Efficient Routing in Road Networks with Turn Costs

Robert Geisberger and Christian Vetter

Karlsruhe Institute of Technology, 76128 Karlsruhe, Germany
{geisberger,christian.vetter}@kit.edu

Abstract. We present an efficient algorithm for shortest path computation in road networks with turn costs. Each junction is modeled as a node, and each road segment as an edge in a weighted graph. Turn costs are stored in tables that are assigned to nodes. By reusing turn cost tables for identical junctions, we improve the space efficiency. Preprocessing based on an augmented node contraction allows fast shortest path queries. Compared to an edge-based graph, we reduce preprocessing time by a factor of 3.4 and space by a factor of 2.4 without change in query time.

Key words: route planning; banned turn; turn cost; algorithm engineering

1 Introduction

Route planning in road networks is usually solved by computing shortest paths in a suitably modeled graph. Each edge of the graph has an assigned weight representing, for example, the travel time. There exists a plethora of speed-up techniques to compute shortest paths in such weighted graphs [1]. The most simple model maps junctions to nodes and road segments to edges. However, this model does not consider turn costs. Turn costs are important to create a more realistic cost model and to respect banned turns.

To incorporate turn costs, usually a pseudo-dual of the simple model is used [2,3], modeling road segments as nodes and turns between two consecutive road segments as edges. Thus the edges in the simple model become nodes in the pseudo-dual. Therefore, we will refer to the result of the simple model as *node-based graph* and to the pseudo-dual as *edge-based graph*. The advantage of the edge-based graph is that no changes to the speed-up techniques are required to compute shortest paths, as only edges carry a weight. The drawback is a significant blowup in the number of nodes compared to the *node-based graph*. To avoid this blowup, we will extend the node-based graph by assigning *turn cost tables* to the nodes, i.e., junctions, and show how to efficiently perform precomputation for a major speed-up technique. We further reduce the space consumption by identifying junctions that can share the same turn cost table.

Related Work

There is only little work on speed-up techniques with respect to turn costs. The main reason is that incorporating them is seen as redundant due to the usage of an edge-based graph [2,3].

Speed-up techniques for routing in road networks can be divided into hierarchical approaches, goal-directed approaches and combinations of both. Delling et al. [1] provide a recent overview of them. In this paper, we focus on the technique of *node contraction* [4,5,6]. It is used by the most successful speed-up techniques known today [6,7,8]. The idea is to remove unimportant nodes and to add shortcut edges (*shortcuts*) to preserve shortest path distances. Then, a bidirectional Dijkstra finds shortest paths, but never needs to relax edges leading to less important nodes. Contraction hierarchies (CH) [6] is the most successful hierarchical speed-up technique using node contraction; it contracts in principle one node at a time. Node contraction can be further combined with goal-directed approaches to improve the overall performance [7]. The performance of CH on edge-based graphs has been studied by Volker [9].

2 Preliminaries

We have a *graph* $G = (V, E)$ with *edge weight function* $c : E \to \mathbb{R}_+$ and *turn cost function* $c^{\mathrm{t}} : E \times E \to \mathbb{R}_+ \cup \{\infty\}$. An edge $e = (v, w)$ has source node v and target node w. A turn with cost ∞ is *banned*. A path $P = \langle e_1, \ldots, e_k \rangle$ must not contain banned turns. The *weight* is $c(P) = \sum_{i=1}^{k} c(e_i) + \sum_{i=1}^{k-1} c^{\mathrm{t}}(e_i, e_{i+1})$. The problem is to compute a path with smallest weight between e_1 and e_k, that is a *shortest path*. Note that source and target of the path are edges and *not* nodes.

