Dynamic Time-Dependent Route Planning in Road Networks with User Preferences

Moritz Baum[1], Julian Dibbelt[1](✉), Thomas Pajor[2], and Dorothea Wagner[1]

[1] Karlsruhe Institute of Technology, Karlsruhe, Germany
{moritz.baum,julian.dibbelt,dorothea.wagner}@kit.edu
[2] Cupertino, CA, USA

Abstract. Algorithms for computing driving directions on road networks often presume constant costs on each arc. In practice, the current traffic situation significantly influences the travel time. One can distinguish traffic congestion that can be predicted using historical traffic data, and congestion due to unpredictable events, e. g., accidents. We study the *dynamic and time-dependent* route planning problem, which takes both live traffic and long-term prediction into account. We propose a practical algorithm that, while robust to user preferences, is able to integrate global changes of the time-dependent metric faster than previous approaches and allows queries in the order of milliseconds.

1 Introduction

To enable responsive route planning applications on large-scale road networks, *speedup techniques* have been proposed [1], employing *preprocessing* to accelerate Dijkstra's shortest-path algorithm [18]. A successful approach [4,9,16,21,28,30] exploits that road networks have small separators [10,22,27,40,41], computing coarsened overlays that maintain shortest path distance. An important aspect [14] in practice is the consideration of traffic patterns and incidents. In *dynamic, time-dependent* route planning, costs vary as a function of time [6,19]. These functions are derived from historic knowledge of traffic patterns [39], but have to be updated to respect traffic incidents or short-term predictions [15]. In this work, we investigate the challenges that arise when extending a separator-based overlay approach to the dynamic, time-dependent route planning scenario.

Related Work. In time-dependent route planning, there are two major query variants: (1) Given the departure time at a source, compute the *earliest arrival time (EA)* at the target; (2) compute earliest arrival times for all departure times of a day (*profile search*). Dijkstra's algorithm [18] can be extended to solve these problems for cost functions with reasonable properties [6,19,38]. However, functional representations of profiles (typically by piecewise-linear functions) are quite complex on realistic instances [13]. Many speedup techniques have been

Partially supported by EU grants 288094 (eCOMPASS) and 609026 (MOVESMART).

adapted to time-dependency. Some use (scalar) lower bounds on the travel time functions to guide the graph search [11,12,37]. TD-CALT [11] yields reasonable EA query times for approximate solutions, allowing dynamic traffic updates, but no profile search. TD-SHARC [8] offers profile search on a country-scale network. Time-dependent Contraction Hierarchies (TCH) [2] enable fast EA and profile searches on continental networks. During preprocessing, TCH computes overlays by iteratively inserting shortcuts [25] obtained from profile searches. Piecewise-linear function approximation [29] is used to reduce shortcut complexity, dropping optimality. A multi-phase extension (ATCH) restores exact results [2]. Time-dependent shortest path oracles described in [33–35] approximate distances in sublinear query time after subquadratic preprocessing. In practical experiments, however, preprocessing effort is still substantial [31,32].

TCH has been generalized to combined optimization of functional travel time and scalar, other costs [3], which poses an NP-hard problem. While this hardness result would of course impact any approach, interestingly, the experiments in [3] suggest that TCH on its own is not particularly robust against user preferences: In a scenario that amounts to the avoidance of highways, preprocessing effort doubles and query performance decreases by an order of magnitude. (Our experiments will confirm this on a non NP-hard formulation of highway avoidance.)

Other works focus on unforeseen dynamic changes (e. g., congestion due to an accident), often by enabling partial updates of preprocessed data [12,20]. Customizable Route Planning (CRP) [9] offloads most preprocessing effort to a metric-independent, separator-based phase. Preprocessed data is then *customized* to a given routing metric for the whole network within seconds or below. This also enables robust integration of user preferences. Customizable Contraction Hierarchies (CCH) [16] follows a similar approach. However, CRP and CCH handle only scalar metrics. To the best of our knowledge, non-scalar metrics for separator-based approaches have only been investigated in the context of electric vehicles (EVCRP) [5], where energy consumption depends on state-of-charge, but functional complexity is very low. On the other hand, the use of scalar approaches for handling live traffic information yields inaccurate results for medium and long distances: Such methods wrongly consider current traffic even at far away destinations—although it will have dispersed once reaching the destination. For realistic results, a combination of dynamic and time-dependent (non-scalar, functional) route planning accounts for current traffic, short-term predictions, and historic knowledge about recurring traffic patterns.

Our Contribution. We carefully extend CRP [9] to time-dependent functions. As such, we are the first to evaluate partition-based overlays on a challenging non-scalar metric. To this end, we integrate profile search into CRP's customization phase and compute time-dependent overlays. Unlike EVCRP and TCH, a naïve implementation fails: Shortcuts on higher-level overlays are too expensive to be kept in memory (and too expensive to evaluate during queries). To reduce functional complexity, we approximate overlay arcs. In fact, approximation subject to a very small error suffices to make our approach practical, in accordance to theory [23]. The resulting algorithmic framework enables interactive queries with low average and maximum error in a very realistic scenario consisting of live traffic, short-term traffic predictions, and historic traffic patterns. Moreover, it supports user preferences such as lower maximum driving speeds or the avoidance of highways. In an extensive experimental setup, we demonstrate that our approach enables integration of custom updates much faster than previous approaches, while allowing fast queries that enable interactive applications. It is also robust to changes in the metric that turn out to be much harder for previous techniques.

