C$, there exists a *shortcut* arc $(v_1, v_k) \in A_C$ in the core graph. This shortcut maintains shortest path distance in the core and thus ensures correctness of the core-based query [15]. Note that it is possible that multi-arcs are created by this construction. The cost vector $c(v_1, v_k)$ of the shortcut is defined as the combination of the cost vectors of the arcs within the path, i.e., $c(v_1, v_k) = c(v_1, v_2) \circ \dots \circ c(v_{k-1}, v_k)$.

3.1 Computing the Core Nodes

Initially, all nodes are core nodes. Then, for each node removed from the core, we potentially have to add shortcuts between *all* pairs of neighbors, in order to maintain shortest path distances for the yet unknown objective function (to be specified in the query). Note that, unlike [6, 7], we must create multi-arcs if an original arc between two neighbors is already present (since we cannot tie-break for an unknown objective function). As the performance of Dijkstra's algorithm depends on both the number of nodes *and* arcs, we would eventually experience diminishing returns if adding too many new arcs while removing nodes from the core.

Hence, our goal is to select as few core nodes as possible while restricting growth in the number of core arcs. In the following, we describe three steps performed in succession to remove nodes from the core, reducing its size and thus accelerating shortest path queries.

Step 1: Removing Dead-Ends.

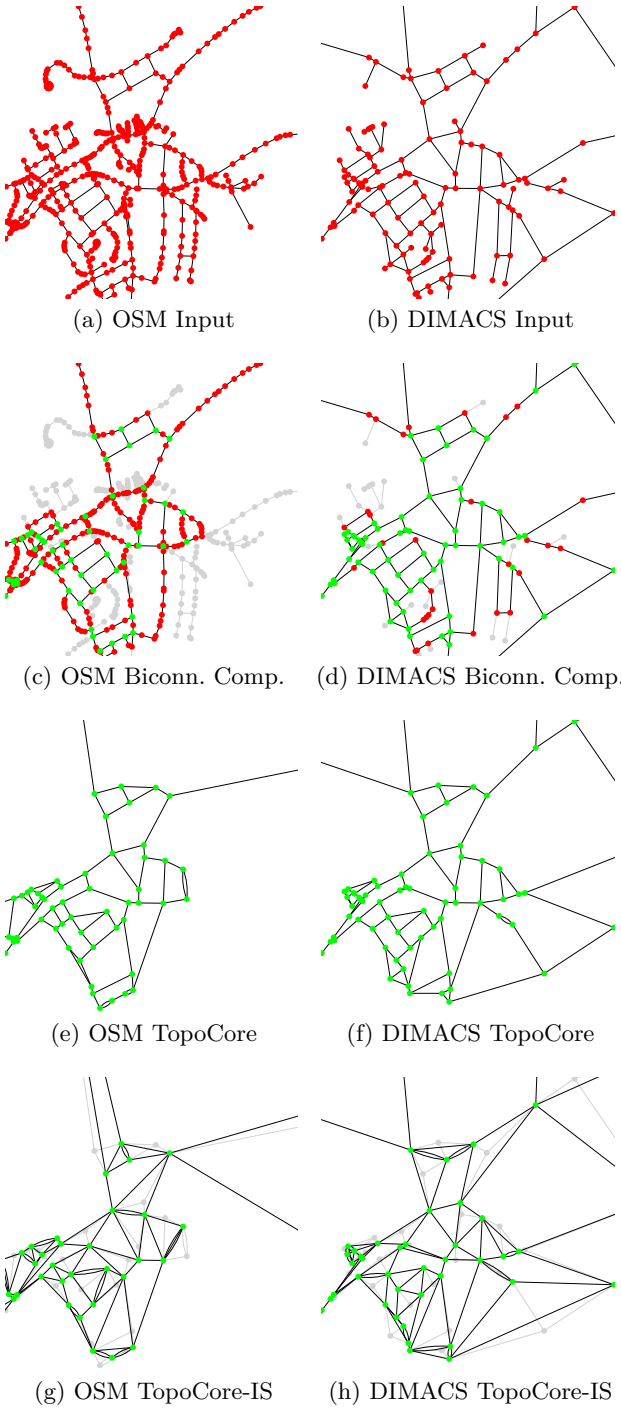
First, we compute the biconnected components of the input graph, employing a linear-time algorithm by Tarjan [17]. (For this, we ignore arc directions.) Each dead-end like structure is its own tiny component. All that entails significant routing decisions, forms a single large component. Hence, we keep every node in the core that is contained in the largest biconnected component. Note that we do not add any shortcuts in this step.