To compute shortest paths with an algorithm that cannot handle a turn cost function, the *edge-based* graph [3] $G' = (V', E')$ is used with $V' = E$ and $E' = \{(e, e') \mid e, e' \in E$, target node of e is source node of e' and $c^{\mathrm{t}}(e, e') < \infty\}$. We define the edge weight by $c' : (e, e') \mapsto c(e') + c^{\mathrm{t}}(e, e')$. Note that the cost of a path $P = \langle e_1, \ldots, e_k \rangle$ in the edge-based graph misses the cost $c(e_1)$ of the first edge. Nevertheless, as each path between e_1 and e_k misses this, shortest path computations are still correct.

To compute a shortest path in the edge-based graph, any shortest path algorithm for non-negative edge weights can be used. E. g., Dijkstra's algorithm

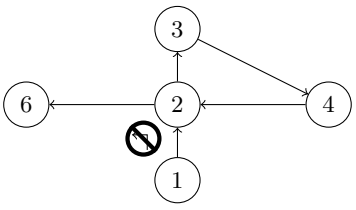

Fig. 1. Graph (unit distance) that restricts the turn $1 \to 2 \to 6$. Therefore, the shortest path $1 \to 2 \to 3 \to 4 \to 2 \to 6$ visits node 2 twice.

computes shortest paths from a single source node to all other nodes by settling nodes in non-decreasing order of shortest path distance. However, on a node-based graph with turn cost function, settling nodes is no longer sufficient, see Figure 1. Instead, we need to settle edges, i.e., nodes in the edge-based graph. Initially, for source edge e_1 the tentative distance $\delta(e_1)$ is set to $c(e_1)$, all other tentative distances are set to infinity. In each iteration the unsettled edge $e = (v, w)$ with smallest tentative distance is *settled*. Then, for each edge e' with source w, the tentative distance $\delta(e')$ is updated with $\min\big(\delta(e'), \delta(e) + c^{\mathrm{t}}(e, e') + c(e')\big)$. This resulting *edge-based Dijkstra* successfully computes shortest paths. By storing parent pointers, shortest paths can be reconstructed in the end.

3 Turn Cost Model

We take into account two kinds of turn costs: a limited turning speed inducing a turn cost and a fixed cost added under certain circumstances. Banned turns have infinite turn cost.

3.1 Fixed cost

We add a fixed turning cost when turning left or right. In addition to this a fixed cost is applied when crossing a junction with traffic lights.

3.2 Turning speed

Tangential acceleration. We can use the turn radius and a limit on the tangential acceleration a to compute a limit on the turning speed v: $\max_v = \sqrt{\max_a * \text{radius}}$. Given a lower limit on the resolution δ_{min} of the underlying data we estimate the resolution δ the turn is modeled with: When turning from edge e into edge e' the respective edge length are l and l'. Then, δ is estimated as the minimum of l, l' and δ_{min}. We compute the turn radius from the angle α between the edges: $\text{radius} = \tan(\alpha/2) * \delta/2$.

Traffic. When turning into less important road categories we restrict the maximum velocity to simulate the need to look out for traffic. We differentiate between left and right turns, e.g. it might not be necessary to look out for incoming traffic when turning right. Furthermore, we limit the maximum turning speed when pedestrians could cross the street.

Turn costs. We can derive turn costs from the turn speed limit $\max_v$. Consider a turn between edges e and e'. When computing $c(e)$ and $c(e')$ we assumed we could traverse these edges at full speed v and v'. When executing the turn between them, we now have to take into account the deceleration and acceleration process. While traversing edge e we assume deceleration a_{dec} from v down to $\max_v$ at the latest point possible and while traversing edge e' we assume immediate start of acceleration a_{acc} from $\max_v$ to v'. The turn cost we apply is the difference

between this time and the projected travel time on the edges without acceleration and deceleration. The resulting turn cost is $c^{\mathrm{t}}(e, e') = (v - \max_v)^2/(2 * a_{\mathrm{dec}}) + (v' - \max_v)^2/(2 * a_{\mathrm{acc}})$. Of course, this is only correct as long as the edge is long enough to fully accelerate and decelerate.