2 Preliminaries

A road network is modeled as a directed *graph* $G = (V, A)$ with $n = |V|$ *vertices* and $m = |A|$ *arcs*, where vertices $v \in V$ correspond to intersections and arcs $(u, v) \in A$ to road segments. An *s–t-path* P (in G) is a sequence $P_{s,t} = [v_1 = s, v_2, \dots, v_k = t]$ of vertices such that $(v_i, v_{i+1}) \in A$. If s and t coincide, we call P a *cycle*. Every arc a has assigned a *periodic travel-time function* $f_a \colon \Pi \to \mathbb{R}^+$, mapping departure time within period $\Pi = [0, \pi]$ to travel time. Given a departure time τ at s, the (time-dependent) travel time $\tau_{[s,\dots,t]}$ of an s–t-path is obtained by consecutive function evaluation, i. e., $\tau_{[s,\dots,v_i]} = f_{(v_{i-1},v_i)}(\tau_{[s,\dots,v_{i-1}]})$. We assume that functions are piecewise linear and represented by *breakpoints*. We denote by $|f|$ the number of breakpoints of a function f. Moreover, we define $f^{\max}$ as the maximum value of f, i.e., $f^{\max} = \max_{\tau \in \Pi} f(\tau)$. Analogously, $f^{\min}$ is the minimum value of f. A function f is *constant* if $f \equiv c$ for some $c \in \Pi$. We presume that functions fulfill the *FIFO property*, i. e., for arbitrary $\sigma \leq \tau \in \Pi$, the condition $\sigma + f(\sigma) \leq \tau + f(\tau)$ holds (waiting at a vertex never pays off). Unless waiting is allowed at vertices, the shortest-path problem becomes $\mathcal{NP}$-hard if this condition is not satisfied for all arcs [7,42]. Given two functions f, g, the *link* operation is defined as $\operatorname{link}(f, g) := f + g \circ (\operatorname{id} + f)$, where id is the identity function and $\circ$ is function composition. The result $\operatorname{link}(f, g)$ is piecewise linear again, with at most $|f| + |g|$ breakpoints (namely, at departure times of breakpoints of f and backward projections of departure times of points of g). We also define *merging* of f and g by $\operatorname{merge}(f, g) := \min(f, g)$. The result of merging piecewise linear functions is piecewise linear, and the number of breakpoints is in $\mathcal{O}(|f| + |g|)$ (containing breakpoints of the two original functions and at most one intersection per linear segment). Linking and merging are implemented by coordinated linear sweeps over the breakpoints of the corresponding functions.

The *(travel-time) profile* of a path $P = [v_1, \dots, v_k]$ is the function $f_P \colon \Pi \to \mathbb{R}^+$ that maps departure time τ at v_1 to travel time on P. Starting at $f_{[v_1,v2]} = f_{(v_1,v_2)}$, we obtain the desired profile by consecutively applying the link operation, i. e., $f_{[v_1,\dots,v_i]} = \operatorname{link}(f_{[v_1,\dots,v_{i-1}]}, f_{(v_{i-1},v_i)})$. Given a set $\mathcal{P}$ of s–t-paths, the corresponding *s–t-profile* is $f_{\mathcal{P}}(\tau) = \min_{P \in \mathcal{P}} f_P(\tau)$ for $\tau \in \Pi$, i. e., the *minimum profile* over all paths in $\mathcal{P}$. The s–t-profile maps departure time to minimum travel time for the given paths. It is obtained by (iteratively) merging the respective paths.

A *partition* of V is a set $\mathcal{C} = \{C_1, \dots, C_k\}$ of disjoint vertex sets such that $\bigcup_{i=1}^{k} C_i = V$. More generally, a *nested multi-level partition* consists of sets $\{\mathcal{C}^1, \ldots, \mathcal{C}^L\}$ such that $\mathcal{C}^\ell$ is a partition of V for all $\ell \in \{1, \ldots, L\}$, and additionally for each cell C_i in $\mathcal{C}^\ell, \ell < L$, there is a partition $\mathcal{C}^{\ell+1}$ at level $\ell + 1$ containing a cell C_j with $C_i \subseteq C_j$. We call C_j the *supercell* of C_i. For consistency, we define $\mathcal{C}^0 = \{\{v\} \mid v \in V\}$ and $\mathcal{C}^{L+1} = \{V\}$. Vertices u and v are *boundary vertices* on level ℓ if they are in different cells of $\mathcal{C}^\ell$. Accordingly, the arc $(u, v) \in A$ is a *boundary arc* on level ℓ.

Query Variants and Algorithms. Given a departure time τ and vertices s and t, an *earliest-arrival (EA)* query asks for the minimum travel time from s to t when departing at time τ. Similarly, a *latest-departure (LD)* query asks for the minimum travel time of an s–t-path arriving at time τ. A *profile query* for given source s and target t asks for the minimum travel time at every possible departure time τ, i. e., a profile $f_{s,t}$ from s to t (over all s–t-paths in G). EA queries can be handled by a time-dependent variant of Dijkstra's algorithm [19], which we refer to as *TD-Dijkstra*. It maintains (scalar) *arrival time labels* $d(\cdot)$ for each vertex, initially set to τ for the source s (∞ for all other vertices). In each step, a vertex u with minimum $d(u)$ is extracted from a priority queue (initialized with s). Then, the algorithm *relaxes* all outgoing arcs (u, v): if $d(u) + f_{(u,v)}(d(u))$ improves $d(v)$, it updates $d(v)$ accordingly and adds v to the priority queue (unless it is already contained). LD queries are handled analogously by running the algorithm from t, relaxing incoming instead of outgoing arcs, and maintaining departure time labels.

Profile queries can be solved by *Profile-Dijkstra* [13], which is based on linking and merging. It generalizes Dijkstra's algorithm, maintaining s–v profiles f_v at each vertex $v \in V$. Initially, it sets $f_s \equiv 0$, and $f_v \equiv \infty$ for all other vertices. The algorithm continues along the lines of TD-Dijkstra, using a priority queue with scalar keys $f_v^{\min}$. For extracted vertices u, arc relaxations propagate profiles rather than travel times, computing $g := \text{link}(f_u, f_{(u,v)})$ and $f_v := \text{merge}(f_v, g)$ for outgoing arcs (u, v). As shown by Foschini et al. [23], the number of breakpoints of the profile of an s–v-paths can be superpolynomial, and hence, so is space consumption *per vertex label* and the running time of Profile-Dijkstra in the worst case. Accordingly, it is not feasible for large-scale instances, even in practice [13].

3 Our Approach

We propose *Time-Dependent CRP (TDCRP)*, a speedup technique for time-dependent route planning allowing fast integration of user-dependent metric changes. Additionally, we enable current and/or predicted traffic updates with limited departure time horizon (accounting for the fact that underlying traffic situations resolve over time). To take historic knowledge of traffic patterns into account, we use functions of departure time at arcs. This conceptual change has important consequences: For plain CRP, the topology data structures is fixed after preprocessing, enabling several micro-optimizations with significant impact on customization and query [9]. In our case, functional complexity is metric-dependent (influenced by, e. g., user preferences) and has to be handled dynamically during customization. Hence, for adaptation to dynamic time-dependent scenarios, we require new data structures and algorithmic changes during customization. Below, we recap the three-phase workflow of CRP [9] that allows fast integration of user-dependent routing preferences, describing its extension to TDCRP along the way. In particular, we incorporate *profile queries* into the customization phase to obtain *time-dependent* shortcuts. Moreover, we adapt the query phase to efficiently compute time-dependent shortest routes.