Step 2: Removing Chains.

Consider the graph induced by all core nodes. Note that removing a node with only two neighbors from the core, while adding shortcuts between its neighbors, does not increase core arc size. Better yet, in our inputs, such nodes are often not isolated but form chains between two nodes of higher degree. Moreover, these chains may grow by first applying Step 1, as intersections exist, where all but two roads lead to dead-ends. First removing dead-ends turns such intersections into degree 2 nodes. We identify such chains and add shortcut arcs to the core that bypass them, removing bypassed nodes from the core. Note that the resulting *TopoCore* may contain multi-arcs. The name was chosen to reflect that we exploit only topological graph features. See Figure 1 for an illustration.

Step 3: Removing Degree-3 Nodes.

Ideally, we would like to remove even more nodes from the core. In case of undirected simple graphs, removing a node of degree d (i. e., with d neighbors) from the core



(a) OSM Input (b) DIMACS Input

(c) OSM Biconn. Comp. (d) DIMACS Biconn. Comp.

(e) OSM TopoCore (f) DIMACS TopoCore

(g) OSM TopoCore-IS (h) DIMACS TopoCore-IS

Figure 1: OSM (left) and DIMACS (right) data sources of the area with a longitude in [8.50103, 8.52117] and latitude in [48.9476,48.9596]. Nodes are drawn at geographical position. Arcs are drawn without direction for clarity. Non-core nodes are red. Nodes not in the largest biconnected component are grayed out. Nodes in the TopoCore are green. The TopoCore-IS is drawn upon a grayed-out TopoCore, with added shortcuts between green nodes.

removes d edges (to these neighbors) from the core, while adding $d(d-1)/2$ new edges to the core, i. e., a net increase of $d(d-3)/2$. Hence for $d = 3$, the number of edges in the core remains unchanged but the number of nodes decreases. It is therefore beneficial to remove degree-3 nodes from the core for a reduction in queue operations during search. But we cannot just remove all of them, as removing a node may increase the degree of its neighbors, turning a degree-3 node into a higher degree node. Therefore, we first compute an independent set (IS) of degree-3 core nodes (iterating over nodes in DFS pre-order, greedily adding degree-3 nodes to the set, if they have no adjacent degree-3 node in the set). We then remove only this independent set from the core. See Figure 1 for an illustration of the resulting *TopoCore-IS*. One could apply this procedure iteratively, but experiments show that in the TopoCore-IS only few degree-3 nodes remain.

3.2 Query

Given a core graph we compute a forward and a backward search graph as follows: The forward graph G_F is the union of G and G_C without the arcs (u, v) that leave the core, i.e., $u \in V_C$ and $v \in V \backslash V_C$. The backward graph G_B is constructed analogously: First compute the union of G and G_C, then reverse the direction of every arc and finally remove the arcs leaving the core.

The query algorithm is a bidirectional variant of Dijkstra's algorithm [3]. The forward search is run on G_F while the backward search runs on G_B. Let d_F and d_B be the min-keys of the respective queue and μ be the tentative source–target distance: We abort the search if $d_F + d_B \geq \mu$ *and* no queue contains a non-core node.

To further improve query speed, we increase memory locality (and thus cache performance): During preprocessing, we first reorder the input graph in DFS pre-order, then compute the core and move core nodes to the front of the order.

4. RELATED WORK

Personalized Route Planning (PRP) was introduced by [10], where it is approached based on *k-path covers*. A k-path cover C is a small node subset of the original graph such that any simple (loop-free) path contains at most $k-1$ successive nodes that are not in C. The core idea for accelerating PRP queries consists of computing a coarsened path that only contains nodes in C where possible. Unfortunately, computing a minimum k-path cover is NP-hard [2]. For this reason in [10] approximate solutions were used.

The PRP problem is essentially a high-dimensional, linear multi-criteria search problem, related to the *parametric shortest path problem*. Extensions of known preprocessing techniques to multi-criteria optimization have been proposed, but were only evaluated experimentally for two criteria [12] and three criteria [11]. Especially for the latter, diminishing returns in query speed over preprocessing effort have been reported [11]. Related approaches include Pareto-SHARC [5] and Contraction Hierarchies with edge restrictions [13].

A different line of work [4, 6, 7] further subdivides the preprocessing phase into topology preprocessing and metric-dependent *customization*. The customization may consider a combination of costs as input, however, its output is a single fixed scalar metric, used by subsequent queries. If costs may change for *every* query, this customization approach does not pay off. More specifically, we show in [8] that it pays off only after eight or more queries on the same costs.