4 Node Contraction

Node contraction without turn costs was introduced in an earlier paper [6]. The basic idea behind the contraction of a node v is to add shortcuts between the neighbors of v to preserve shortest path distances. In a graph without turn costs, this is done by checking for each incoming edge (u, v) from remaining neighbor u and for each outgoing edge (v, w) to remaining neighbor w, whether the path $\langle u, v, w\rangle$ is the only shortest path between u and w. If this is the case, then a *shortcut* (u, w) representing this path needs to be added. This is usually decided using a node-based Dijkstra search (*local search*) starting from node u. The neighbors u and w are more important than node v, as they are not contracted so far. A query algorithm that reaches node u or w never needs to relax an edge to the less important node v, as the shortcut can be used. The query is bidirected and meets at the most important node of a shortest path. This shortest path $P' = \langle e_1, e_2, \ldots, e_k\rangle$ found by the query can contain shortcuts. To obtain the path P in the original graph, consisting only of *original edges*, each shortcut e' needs to be replaced by the path $\langle e'_1, \ldots, e'_\ell\rangle$ it represents.

4.1 With Turn Costs

Turn restrictions complicate node contraction. As we showed in Section 2, we need an edge-based query instead of a node-based one. Therefore, we have to preserve shortest path distances between edges, and not nodes. An important observation is that it is sufficient to preserve shortest path distances only between original edges. This can be exploited during the contraction of node v if the incoming edge (u, v) and/or the outgoing edge (v, w) is a shortcut. Assume that (u, u') is the *first* original edge of the path represented by (u, v) and (w', w) is the *last* original edge of the path represented by (v, w). We do not need to add a shortcut for $\langle (u, v), (v, w)\rangle$ if it does not represent a shortest path between (u, u') and (w', w) in the original graph. The weight of the shortcut is the sum of the weights of the original edges plus the turn costs between the original edges.

We introduce the following notation: A shortcut $(u \to u', w' \to w)$ is a shortcut between nodes u and w, the first original edge is (u, u') and the last original edge is (w', w). If two nodes are connected by an arrow, e. g., $u \to u'$, then this always represents an original edge (u, u'). A node-triplet connected by arrows, e. g., $u'' \to u \to u'$, always represents a turn between the original edges (u'', u) and (u, u') with cost $c^{\mathrm{t}}(u'' \to u \to u')$.

Local search using original edges. Now that we have established the basic idea of node contraction in the presence of turn costs, we will provide more details.

An observation is that we cannot avoid parallel edges and loops between nodes in general, if they have different first or last original edge. Therefore, we can only uniquely identify an edge by its endpoints *and* the first and last original edge. Loops at a node v make the discovery of potentially necessary shortcuts more complicated, as given an incoming edge $(u \to u', v' \to v)$ and outgoing edge $(v \to v'', w' \to w)$, the potential shortcut $(u \to u', w' \to w)$ may not represent $\langle (u \to u', v' \to v), (v \to v'', w' \to w) \rangle$ but has to include one or more loops at v in between. This can happen, e. g., in Figure 1, if nodes 2, 3 and 4 are contracted, then there has to be a shortcut between nodes 1 and 6 including a loop. Therefore, we use the local search to not only to decide on the necessity of a shortcut, but also to find them. The local search from incoming edge $(u \to u', v' \to v)$ computes tentative distances $\delta(\cdot)$ for *original* edges only. Initially, for each remaining edge $(u \to u', x' \to x)$ with first original edge $u \to u'$, $\delta(x' \to x) := c(u \to u', x' \to x)$, and all other distances are set to infinity. To settle an original edge $x' \to x$, for each edge $e' = (x \to x'', y' \to y)$ with source x, the tentative distance $\delta(y' \to y)$ is updated with $\min\big(\delta(y' \to y), \delta(x' \to x) + c^{\mathrm{t}}(x' \to x \to x'') + c(e')\big)$. A shortcut $(u \to u', w' \to w)$ is added iff the path computed to $w' \to w$ only consists of the incoming edge from u, the outgoing edge to w and zero or more loops at v in between. Otherwise a *witness* is found of smaller or equal weight. The weight of the shortcut is $\delta(w' \to w)$.