3.1 Preprocessing

The (metric-independent) preprocessing step of CRP computes a multi-level partition of the vertices, with given number L of levels. Several graph partition algorithms tailored to road networks exist, providing partitions with balanced cell sizes and small cuts [10,27,40,41]. For each level $\ell \in \{1, \ldots, L\}$, the respective partition $\mathcal{C}^\ell$ induces an *overlay graph* H^ℓ, containing all boundary vertices and boundary arcs in $\mathcal{C}^\ell$ and *shortcut* arcs between boundary vertices within each cell $C_i^\ell \in \mathcal{C}^\ell$. We define $\mathcal{C}^0 = \{\{v\} \mid v \in V\}$ and $H^0 := G$ for consistency. Building the overlay, we use the clique matrix representation, storing cliques of boundary vertices in matrices of contiguous memory [9]. Matrix entries represent *pointers* to functions (whose complexity is not known until customization). This dynamic data structure rules out some optimizations for plain CRP, such as microcode instructions, that require preallocated ranges of memory for the metric [9]. To improve locality, all functions are stored in a single array, such that profiles corresponding to outgoing arcs of a boundary vertex are in contiguous memory.

3.2 Customization

In the customization phase, costs of all shortcuts (added to the overlay graphs during preprocessing) are computed. We run profile searches to obtain these time-dependent costs. In particular, we require, for each boundary vertex u (in some cell C_i at level $\ell \geq 1$), the time-dependent distances for all $\tau \in \Pi$ to all boundary vertices $v \in C_i$. To this end, we run a profile query on the overlay $H^{\ell-1}$. By design, this query is restricted to *subcells* of C_i, i. e., cells C_j on level $\ell - 1$ for which $C_j \subseteq C_i$ holds. This yields profiles for all outgoing (shortcut) arcs (u, v) in C_i from u. On higher levels, previously computed overlays are used for faster computation of shortcuts. Unfortunately, profile queries are expensive in terms of both running time and space consumption. Below, we describe improvements to remedy these effects, mostly by tuning the profile searches.

Improvements. The main bottleneck of profile search is performing link and merge operations, which require linear time in the function size (cf. Sect. 2). To avoid unnecessary operations, we explicitly compute and store the minimum $f^{\min}$ and the maximum $f^{\max}$ of a profile f in its corresponding label and in shortcuts of overlays. These values are used for early pruning, avoiding costly link and merge operations: Before relaxing an arc (u,v), we check whether $f_u^{\min} + f_{(u,v)}^{\min} > f_v^{\max}$, i. e., the minimum of the linked profile exceeds the maximum of the label at v. If this is the case, the arc (u,v) does not need to be relaxed. Otherwise, the functions are linked. We distinguish four cases, depending on whether the first or second function are constant, respectively. If both are constant, linking becomes trivial (summing up two integers). If one of them is constant, simple shift operations suffice (we need to distinguish two cases, depending on which of the two functions is constant). Only if no function is constant, we apply the link operation.

After linking $f_{(u,v)}$ to f_u, we obtain a tentative label $\tilde{f}_v$ together with its minimum $\tilde{f}_v^{\min}$ and maximum $\tilde{f}_v^{\max}$. Before merging f_v and $\tilde{f}_v$, we run additional checks to avoid unnecessary merge operations. First, we perform bound checks: If $\tilde{f}_v^{\min} > f_v^{\max}$, the function f_v remains unchanged (no merge necessary). Note that this may occur although we checked bounds before linking. Conversely, if $\tilde{f}_v^{\max} < f_v^{\min}$, we simply replace f_v by $\tilde{f}_v$. If the checks fail, and one of the two functions is constant, we must merge. But if f_v and $\tilde{f}_v$ are both nonconstant, one function might still dominate the other. To test this, we do a coordinated linear-time sweep over the breakpoints of each function, evaluating the current line segment at the next breakpoint of the other function. If during this test $\tilde{f}_v(\tau) < f_v(\tau)$ for any point $(\tau, \cdot)$, we must merge. Otherwise we can avoid the merge operation and its numerically unstable line segment intersections.

Additionally, we use *clique flags*: For a vertex v, define its *parents* as all direct predecessors on paths contributing to the profile at the current label of v. For each vertex v of an overlay H^ℓ, we add a flag to its label that is true if *all* parents of v belong to the same cell at level ℓ. This flag is set to true whenever the corresponding label f_v is replaced by the tentative function $\tilde{f}_v$ after relaxing a clique arc (u,v), i. e., the label is set for the first time or the label f_v is dominated by the tentative function $\tilde{f}_v$. It is set to false if the vertex label is partially improved after relaxing a boundary arc. For flagged vertices, we do not relax outgoing clique arcs, as this cannot possibly improve labels within the same cell (due to the triangle inequality and the fact that we use full cliques).

Parallelization. Cells on a given level are processed independently, so customization can be parallelized naturally, assigning cells to different threads [9]. In our scenario, however, workload is strongly correlated with the number of time-dependent arcs in the search graph. It may differ significantly between cells: In realistic data sets, the distribution of time-dependent arcs is clearly not uniform, as it depends on the road type (highways vs. side roads) and the area (rural vs. urban). To balance load, we parallelize per boundary vertex (and not per cell).

Shortcut profiles are written to dynamic containers, as the number of breakpoints is not known in advance. Thus, we must prohibit parallel (writing) access to these data structure. One way to solve this is to make use of locks. However, this is expensive if many threads try to write profiles at the same time. Instead, we use thread-local profile containers, i. e., each thread uses its own container to store profiles. After customization of each level, we synchronize data by copying profiles to the global container sequentially. To improve spatial locality during queries, we maintain the relative order of profiles wrt. the matrix layout (so profiles of adjacent vertices are likely to be contiguous in memory). Since relative order within each thread-local containers is maintained easily (by running queries accordingly), we can use merge sort when writing profiles to the global container.