Table 1: Comparison to related work. We report the number of criteria (# cr.) considered, benchmark size, preprocessing and query time. Differences in OSM graph size of the same instance (marked by *) are likely due to different extraction dates.

Algorithm	# cr.	Instance	|V| [10^6]	|A| [10^6]	Prep. [s]	Query [ms]
CH [14]	1	Dimacs-Eur	18.0	42.2	165	0.152
FlexCH [12]	2	Dimacs-Eur	18.0	42.2	18 720	0.98
MultiCH [11]	2	OSM-BW*	2.5	5.0	121	0.42
MultiCH [11]	3	OSM-BW*	2.5	5.0	68	3.16
k-Pathc. [10]	8	OSM-BW**	2.2	4.6	12	35
k-Pathc. [10]	8	OSM-Ger**	17.7	36.1	149	249
TopoCore-IS	8	OSM-BW	3.1	6.2	3	9
TopoCore-IS	8	OSM-Ger	20.7	41.8	35	86
TopoCore-IS	8	OSM-Eur	174.8	348.0	657	558
TopoCore-IS	8	Dimacs-Eur	18.0	42.2	36	279
TopoCore-IS	8	Dimacs-US	23.9	57.7	43	386

5. EXPERIMENTS

We implemented our algorithms in C++, compiling on g++ 4.6.3 with optimization level `-O3`. Our experiments were performed on a *single core* of an Intel Xeon E5-2670 processor clocked at 2.6 GHz, with DDR3-1600 RAM clocked at 1.6 GHz, 20 MiB of L3 and 256 KiB of L2 cache. We consider five different instances: The continental road networks of Europe (DIMACS-Eur) and the US (DIMACS-US) from the 9th DIMACS Implementation Challenge on shortest paths; The continental road network of Europe (OSM-Eur), the national network of Germany (OSM-Ger) and the state network of Baden-Württemberg (OSM-BW), extracted from OpenStreetMap.[1] On these, we evaluate preprocessing and query performance of TopoCore-IS, the latter run with 1,000 pairs of source and target nodes picked uniformly at random.

Table 1 reports a performance comparison of TopoCore-IS and related work. While plain CH (single fixed criterion, i. e., travel time) yields query times more than three orders of magnitude faster than ours, performance quickly degrades when considering multiple criteria: While exact comparisons are difficult due to differences in benchmark instances, one observes that each additional criterion considered roughly decreases query speed by about an order of magnitude (0.152 ms $\rightarrow$ 0.98 ms, 0.42 ms $\rightarrow$ 3.16 ms). For the three criteria distance, travel time, and fuel costs, which are not completely uncorrelated, CH on OSM-BW [11] is already only factor 3–9 faster than our approach in terms of query times. This degradation of performance for more than two criteria likely means that the Contraction Hierarchies approach does not extend well to the PRP scenario considered in this work, an assessment also made by [10].

We also compare to the k-Path Cover approach of [10], which introduced the PRP scenario. Both theirs and our technique are core-based. However, by keeping every kth node in the core, for a fixed parameter k, their approach cannot flexibly handle dead-ends or chains of deg-2 nodes of vastly different length. It therefore adds more shortcuts than necessary, which slows down both preprocessing and

[1]For details on the instances, see `http://i11www.iti.kit.edu/resources/roadgraphs.php`.

Table 2: Node degree distribution of instances.

	# Nodes per degree				
	1	2	3	4	5+
OSM-BW	13.3%	**72.6%**	12.6%	1.2%	0.01%
OSM-Ger	14.2%	**70.9%**	13.5%	1.3%	0.01%
OSM-Eur	12.1%	**76.7%**	10.1%	1.1%	0.01%
DIMACS-Eur	26.5%	18.7%	49.1%	5.7%	0.1%
DIMACS-US	19.9%	30.3%	39.0%	10.7%	0.1%

queries. In contrast, our technique exploits such structures of the road network much more directly. This is confirmed by our experimental evaluation, which shows that on OSM-Ger and OSM-BW, the TopoCore-IS query is faster by a factor of 3 to 4, while having lower preprocessing overhead by about factor 4, even without considering the respective increase in OSM dataset size (OSM grows rapidly from month to month). Unfortunately, for their query experiments the authors of [10] focus exclusively on OSM graphs, hence we cannot compare on DIMACS graphs.

OpenStreetMap Instances are Easier than Expected.