4.2 Optimizations

The algorithm described so far preserves shortest path distances between all remaining uncontracted nodes. However, as our query already fixes the first and last original edge of the shortest path, we can further reduce the number of shortcuts. It would be sufficient to only add a shortcut $(u \to u', w' \to w)$ if there are two original edges, a source edge (u'', u) and a target edge (w, w'') such that $\langle (u'', u), (u \to u', w' \to w), (w, w'') \rangle$ would be only shortest path in the remaining graph together with (u'', u) and (w, w'') but without node v. This allows to avoid a lot of unnecessary and 'unnatural' shortcuts. E. g., a query starts from a southbound edge of a highway but the journey should go north. Naturally, one would leave at the first exit, usually the target of the edge, and reenter the highway northbound. Our improvement allows to avoid shortcuts representing such changes of direction.

Aggressive local search. We can use the above observation to enhance the local search in a straightforward manner. Instead of executing a local search from the original edge (u, u'), we perform a local search separately from each original incoming edge (u'', u). Then, we check for each original edge (w, w'') whether the shortcut is necessary. While this approach can reduce the number of shortcuts, it increases the number of local searches, and therefore the preprocessing time.

Turn replacement. To avoid performing a large amount of local queries we try to combine the searches from all the original edges incoming to u into one search. We cannot start from all these original edges simultaneously while still computing

just a single distance per original edge. It is better to start from all original edges outgoing from u simultaneously. We initialize the local search as in Section 4.1. Furthermore, we consider all other remaining edges $(u \to u'_2, x' \to x)$ outgoing from u. However, as we now replace a turn $u'' \to u \to u'$ by a turn $u'' \to u \to u'_2$, an *outgoing turn replacement difference* $\overrightarrow{r}(u \to u', u'_2) := \max_{u''} \left(c^{\mathrm{t}}(u'' \to u \to u'_2) - c^{\mathrm{t}}(u'' \to u \to u')\right)$ needs to be added to account for the different turn costs, see Figure 2. Note that we consider the worst increase in the turn cost over all incoming original edges of u. So $\delta(x' \to x) := \overrightarrow{r}(u \to u', u'_2) + c(u \to u'_2, x' \to x)$. The local search settles original edges as before, but has a different criterion to add shortcuts. We add a shortcut $(u \to u', w' \to w)$ with weight $\delta(w' \to w)$ iff the path computed to $w' \to w$ only consists of the incoming edge $(u \to u', v' \to v)$, the outgoing edge to $(v'' \to v, w' \to w)$ and zero or more loops at v in between, *and* none of the other edges incoming to w offers a witness. Consider a path computed to an original edge $w'_2 \to w$ incoming to node w. If we consider this path instead of the one computed to $w' \to w$, we would replace the turn $w' \to w \to w''$ by the turn $w'_2 \to w \to w''$. A *incoming turn replacement difference* $\overleftarrow{r}(w'_2, w' \to w) := \max_{w''} \left(c^{\mathrm{t}}(w'_2 \to w \to w'') - c^{\mathrm{t}}(w' \to w \to w'')\right)$ is required to account for the different turn costs. We do not need to add a shortcut if $\overleftarrow{r}(w'_2, w' \to w) + \delta(w'_2 \to w) < \delta(w' \to w)$.

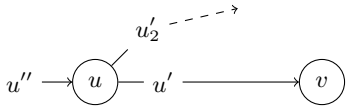

Fig. 2. If a witness uses turn $u'' \to u \to u'_2$ instead of $u'' \to u \to u'$, we have to account for the turn cost difference $c^{\mathrm{t}}(u'' \to u \to u'_2) - c^{\mathrm{t}}(u'' \to u \to u')$.