Approximation. On higher levels of the partition, shortcuts represent larger parts of the graph. Accordingly, they contain more breakpoints and consume more space. This makes profile searches fail on large graphs due to insufficient memory, even on modern hardware. Moreover, running time is strongly correlated to the complexity of profiles. To save space and time, we *simplify* functions during customization. To this end, we use the algorithm of Imai and Iri [29]. For a maximum (relative or absolute) error bound ε, it computes an approximation of a given piecewise linear function with minimum number of breakpoints. In TCH [2], this technique is applied after preprocessing to reduce space consumption. Instead, we use the algorithm to simplify profiles after computing all shortcuts of a certain level. Therefore, searches on higher levels use approximated functions from lower levels, leading to slightly less accurate profiles but faster customization; see Sect. 4. The bound ε is a tuning parameter: Larger values allow faster customization, but decrease quality. Also, approximation is not necessarily applied on all levels, but can be restricted to the higher ones. Note that after approximating shortcuts, the triangle inequality may no longer hold for the corresponding overlay. This is relevant when using clique flags: They yield faster profile searches, but slightly decrease quality (additional arc relaxations may improve shortcut bounds).

3.3 Live Traffic and Short-Term Traffic Predictions

Updates due to, e. g., live traffic, require that we rerun parts of the customization. Clearly, we only have to run customization for *affected* cells, i. e., cells containing arcs for which an update is made. We can do even better if we exploit that live traffic and short-term updates only affect a limited *time horizon*. Thus, we do not propagate updates to boundary vertices that cannot reach an affected arc before the end of its time horizon.

We assume that short-term updates are *partial functions* $f\colon [\pi', \pi''] \to \mathbb{R}^+$, where $\pi' \in \Pi$ and $\pi'' \in \Pi$ are the *beginning* and *end* of the time horizon, respectively. Let $a_1 = (u_1, v_1), \dots, a_k = (u_k, v_k)$ denote the *updated* arcs inside some cell C_i at level ℓ, and let $f_1, \dots, f_k$ be the corresponding partial functions representing time horizons. Moreover, let τ be the current point in time. To update C_i we run, on its induced subgraph, a backward *multi-target* latest departure (LD) query from the tails of all updated arcs. In other words, we initially insert the vertices $u_1, \dots, u_k$ into the priority queue. For each $i \in \{1, \dots, k\}$, the label of u_i is set to π''_i, i. e., the end of the time horizon $[\pi'_i, \pi''_i]$ of the partial function f_i. Consequently, the LD query computes, for each vertex of the cell C_i, the latest possible departure time such that some affected arc is reached before the end of its time horizon. Whenever the search reaches a boundary vertex of the cell, it is marked as *affected* by the update. We stop the search as soon as the departure time label of the current vertex is below τ. (Recall that LD visits vertices in decreasing order of departure time.) Thereby, we ensure that only such boundary vertices are marked from which an updated arc can be reached in time.

Afterwards, we run profile searches for C_i as in regular customization, but only from affected vertices. For profiles obtained during the searches, we test whether they improve the corresponding stored shortcut profile. If so, we add the *affected* interval of the profile for which a change occurs to the set of time horizons of the next level. If shortcuts are approximations, we test whether the change is *significant*, i. e., the maximum difference between the profiles exceeds some bound. We continue the update process on the next level accordingly.

3.4 Queries

The query algorithm makes use of shortcuts computed during customization to reduce the search space. Given a source s and a target t, the search graph consists of the overlay graph induced by the top-level partition $\mathcal{C}^L$, all overlays of cells of lower levels containing s or t, and the level-0 cells in the input graph G that contain s or t. Note that the search graph does not have to be constructed explicitly, but can be obtained on-the-fly [9]: At each vertex v, one computes the highest levels $\ell_{s,v}$ and $\ell_{v,t}$ of the partition such that v is not in the same cell of the partition as s or t, respectively (or 0, if v is in the same level-1 cell as s or t). Then, one relaxes outgoing arcs of v only at level $\min\{\ell_{s,v}, \ell_{v,t}\}$ (recall that $H^0 = G$).

To answer EA queries, we run TD-Dijkstra on this search graph. For faster queries, we make use of the minimum values $f^{\min}_{(u,v)}$ stored at arcs: We do not relax an arc (u, v) if $d(u) + f^{\min}_{(u,v)}$ does not improve $d(v)$. Thereby, we avoid costly function evaluation. Note that we do not use clique flags for EA queries, since we have observed rare but high maximum errors in our implementation when combined with approximated clique profiles.

To answer profile queries, Profile-Dijkstra can be run on the CRP search graph, using the same optimizations as described in Sect. 3.2.

4 Experiments

We implemented all algorithms in C++ using g++ 4.8 (flag -O3) as compiler. Experiments were conducted on a dual 8-core Intel Xeon E5-2670 clocked at 2.6 GHz, with 64 GiB of DDR3-1600 RAM, 20 MiB of L3 and 256 KiB of L2 cache. We ran customization in parallel (using all 16 threads) and queries sequentially.

Input Data and Methodology. Our main test instance is the road network of Western Europe ($|V| = 18$ million, $|A| = 42.2$ million), kindly provided by PTV AG. For this well-established benchmark instance [1], travel time functions were generated synthetically [37]. We also evaluate the subnetwork of

Table 1. Customization performance on Europe for varying approximation parameters (ε). We report, per level, the number of breakpoints (bps, in millions) in the resulting overlay, the percentage of clique arcs that are time-dependent (td.clq.arcs), average complexity of time-dependent arcs (td.arc.cplx), as well as customization time. Without approximation, Levels 5 and 6 cannot be computed as they do not fit into main memory.