Besides solving Personalized Route Planning faster than before, we also find the comparison of OSM and DIMACS instances very interesting. Consider the three instances OSM-Ger, DIMACS-Eur and DIMACS-US: They are very similar in graph size. Moreover, the experimental results presented in Table 1 show that preprocessing correlates quite well with the size of the graph. However, query times achieved by TopoCore-IS are significantly faster on the OSM-based graph. A similar effect has been observed by [4]: The speedup of their technique over Dijkstra's algorithm is up to 14.2 times higher on OSM than on non-OSM graphs of comparable size.

This is an important property, which we attribute to the significantly higher number of degree-2 nodes in instances obtained from OSM (where they are used to model path geometry, which is not important for routing). Table 2 shows the vastly skewed node degree distribution of OSM graphs.

These observations suggest that, for a fixed graph size, OSM-based instances are easier for speedup techniques than instances from other data sources—unless some kind of normalization is applied (e. g., elimination of all degree-2 paths). This should generally be taken into account when comparing different route planning techniques that were experimentally evaluated on road networks of different origin.

6. CONCLUSIONS

We proposed a new preprocessing-based speedup technique for faster Personalized Route Planning. In this approach, weighting of routing criteria can individually be adjusted for every user and *every query* in a very flexible way, without per-user storage overhead. Preprocessing needs only be repeated when roads are build or cost vectors are adjusted, e. g., a new speed limit is posted. We experimentally evaluated our technique on datasets both from OpenStreetMap and the 9th DIMACS Implementation Challenge, showing good performance on a large range of instances, achieving query speeds well below a second even on large continental road networks.

7. REFERENCES

[1] H. Bast, D. Delling, A. V. Goldberg, M. Müller–Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck. Route planning in transportation networks. Technical Report abs/1504.05140, ArXiv e-prints, 2015.

[2] B. Bresar, F. Kardos, J. Katrenic, and G. Semanisin. Minimum k-path vertex cover. *Discrete Applied Mathematics*, 159(12):1189–1195, 2011.

[3] G. B. Dantzig. *Linear Programming and Extensions.* Princeton University Press, 1962.

[4] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Customizable route planning in road networks. *Transportation Science*, 2015.

[5] D. Delling and D. Wagner. Pareto paths with SHARC. In *Proceedings of the 8th International Symposium on Experimental Algorithms (SEA'09)*, LNCS 5526, pages 125–136. Springer, 2009.

[6] J. Dibbelt, B. Strasser, and D. Wagner. Customizable contraction hierarchies. In *Proceedings of the 13th International Symposium on Experimental Algorithms (SEA'14)*, LNCS 8504, pages 271–282. Springer, 2014.

[7] J. Dibbelt, B. Strasser, and D. Wagner. Customizable contraction hierarchies. Technical Report abs/1402.0402, ArXiv e-prints, 2014.

[8] J. Dibbelt, B. Strasser, and D. Wagner. Fast exact shortest path and distance queries on road networks with parametrized costs. Technical Report abs/1509.03165, ArXiv e-prints, 2015.

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

[10] S. Funke, A. Nusser, and S. Storandt. On k-path covers and their applications. In *Proceedings of the 40th International Conference on Very Large Databases (VLDB 2014)*, pages 893–902, 2014.

[11] S. Funke and S. Storandt. Polynomial-time construction of contraction hierarchies for multi-criteria objectives. In *Proceedings of the 15th Meeting on Algorithm Engineering and Experiments (ALENEX'13)*, pages 31–54. SIAM, 2013.

[12] R. Geisberger, M. Kobitzsch, and P. Sanders. Route planning with flexible objective functions. In *Proceedings of the 12th Workshop on Algorithm Engineering and Experiments (ALENEX'10)*, pages 124–137. SIAM, 2010.

[13] R. Geisberger, M. N. Rice, P. Sanders, and V. J. Tsotras. Route planning with flexible edge restrictions. *ACM Journal of Experimental Algorithmics*, 17(1), 2012.

[14] R. Geisberger, P. Sanders, D. Schultes, and C. Vetter. Exact routing in large road networks using contraction hierarchies. *Transportation Science*, 46(3):388–404, 2012.

[15] F. Schulz, D. Wagner, and K. Weihe. Dijkstra's algorithm on-line: An empirical case study from public railroad transport. *ACM Journal of Experimental Algorithmics*, 5(12):1–23, 2000.

[16] C. Sommer. Shortest-path queries in static networks. *ACM Computing Surveys*, 46(4), 2014.

[17] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1972.