Loop avoidance. Even with the turn replacement approach of the previous paragraph, there can still be a lot of unnecessary loop shortcuts. E. g., in Figure 1, assume that nodes 1 and 6 are not present. After the contraction of nodes 3 and 4, there would be a loop shortcut at node 2 although it is never necessary. We only need to add a loop shortcut, if it has smaller cost than a direct turn. That is, considering a loop shortcut $(u \to u', u'' \to u)$ at node u, if there are neighbors u'_2 and u''_2 such that $c^{\mathrm{t}}(u'_2 \to u \to u') + c(u \to u', u'' \to u) + c^{\mathrm{t}}(u'' \to u \to u''_2) < c^{\mathrm{t}}(u'_2 \to u \to u''_2)$.

Limited local searches. Without turn costs, local searches only find witnesses to avoid shortcuts. Therefore, they can be arbitrarily pruned, as long as the computed distances are upper bounds on the shortest path distances [6]. That ensures that all necessary, and maybe some superfluous shortcuts are added. But with turn costs, a local search also needs to find the necessary shortcuts. Therefore, we cannot prune the search for those. We limit local searches by the number of settled original edges. Once we settled a certain number, we

only settle original edges whose path from the source is a prefix of a potential shortcut. Furthermore, if all reached but unsettled original edges cannot result in a potential shortcut, we prune the whole search.

5 Preprocessing

To support turn costs, the CH preprocessing [6] only needs to use the enhanced node contraction described in Section 4. The preprocessing performs a contraction of all nodes in a certain order. The original graph together with all shortcuts is the result of the preprocessing. The order in which the nodes are contracted is deduced from a node priority consisting of: (a) The edge quotient, i.e., the quotient between the amount of shortcuts added and the amount of edge removed from the remaining graph. (b) The original edge quotient, i.e., the quotient between the number of original edges represented by the shortcuts and the number of original edges represented by the edges removed from the remaining graph. (c) The hierarchy depth, i.e., an upper bound on the amount of hops that can be performed in the resulting hierarchy. Initially, we set $\text{depth}(u) = 0$ and when a node v is contracted, we set $\text{depth}(u) = \max(\text{depth}(u),\text{depth}(v)+1)$ for all neighbors u. We weight (a) with 8, (b) with 4 and (c) with 1 in a linear combination to compute the node priorities. Nodes with higher priority are more important and get contracted later. The nodes are contracted in parallel by computing independent node sets with a 2-neighborhood [10].

6 Query

The query computes a shortest path between two original edges, a source $s \to s'$ and a target $t' \to t$. It consists of two Dijkstra-like searches that settle original edges (cf. Section 2) one in forward direction starting at $s \to s'$, and one in backward direction starting at $t' \to t$. The only restriction is that it never relaxes edges leading to less important nodes. Both search scopes meet at the most important node z of a shortest path. E. g., the forward search computes a path to $z' \to z$, and the backward search computes a path to $z \to z''$. As usually $z' \neq z''$, when we settle an original edge $x' \to x$ in forward direction, we need to check whether the backward search reached any outgoing edge $x \to x''$, and vice versa. Such a path with smallest weight among all meeting nodes is a shortest path.

Stall-on-demand. As our search does not relax all edges, it is possible that an original edge $x' \to x$ is settled with suboptimal distance. In this case, we can prune the search at this edge, since the computed path cannot be part of a shortest path. To detect some of the suboptimally reached edges, the *stall-on-demand technique* [5] can be used, but extended to the scenario with turn costs: The edge $x' \to x$ is settled suboptimally if there is an edge $(y \to y', x' \to x)$ and an original edge $y'' \to y$ such that $\delta(y'' \to y) + c^{\text{t}}(y'' \to y \to y') + c(y \to y', x' \to x) < \delta(x' \to x)$.

Path unpacking. To unpack the shortcuts into original edges, we can store the *middle node* whose contraction added the shortcut. Then, we can recursively unpack shortcuts [6]. Note that if there are loops at the middle node, they may be part of the shortcut. A local search that only relaxes original edges incident to the middle node can identify them.