ε		Lvl1	Lvl2	Lvl3	Lvl4	Lvl5	Lvl6	Total
—	bps [10^6]	99.1	398.4	816.4	1 363.4	—	—	2 677.4
	td.clq.arcs [%]	17.0	52.6	76.0	84.2	—	—	—
	td.arc.cplx	21.0	68.9	189.0	509.3	—	—	—
	time [s]	11.4	52.0	152.9	206.2	—	—	375.7
0.01 %	bps [10^6]	75.7	182.7	244.6	240.8	149.3	59.2	952.2
	td.clq.arcs [%]	17.0	52.6	76.0	84.2	85.2	82.5	—
	td.arc.cplx	16.0	31.6	56.6	90.0	108.6	108.0	—
	time [s]	4.5	18.0	32.7	82.1	150.3	151.5	439.1
0.1 %	bps [10^6]	60.7	107.5	111.5	87.9	47.9	17.6	432.9
	td.clq.arcs [%]	17.0	52.7	76.0	84.2	85.2	82.5	—
	td.arc.cplx	12.9	18.6	25.8	32.8	34.8	32.1	—
	time [s]	4.2	16.0	21.4	40.7	62.4	55.0	199.7
1.0 %	bps [10^6]	45.7	58.0	45.6	29.2	14.7	5.4	198.5
	td.clq.arcs [%]	17.0	52.7	76.0	84.2	85.2	82.5	—
	td.arc.cplx	9.7	10.0	10.6	10.9	10.7	9.8	—
	time [s]	4.1	14.1	14.8	22.7	29.6	24.1	109.2

Germany ($|V| = 4.7$ million, $|A| = 10.8$ million), where time-dependent data from historical traffic is available (we extract the 24 h profile of a Tuesday).[1] For partitioning, we use PUNCH [10], which is explicitly developed for road networks and aims at minimizing the number of boundary arcs. For Europe, we consider a 6-level partition, with maximum cell sizes $2^{[4:8:11:14:17:20]}$. For Germany, we use a 5-level partition, with cell sizes of $2^{[4:8:12:15:18]}$. Compared to plain CRP, we use partitions with more levels, to allow fine-grained approximation. Computing the partition took 5 min for Germany, and 23 min for Europe. Given that road topology changes rarely, this is sufficiently fast in practice.

Evaluating Customization. Table 1 details customization for different approximation parameters ε on the Europe instance. We report, for several choices of ε and for each level of the partition, figures on the complexity of shortcuts in the overlays and the parallelized customization time. The first block shows figures for exact profile computation. Customization had to be aborted after the fourth level, because the 64 GiB of main memory were not sufficient to store

[1] The Germany and Europe instances can be obtained easily for scientific purposes, see http://i11www.iti.uni-karlsruhe.de/resources/roadgraphs.php.

Table 2. Query performance on Europe as a trade-off between customization effort and approximation. For customization, we set different approximation parameters (ε) and disable (○) or enable (●) clique flags (Cl.). For the different settings, we report query performance in terms of number of vertices extracted from the queue, scanned arcs, evaluated function breakpoints (# Bps), running time, and average and maximum error, each averaged over 100 000 random queries. As we employ approximation per level, resulting query errors can be higher than the input parameter.

Customization			Query					
Approx. ε	Cl.	Time [s]	# Vertices	# Arcs	# Bps	Time [ms]	Err. [%]	
							avg.	max.
0.01 %	○	1 155.1	3 499	541 091	433 698	14.69	<0.01	0.03
0.01 %	●	439.1	3 499	541 090	434 704	14.53	<0.01	0.03
0.10 %	○	533.0	3 499	541 088	96 206	7.63	0.04	0.28
0.10 %	●	199.7	3 499	541 088	99 345	6.47	0.04	0.29
1.00 %	○	284.4	3 499	541 080	67 084	5.66	0.51	3.15
1.00 %	●	109.2	3 499	541 058	70 202	5.75	0.54	3.21

the profiles of all vertex labels. For remaining levels, we clearly see the strong increase in the total number of breakpoints per level. Also, the relative amount of time-dependent arcs rises with each level, since shortcuts become longer. Customization time clearly correlates with profile complexity, from 10 s on the lowest level, to more then three minutes on the fourth. When approximating, we see that customization becomes faster for larger values of ε. We apply approximation to all levels of the partition (using it only on the topmost levels did not provide significant benefits in preliminary experiments). Recall that higher levels work on approximated shortcuts of previous levels, so ε does not provide a bound on the error of the shortcuts. We see that even a very small value (0.01 %) yields a massive drop of profile complexity (more than a factor 5 at Level 4), and immediately allows full customization. For reasonably small values ($\varepsilon = 0.1\,\%, \varepsilon = 1.0\,\%$), we see that customization becomes much faster (less than two minutes for $\varepsilon = 1.0\,\%$). In particular, this is fast enough for traffic updates. Even for larger values of ε, the higher levels are far more expensive: This is due to the increasing amount of time-dependent arcs, slowing down profile search.

Evaluating Customization and Queries. In Table 2, we show query performance for different values of the approximation parameter ε on the Europe instance. We also show the effect of using clique flags during customization: they improve customization performance by about a factor of 2.6, while having a negligible influence on query results. For each value of ε, we report timings as well as average and maximum error for 100 000 point-to-point queries. For each query, the source and target vertex and the departure time were picked uniformly at random. Similar to customization, the data shows that query times decrease with higher approximation ratio. Again, this is due to the smaller number of break-

Table 3. Robustness comparison for TCH [2] and TDCRP. For different input instances, we report timing of metric-dependent preprocessing (always run on 16 cores) and sequential queries. Query times are averaged over the same 100 000 random queries as in Table 2.

Network	TCH		TDCRP	
	Prepro. [s]	Query [ms]	Custom. [s]	Query [ms]
Europe	1 479	1.37	109	5.75
Europe, bad traffic	7 772	5.87	208	8.01
Europe, avoid highways	8 956	19.54	127	8.29

points in profiles (observe that the number of visited vertices and arcs is almost identical in all cases). As expected, both average and maximum error clearly correlate with (but are larger than) ε. There are two reasons for this: As shown in [24,32,35], query errors not only depend on ε but also on the maximum slope of any approximated function. Moreover, since we apply approximation per level, the error bound in [24] applies recursively, leading to a higher theoretical bound. Still, we observe that even for the parameter choice $\varepsilon = 1.0\,\%$, the maximum error is very low (about $3\,\%$). Moreover, query times are quite practical for all values of ε, ranging from 5 ms to 15 ms. In summary, our approach allows query times that are fast enough for interactive applications, if a reasonable, small error is allowed. Given that input functions are based on statistical input with inherent inaccuracy, the error of TDCRP is more than acceptable for realistic applications.

Evaluating Robustness. We also evaluate robustness of our approach against dynamic updates and user-dependent custom metrics. The first scenario (bad traffic) simulates a highly congested graph: for every time-dependent arc in the Western Europe instance with associated travel-time function f, we replace f by f' defined as $f'(\tau) := 2(f(\tau) - f^{\min}(\tau)) + f^{\min}(\tau)$, while maintaining the FIFO property on f'. In the second scenario, we consider user restrictions (avoid highways). For each scenario, customization and the same set of 100 000 random queries as before are run on the respective modified instance. (Hence, we do not remove highways for the second scenario, setting very high costs instead.) Table 3 compares results of the original instance (Europe) to the modified ones.

Besides our approach, which is run using parameter $\varepsilon = 1.0$ for customization, we also evaluate TCH [2], the fastest known approach for time-dependent route planning. All measurements for TCH are based on this freely available implementation: https://github.com/GVeitBatz/KaTCH. While TCH allows faster queries on the original instance, we see that running times increase significantly for the modified ones. Preprocessing time also increases to several hours in both cases. In the first scenario (bad traffic), this can be explained by a larger number of paths that are relevant at different points in time (more congested roads need to be bypassed). Consequently, customization time of TDCRP rises as well but by a much smaller factor. In the second scenario (avoid highways), the TCH hierarchy clearly deteriorates. While TDCRP is quite robust to this change (both customization and query times increase by less than 50 %), TCH queries slow down by more than an order of magnitude.

While possibly subject to implementation, our experiment indicates that underlying vertex orderings of TCH are not robust against less well-behaved metrics. Similar effects can be shown for scalar Contraction Hierarchies (CH) on metrics reflecting, e. g., travel distance [9,25]. In summary, TDCRP is much more robust in both scenarios.

Comparison with Related Work. Finally, Table 4 provides an overview comparing our results to the most relevant existing approaches for time-dependent route planning. For the related work, we show measurements in the fastest reported variant (e. g., if parallelized) but we scale all timings to our hardware as detailed in Table 5 using a benchmark tool [1] available at http://tpajor.com/projects/.

For TCH and ATCH [2], preprocessing can be further split into *node order computation* and *contraction*. Since it has been shown in [2] that node orders can be re-used for certain other metrics (e. g., other week days), we report running times of the contraction as rudimentary customization times. Recall, however, that our robustness tests in Table 3 suggest that there is a limit to the applicability of such a customization approach based on current TCH orders.

We evaluated our approach on both benchmark instances (Germany, and Europe) for the two fastest variants ($\varepsilon = 0.1$ and $\varepsilon = 1.0$) and we see that it competes very well with the previous techniques: While providing query times similar to the fastest existing approaches, TDCRP has by far the lowest metric-dependent preprocessing time (i. e., customization time) and a good parallel speedup (factor 13.9 to 14.2 on Europe for 16 threads). At the same time, resulting average and maximum errors (due to approximating profiles during customization) are similar to previous results and low enough for practical purposes. When parallelized, customization of the whole network is fast enough for regular live-traffic updates: 8 to 16 s on Germany, and 2 to 3 min on Europe. Note, however, that other approaches are also able to handle live traffic by providing *partial updates* of the preprocessed data: For example, by exploiting the fact that effects of live traffic are locally and temporally limited, FLAT [32]) and TDCALT [11] achieve partial update times in well below a minute (for 1,000 traffic-affected arcs).

Interestingly, TDCALT's preprocessing is also quite fast. This could make it an interesting alternative candidate for our scenario (metric customization); since it is mostly based on lower bounds and only light contraction, it might also be fairly robust to sensible, user-defined metrics (unlike TCH, cf. Table 3). Note, however, that TDCALT on Europe requires a significantly higher approximation to achieve a similar level of query performance (even scaled), yielding a high maximum error. Furthermore, in the evaluated variant, landmarks are chosen after the graph contraction routine, making it hard to parallelize the preprocessing (which also has not been attempted). Additionally, TDCALT allows no practical profile search on large instances [8,11], making it a less versatile approach.

Table 4. Comparison of time-dependent speedup techniques on instances of Germany, and Europe. We present figures for variants of TDCALT [11], SHARC [8], TCH and ATCH [2], FLAT [32], and TDCRP. For better comparability across different hardware, we scale all sequential (Seq.) and parallel (Par.) timings to our machine; see Table 5 for factors. For preprocessing, customization, and live traffic updates, we show the number of threads used (Thr.). For EA queries, we present average numbers on queue extractions (# Vert.), scanned arcs, sequential running time in milliseconds, and average and maximum relative error.

Algorithm	Inst.	Thr.	Preprocessing		Customization			Traffic	EA Queries				
			Par.	Space	Par.	Seq.	Space	Par.	# Vert.	# Arcs	Seq.	Err. [%]	
			[h:m:s]	[B/n]	[m:s]	[m:s]	[B/n]	[m:s]			[ms]	avg.	max.
TDCALT	Germany	1	3:14	50	—	—	—	n/a	3 190	12 255	1.93	—	—
TDCALT-K1.15	Germany	1	3:14	50	—	—	—	n/a	1 593	5 339	0.67	0.05	13.84
eco L-SHARC	Germany	1	28:03	219	—	—	—	—	2 776	19 005	2.27	—	—
heu SHARC	Germany	1	1:14:06	137	—	—	—	—	818	1 611	0.25	n/a	0.61
ATCH (1.0)	Germany	8	5:50	239	1:09	6:59	239	—	588	7 993	1.15	—	—
inex. TCH (0.1)	Germany	8	5:50	286	1:09	6:59	286	—	642	7 138	0.65	0.02	0.10
inex. TCH (2.5)	Germany	8	5:50	172	1:09	6:59	172	—	668	7 429	0.67	0.79	2.44
FLAT/FCA	Germany	6	>1 day	>10 000	—	—	—	0:44	1 122	n/a	1.51	n/a	1.53
TDCRP (0.1)	Germany	16	4:33	29	0:16	3:30	166	0:16	2 152	167 263	1.92	0.05	0.25
TDCRP (1.0)	Germany	16	4:33	29	0:08	1:43	77	0:08	2 152	167 305	1.66	0.68	2.85
TDCALT	Europe	1	21:35	61	—	—	—	0:22	60 961	356 527	43.67	—	—
TDCALT-K1.05	Europe	1	21:35	61	—	—	—	0:22	32 405	n/a	22.48	0.01	3.94
TDCALT-K1.15	Europe	1	21:35	61	—	—	—	0:22	6 365	32 719	3.31	0.26	8.69
eco L-SHARC	Europe	1	2:27:07	198	—	—	—	—	18 289	165 382	13.77	—	—
heu SHARC	Europe	1	7:59:08	127	—	—	—	—	5 031	8 411	1.06	n/a	1.60
ATCH (1.0)	Europe	8	42:21	208	7:26	48:07	208	—	1 223	20 336	2.68	—	—
inex. TCH (0.1)	Europe	8	42:21	239	7:26	48:07	239	—	1 722	24 389	2.50	0.02	0.15
inex. TCH (2.5)	Europe	8	42:21	175	7:26	48:07	175	—	1 875	26 948	2.72	0.48	3.37
TDCRP (0.1)	Europe	16	22:33	32	3:20	47:10	237	3:20	3 499	541 088	6.47	0.04	0.29
TDCRP (1.0)	Europe	16	22:33	32	1:49	25:16	133	1:49	3 499	541 058	5.75	0.54	3.21