7 Turn Cost Tables

We can store the turn cost function $c^{\mathrm{t}} : E \times E \to \mathbb{R}_+ \cup \{\infty\}$ efficiently using a single table per node. Each adjacent incoming and outgoing edge gets assigned a local identifier that is used to access the table. To avoid assigning bidirectional edges two identifiers we take care to assign them the same one in both directions. This can easily be achieved by assigning the bidirectional edges the smallest identifiers.

To look up the correct turn costs in the presence of shortcuts we need to store additional information with each shortcut: A shortcut $(u \to u', w' \to w)$ has to store the identifier of (u, u') at u and the identifier of (w', w) at w.

Storing these identifiers does not generate much overhead as their value is limited by the degree of the adjacent node.

7.1 Redundancy

Since the turn cost tables model the topology of road junction they tend to be similar. In fact many tables model exactly the same set of turn costs. We can take advantage of this by replacing those instances with a single table. To further decrease the amount of tables stored we can rearrange the local identifiers of a table to match another table. Of course, we have to take care to always assign bidirectional edges the smallest identifiers.

Given a reference table t and a table t' we check whether t' can be represented by t by trying all possible permutations of identifiers. Bidirectional identifiers are only permuted amongst themselves. Because the amount of possible permutations increases exponentially with the table size we limit the number of permutations tested. Most junctions only feature a limited amount of adjacent edges and are not affected by this pruning. Nevertheless, it is necessary as the data set may contain erroneous junctions with large turn cost tables.

To avoid checking a reference table against all other tables we compute hash values $h(t)$ for each table t. $h(t)$ has the property that if $h(t) \neq h(t')$ neither t can be represented by t' nor t' by t. We compute $h(t)$ as follows: First, we sort each row of the table, then sorting the rows lexicographically. Finally, we compute a hash value from the resulting sequence of values.

We use this hash values to greedily choose an unmatched table and match as many other tables to it as possible.

8 Experiments

Environment. The experimental evaluation was done on a machine with four AMD Opteron 8350 processors (Quad-Core) clocked at 2 GHz with 64 GiB of RAM and 2 MiB of Cache running SuSE Linux 11.1 (kernel 2.6.27). The program was compiled by the GNU C++ compiler 4.3.2 using optimization level 3.

Instances. We use three road networks derived from the publicly available data of OpenStreetMap, see Table 1. Travel times where computed using the MoNav Motorcar Profile [11]. Using the node-based model with turn cost tables requires about 30% less space than the edge-based model. That is despite the fact that in the node-based model, we need more space per node and edge: Per node, we need to store an additional pointer to the turn cost table (4 Bytes), and an offset to compute a global identifier from the identifier of an original edge (u, u') local to a node u (4 Bytes). Per edge, we need to additionally store the local identifier of the first and last original edge (2×1 Byte rounded to 4 Byte due to alignment).

Table 1. Input instances. In the edge-based model, a node requires 4 Bytes (pointer in edge array), and an edge requires 8 Bytes (target node + weight + flags). In the node-based model, a node requires additional 8 Bytes (pointer to turn cost table + offset to address original edges), and an edge requires additional 4 Bytes (first and last original edge). An entry in a turn cost table requires 1 Byte.

graph	model	nodes		edges		turn cost tables		
		[$\times 10^6$]	[MiB]	[$\times 10^6$]	[MiB]	[$\times 10^3$]	%	[MiB]
Netherlands	node-based	0.8	9.4	1.9	22.2	79	9.9%	0.8
	edge-based	1.9	7.4	5.2	39.7	-	-	-
Germany	node-based	3.6	41.3	8.5	97.1	267	7.4%	3.1
	edge-based	8.5	32.4	23.1	176.3	-	-	-
Europe	node-based	15.0	171.1	35.1	401.3	834	5.6%	9.5
	edge-based	35.1	133.8	95.3	727.0	-	-	-

Redundant turn cost tables. Already for the Netherlands, only one table per ten nodes needs to be stored. The best ratio is for the largest graph, Europe, with one table per 18 nodes. This was to be expected as most unique junctions types are already present in the smaller graphs. Identifying the redundant tables is fast, even for Europe, it took only 20 seconds.