Table 5. Scaling factors for different machines, used in Table 4. Scores were determined by a shared Dijkstra implementation [1] on the same graph. These factors have to be taken with a grain of salt, since Dijkstra's algorithm is not a good indicator of cache performance. When scaling on TDCRP performance, instead, we observe a factor of 2.06–2.18 for the Opteron 2218 (which we have access to), depending on the instance.

Machine	Used by	Score [ms]	Factor
2× 8-core Intel Xeon E5-2670, 2.6 GHz	TDCRP	36 582	—
AMD Opteron 2218, 2.6 GHz	TDCALT [11], SHARC [8]	101 552	2.78
2× 4-core Intel Xeon X5550, 2.66 GHz	TCH, ATCH [2]	39 684	1.08
6-core Intel Xeon E5-2643v3, 3.4 Ghz	FLAT/FCA [32]	30 901	0.84

To summarize, we see that TDCRP clearly broadens the state-of-the-art of time-dependent route planning, handling a wider range of practical requirements (e. g., fast metric-dependent preprocessing, robustness to user preferences, live traffic) with a query performance close to the fastest known approaches.

5 Conclusion

In this work, we introduced TDCRP, a separator-based overlay approach for dynamic, time-dependent route planning. We showed that, unlike its closest competitor (A)TCH, it is robust against user-dependent metric changes, very much like CRP is more robust than CH. Most importantly, unlike scalar CRP, we have to deal with time-dependent shortcuts, and a strong increase in functional complexity on higher levels; To reduce memory consumption, we approximate the overlay arcs at each level, accelerating customization and query times. As a result, we obtain an approach that enables fast near-optimal, time-dependent queries, with quick integration of user preferences, live traffic, and traffic predictions.

There are several aspects of future work. Naturally, we are interested in alternative customization approaches that avoid label-correcting profile searches. This could be achieved, e. g., by using kinetic data structures [23], or balanced contraction [2] within cells. It would be interesting to re-evaluate (A)TCH in light of Customizable CH [16,17]. Also, while we customized time-dependent overlay arcs with both historic travel time functions (changes seldom) and user preferences (changes often) at once, in practice, it might pay off to separate this into two further phases (yielding a 4-phase approach). Furthermore, one could aim at exact queries based on approximated shortcuts as in ATCH.

While our approach is customizable, it requires arc cost functions that map time to time. This allows to model avoidance of highways or driving slower than the speed limit, but it cannot handle combined linear optimization of (time-dependent) travel time and, e. g., toll costs. For that, one should investigate the application of generalized time-dependent objective functions as proposed in [3].

Finally, functional complexity growth of time-dependent shortcuts is problematic, and from what we have seen, it is much stronger than the increase in the number of corresponding paths. It seems wasteful to apply the heavy machinery of linking and merging during preprocessing, when time-dependent evaluation of just a few paths (more than one is generally needed) would give the same results. This might explain why TDCALT, which is mostly based just on scalar lower bounds, is surprisingly competitive. So re-evaluation seems fruitful, possibly exploiting insights from [20]. Revisiting hierarchical preprocessing techniques that are not based on shortcuts [26,36] could also be interesting.

Acknowledgements. We thank Gernot Veit Batz, Daniel Delling, Moritz Kobitzsch, Felix König, Spyros Kontogiannis, and Ben Strasser for interesting conversations.

References

1. Bast, H., Delling, D., Goldberg, A.V., Müller-Hannemann, M., Pajor, T., Sanders, P., Wagner, D., Werneck, R.F.: Route Planning in Transportation Networks. CoRR abs/1504.05140 (2015)
2. Batz, G.V., Geisberger, R., Sanders, P., Vetter, C.: Minimum time-dependent travel times with contraction hierarchies. ACM J. Exp. Algorithmics **18**(1.4), 1–43 (2013)
3. Batz, G.V., Sanders, P.: Time-dependent route planning with generalized objective functions. In: Epstein, L., Ferragina, P. (eds.) ESA 2012. LNCS, vol. 7501, pp. 169–180. Springer, Heidelberg (2012)
4. Bauer, R., Columbus, T., Rutter, I., Wagner, D.: Search-space size in contraction hierarchies. In: Fomin, F.V., Freivalds, R., Kwiatkowska, M., Peleg, D. (eds.) ICALP 2013, Part I. LNCS, vol. 7965, pp. 93–104. Springer, Heidelberg (2013)
5. Baum, M., Dibbelt, J., Pajor, T., Wagner, D.: Energy-optimal routes for electric vehicles. In: SIGSPATIAL 2013, pp. 54–63. ACM Press (2013)
6. Cooke, K., Halsey, E.: The shortest route through a network with time-dependent internodal transit times. J. Math. Anal. Appl. **14**(3), 493–498 (1966)
7. Dean, B.C.: Algorithms for minimum-cost paths in time-dependent networks with waiting policies. Networks **44**(1), 41–46 (2004)
8. Delling, D.: Time-dependent SHARC-routing. Algorithmica **60**(1), 60–94 (2011)
9. Delling, D., Goldberg, A.V., Pajor, T., Werneck, R.F.: Customizable route planning in road networks. Transport. Sci. (2015)
10. Delling, D., Goldberg, A.V., Razenshteyn, I., Werneck, R.F.: Graph partitioning with natural cuts. In: IPDPS 2011, pp. 1135–1146. IEEE Computer Society (2011)
11. Delling, D., Nannicini, G.: Core routing on dynamic time-dependent road networks. Informs J. Comput. **24**(2), 187–201 (2012)
12. Delling, D., Wagner, D.: Landmark-based routing in dynamic graphs. In: Demetrescu, C. (ed.) WEA 2007. LNCS, vol. 4525, pp. 52–65. Springer, Heidelberg (2007)
13. Delling, D., Wagner, D.: Time-dependent route planning. In: Ahuja, R.K., Möhring, R.H., Zaroliagis, C.D. (eds.) Robust and Online Large-Scale Optimization. LNCS, vol. 5868, pp. 207–230. Springer, Heidelberg (2009)
14. Demiryurek, U., Banaei-Kashani, F., Shahabi, C.: A case for time-dependent shortest path computation in spatial networks. In: SIGSPATIAL 2010, pp. 474–477. ACM Press (2010)