Node Contraction. Preprocessing is done in parallel on all 16 cores of our machine. We compare the node contraction in the node-based and the edge-based model in Table 2. In the node-based model, we distinguish between the *basic* contraction without the optimizations of Section 4.2, the *aggressive* contraction mentioned in Section 4.2, and the contraction using turn replacement (TR) and

loop avoidance (LA). Clearly, TR+LA is the best contraction method. The basic contraction requires about a factor 3–4 times more preprocessing time, about 5–7 times more space, and results in 3–4 times slower queries. It misses a lot of witnesses which leads to denser remaining graphs, so that its preprocessing is even slower than the aggressive contraction's. The aggressive contraction finds potentially more witnesses as TR+LA, but shows no significant improvement, neither in preprocessing space nor query performance. For Europe, its performance even slightly decreases, potentially due to the limited local searches and because a different node order is computed. Furthermore, its preprocessing is about a factor 3 slower, because we execute several local searches per neighbor with an edge incoming to the contracted node.

Table 2. Performance of contraction hierarchies (TR = turn replacement, LA = loop avoidance).

graph	model	contraction	preprocessing time [s]	preprocessing space [MiB]	preprocessing space %	query time [μs]	query settled edges
Netherlands	node-based	basic	66	31.9	144%	1 177	713
		aggressive	57	7.0	32%	319	367
		TR + LA	19	7.0	32%	315	363
	edge-based	regular	63	46.6	117%	348	358
Germany	node-based	basic	250	124.2	128%	2 339	1 158
		aggressive	244	17.3	18%	735	594
		TR + LA	73	17.3	18%	737	597
	edge-based	regular	253	183.9	104%	751	535
Europe	node-based	basic	1 534	592.2	148%	4 075	1 414
		aggressive	1 318	117.4	29%	1 175	731
		TR + LA	392	116.1	29%	1 107	651
	edge-based	regular	1308	817.1	112%	1061	651

We will compare the contraction in the edge-based model only to the contraction in the node-based model using TR+LA. Its preprocessing is about 3.4 times faster than in the edge-based model. One reason is that there are about 2.3 fewer nodes need to be contracted, and TR+LA, compared to the aggressive contraction, needs only one local search per incoming edge. We argue that the additional speed-up comes from the fact that contracting junctions instead of road segments works better. Note that there is a fundamental difference in contracting a node in the node-based and edge-based model: Adding a shortcut in the node-based model would map to an additional node in the edge-based model. We observe that the total space required including preprocessed data is about a factor 2.4 larger for the edge-based model.

Furthermore, in the node-based model, bidirected road segments can be stored more efficiently by using forward/backward flags. In comparison, assume that you have a bidirected edge in the original edge-based graph. This implies

that the edge represents two U-turns between (u, v) and (v, u), see Figure 3. Therefore, bidirected road segments cannot be stored efficiently in the edge-based model.

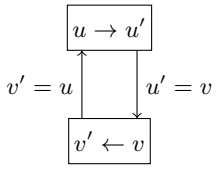

Fig. 3. A bidirected edge in the original edge-based graph between two original edges (u, u') and (v, v') in the node-based graph. Because a turn from (u, u') to (v, v') is possible, $u' = v$, and because a turn from (v, v') to (u, u') is possible, $v' = u$. Therefore, both turns are U-turns.

Query. Query performance is averaged over 10 000 shortest path distance queries run sequentially on a single core of our machine. Source and target edge have been selected uniformly at random. The resulting distances were compared to a plain edge-based Dijkstra for correctness. Interestingly, the best query times that can be achieved in both models are almost the same. One reason might be that both queries settle original edges. For the smaller graphs the query time is even a bit faster in the node-based model, because most of the turn cost tables fit into cache, thus causing almost no overhead.

9 Conclusions

Our work shows the advantages of the node-based model over the edge-based one. The node-based model stores tables containing the turn costs. By identifying redundant turn cost tables, we can decrease the space required to store them by one order of magnitude. Still, our query has to settle original edges so that we need to store a local identifier per edge and an offset to obtain a global identifier per node. Therefore, a query in the original node-based graph is the same as in the original edge-based graph, but storing the graph requires 30% less space.

Our preprocessing based on node contraction works better in the node-based model in terms of preprocessing time (factor ≈ 3.4) and space (factor ≈ 2.4) without affecting the query time. To augment the node-based contraction to turn cost tables, we had to augment the local searches to not only identify witnesses, but also shortcuts, because parallel and loop shortcuts can be necessary. To restrict the node contraction to one local search per incoming edge (factor ≈ 3 faster) without missing too many witnesses, we developed the techniques of turn replacement and loop avoidance.

9.1 Future Work

We want to integrate the turn cost tables into an existing mobile implementation of contraction hierarchies [12]. To further reduce the space requirements of the turn cost tables, we can approximate their entries. This not only reduces the number of different turn cost tables we need to store, but also the bits required to store a table entry.

Acknowledgement. The authors would like to thank Dennis Luxen for his valuable comments and suggestions.

References

1. Delling, D., Sanders, P., Schultes, D., Wagner, D.: Engineering Route Planning Algorithms. In Lerner, J., Wagner, D., Zweig, K.A., eds.: Algorithmics of Large and Complex Networks. Volume 5515 of Lecture Notes in Computer Science. Springer (2009) 117–139
2. Caldwell, T.: On Finding Minimum Routes in a Network With Turn Penalties. Communications of the ACM **4**(2) (1961)
3. Winter, S.: Modeling Costs of Turns in Route Planning. GeoInformatica **6**(4) (2002) 345–361
4. Sanders, P., Schultes, D.: Highway Hierarchies Hasten Exact Shortest Path Queries. In: Proceedings of the 13th Annual European Symposium on Algorithms (ESA'05). Volume 3669 of Lecture Notes in Computer Science., Springer (2005) 568–579
5. Schultes, D., Sanders, P.: Dynamic Highway-Node Routing. In Demetrescu, C., ed.: Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07). Volume 4525 of Lecture Notes in Computer Science., Springer (June 2007) 66–79
6. Geisberger, R., Sanders, P., Schultes, D., Delling, D.: Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In McGeoch, C.C., ed.: Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08). Volume 5038 of Lecture Notes in Computer Science., Springer (June 2008) 319–333
7. Bauer, R., Delling, D., Sanders, P., Schieferdecker, D., Schultes, D., Wagner, D.: Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. ACM Journal of Experimental Algorithmics **15**(2.3) (January 2010) 1–31 Special Section devoted to WEA'08.
8. Abraham, I., Delling, D., Goldberg, A.V., Werneck, R.F.: A Hub-Based Labeling Algorithm for Shortest Paths on Road Networks. In Pardalos, P.M., Rebennack, S., eds.: Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11). Lecture Notes in Computer Science, Springer (2011) To appear.
9. Volker, L.: Route Planning in Road Networks with Turn Costs (2008) Student Research Project. http://algo2.iti.uni-karlsruhe.de/documents/routeplanning/volker_sa.pdf.
10. Vetter, C.: Parallel Time-Dependent Contraction Hierarchies (2009) Student Research Project. http://algo2.iti.kit.edu/download/vetter_sa.pdf.
11. Vetter, C.: MoNav. http://code.google.com/p/monav/ (2011)
12. Vetter, C.: Fast and Exact Mobile Navigation with OpenStreetMap Data. Master's thesis, Karlsruhe Institute of Technology (2010)