15. Diamantopoulos, T., Kehagias, D., König, F., Tzovaras, D.: Investigating the effect of global metrics in travel time forecasting. In: ITSC 2013, pp. 412–417. IEEE (2013)
16. Dibbelt, J., Strasser, B., Wagner, D.: Customizable contraction hierarchies. In: Gudmundsson, J., Katajainen, J. (eds.) SEA 2014. LNCS, vol. 8504, pp. 271–282. Springer, Heidelberg (2014)
17. Dibbelt, J., Strasser, B., Wagner, D.: Customizable contraction hierarchies. J. Exp. Algorithmics. **21**(1), 1.5:1–1.5:49 (2016). doi:10.1145/2886843
18. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numer. Math. **1**(1), 269–271 (1959)
19. Dreyfus, S.E.: An appraisal of some shortest-path algorithms. Oper. Res. **17**(3), 395–412 (1969)
20. Efentakis, A., Pfoser, D.: Optimizing landmark-based routing and preprocessing. In: IWCTS 2013, pp. 25:25–25:30. ACM Press (2013)
21. Efentakis, A., Pfoser, D., Vassiliou, Y.: SALT. a unified framework for all shortest-path query variants on road networks. In: Bampis, E. (ed.) SEA 2015. LNCS, vol. 9125, pp. 298–311. Springer, Heidelberg (2015)
22. Eppstein, D., Goodrich, M.T.: Studying (non-planar) road networks through an algorithmic lens. In: SIGSPATIAL 2008, pp. 16:1–16:10. ACM Press (2008)
23. Foschini, L., Hershberger, J., Suri, S.: On the complexity of time-dependent shortest paths. Algorithmica **68**(4), 1075–1097 (2014)
24. Geisberger, R., Sanders, P.: Engineering time-dependent many-to-many shortest paths computation. In: ATMOS 2010, pp. 74–87. OASIcs (2010)
25. Geisberger, R., Sanders, P., Schultes, D., Vetter, C.: Exact routing in large road networks using contraction hierarchies. Transp. Sci. **46**(3), 388–404 (2012)
26. Gutman, R.J.: Reach-based routing: a new approach to shortest path algorithms optimized for road networks. In: ALENEX 2004, pp. 100–111. SIAM (2004)
27. Hamann, M., Strasser, B.: Graph bisection with pareto-optimization. In: ALENEX 2016, pp. 90–102. SIAM (2016)
28. Holzer, M., Schulz, F., Wagner, D.: Engineering multilevel overlay graphs for shortest-path queries. ACM J. Exp. Algorithmics **13**(2.5), 1–26 (2008)
29. Imai, H., Iri, M.: An optimal algorithm for approximating a piecewise linear function. J. Inf. Process. **9**(3), 159–162 (1986)
30. Jung, S., Pramanik, S.: An efficient path computation model for hierarchically structured topographical road maps. IEEE Trans. Knowl. Data Eng. **14**(5), 1029–1046 (2002)
31. Kontogiannis, S., Michalopoulos, G., Papastavrou, G., Paraskevopoulos, A., Wagner, D., Zaroliagis, C.: Analysis and experimental evaluation of time-dependent distance oracles. In: ALENEX 2015, pp. 147–158. SIAM (2015)
32. Kontogiannis, S., Michalopoulos, G., Papastavrou, G., Paraskevopoulos, A., Wagner, D., Zaroliagis, C.: Engineering oracles for time-dependent road networks. In: ALENEX 2016, pp. 1–14. SIAM (2016)
33. Kontogiannis, S., Wagner, D., Zaroliagis, C.: Hierarchical Oracles for Time-Dependent Networks. CoRR abs/1502.05222 (2015)
34. Kontogiannis, S., Zaroliagis, C.: Distance oracles for time-dependent networks. In: Esparza, J., Fraigniaud, P., Husfeldt, T., Koutsoupias, E. (eds.) ICALP 2014. LNCS, vol. 8572, pp. 713–725. Springer, Heidelberg (2014)
35. Kontogiannis, S., Zaroliagis, C.: Distance oracles for time-dependent networks. Algorithmica **74**(4), 1404–1434 (2015)
36. Maervoet, J., Causmaecker, P.D., Berghe, G.V.: Fast approximation of reach hierarchies in networks. In: SIGSPATIAL 2014, pp. 441–444. ACM Press (2014)

37. Nannicini, G., Delling, D., Liberti, L., Schultes, D.: Bidirectional A* search on time-dependent road networks. Networks **59**, 240–251 (2012)
38. Orda, A., Rom, R.: Shortest-path and minimum delay algorithms in networks with time-dependent edge-length. J. ACM **37**(3), 607–625 (1990)
39. Pfoser, D., Brakatsoulas, S., Brosch, P., Umlauft, M., Tryfona, N., Tsironis, G.: Dynamic travel time provision for road networks. In: SIGSPATIAL 2008, pp. 68:1–68:4. ACM Press (2008)
40. Sanders, P., Schulz, C.: Distributed evolutionary graph partitioning. In: ALENEX 2012, pp. 16–29. SIAM (2012)
41. Schild, A., Sommer, C.: On balanced separators in road networks. In: Bampis, E. (ed.) SEA 2015. LNCS, vol. 9125, pp. 286–297. Springer, Heidelberg (2015)
42. Sherali, H.D., Ozbay, K., Subramanian, S.: The time-dependent shortest pair of disjoint paths problem: complexity, models, and algorithms. Networks **31**(4), 259–272 (1998)