Time-Dependent Alternative Route Planning

Spyros Kontogiannis
Department of Computer Science & Engineering, University of Ioannina, Greece
Computer Technology Institute & Press "Diophantus", Patras, Greece
kontog@uoi.gr

Andreas Paraskevopoulos
Department of Computer Engineering & Informatics, University of Patras, Greece
paraskevop@ceid.upatras.gr

Christos D. Zaroliagis
Department of Computer Engineering & Informatics, University of Patras, Greece
Computer Technology Institute & Press "Diophantus", Patras, Greece
zaro@ceid.upatras.gr

Abstract

We present a new method for computing a set of alternative origin-to-destination routes in road networks with an underlying time-dependent metric. The resulting set is aggregated in the form of a time-dependent alternative graph and is characterized by minimum route overlap, small stretch factor, small size and low complexity. To our knowledge, this is the first work that deals with the time-dependent setting in the framework of alternative routes. Based on preprocessed minimum travel-time information between a small set of nodes and all other nodes in the graph, our algorithm carries out a collection phase for candidate alternative routes, followed by a pruning phase that cautiously discards uninteresting or low-quality routes from the candidate set. Our experimental evaluation on real time-dependent road networks demonstrates that the new algorithm performs much better (by one or two orders of magnitude) than existing baseline approaches. In particular, the entire alternative graph can be computed in less than 0.384sec for the road network of Germany, and in less than 1.24sec for that of Europe. Our approach provides also "quick-and-dirty" results of decent quality, in about 1/300 of the above mentioned query times for continental-size instances.

2012 ACM Subject Classification Computing methodologies → Combinatorial algorithms

Keywords and phrases time-dependent shortest path, alternative routes, travel-time oracle, plateau and penalty methods

Digital Object Identifier 10.4230/OASIcs.ATMOS.2020.8

Funding This research was supported by the Operational Program Competitiveness, Entrepreneurship and Innovation (call Research–Create–Innovate, co-financed by EU and Greek national funds), under contract no. T1EDK-03616 (project SocialPARK).

1 Introduction

Querying a route planning service is nowadays a common daily-routine activity. The majority of such services, as well as of the underlying route planning algorithms, answer queries by offering a best route from an origin o to a destination d, under a certain optimization criterion (e.g., distance, arrival-time, etc.). Nevertheless, such an answer may not always be desirable or satisfactory, since: (i) humans typically prefer to have choices; (ii) every human has his/her own personal preferences that vary and depend on specialized knowledge or subjective criteria (e.g., like/dislike certain parts of a route), which are not always easy to quantify or estimate; (iii) a traveler may have to occasionally follow a different route than the originally planned due to an emergent traffic condition (accident, road works, etc.). Consequently, a route planning service offering a *set* of good/reasonable alternative routes is more likely to satisfy the traveler's needs; and vice versa, the traveler can use alternative routes as back-up choices, in case of emergent traffic conditions or other unforeseen incidents.

© Spyros Kontogiannis, Andreas Paraskevopoulos, and Christos D. Zaroliagis;
licensed under Creative Commons License CC-BY
20th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2020).
Editors: Dennis Huisman and Christos D. Zaroliagis; Article No. 8; pp. 8:1–8:14

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In all these cases, the essential task is to compute, *efficiently*, reasonable alternatives to an optimal *od*-route. Towards this direction, recent works in the literature investigate the efficient computation of alternative routes in *time-independent* road networks (i.e., networks with scalar edge-costs). There are two prevailing algorithmic approaches for alternative routes in these networks: The first approach, initiated in [1] and further extended in [13, 18], computes a few (e.g., 2 or 3) alternative *od*-routes that pass through specific vertices in the network, called *via-nodes.* The second approach, introduced in [2] and further extended in [21], creates a set of reasonable alternative routes in the form of a subgraph, called the *alternative graph.* Moreover, there are some proprietary algorithms which are used by commercial systems (e.g., Google and TomTom) to suggest alternative routes.

The notion of an Alternative Graph (AG) turned out to be more suitable for high-demanding navigation systems [9, 14], since the approach with via-nodes is restricted on fixed optimization criteria and it may create (higher than required) overlapping among the alternative routes, or may not even be successful in finding a sufficient number of alternatives for certain scenarios. Generic quality characteristics of AG were described in [2], using three optimization criteria: the *totalDistance* criterion, quantifying the *total-overlappingness* of the best subset of routes within AG, the *averageDistance* criterion, quantifying the *stretch* of these routes, and the complexity of the entire AG subgraph, counted as the number of *decision edges* (sum of alternatives per intermediate node visited, other than the out-edge belonging to the optimal remaining path to the destination). As it is shown in [2], all of them together are important in order to produce a high-quality AG.

However, optimizing a simple objective function combining just any two of them is an NP-hard problem [2]. Hence, one has to concentrate on heuristics. Four heuristic approaches were investigated in [2], based on the Plateau [4] and the Penalty [5] methods. Experimental evaluations in [2, 4] demonstrated that a combination of them seems to be the best choice. A new set of heuristics, including improved extensions of both the Plateau and Penalty methods, were proposed in [21]. As a result, computing an AG subgraph of much better quality than the ones in [2] became possible, and this was verified on several static, (i.e., time-independent) road networks of Western Europe.

In this work, we investigate the AG concept on the more realistic setting of *time-dependent* road networks, represented as directed graphs whose edge costs are determined by travel-time *functions.* In such a setting there exist approaches that compute only the best *od*-route, using either heuristic methods (see e.g., [3]), or earliest-arrival-time oracles (see e.g., [15, 16, 17]). The latter case, of an oracle, consists of a (subquadratic in size) carefully designed data structure, created during a preprocessing phase, along with a query algorithm that exploits this data structure in order to respond to arbitrary earliest-arrival-time queries in sublinear time, with a *provably small* approximation guarantee for the quality of the solution.

Our main contribution is a new heuristic algorithm, called *TDAG*, that computes a *time-dependent* AG which succinctly represents alternative routes of guaranteed quality in a time-dependent road network. Based on precomputed minimum-travel-time information between a small set of nodes and all other nodes in the graph, TDAG selects carefully an initial candidate set of *od* routes that subsequently improves in an iterative pruning phase that discards uninteresting or low-quality routes, until the resulting AG meets the quality criteria set. Our experimental evaluation of TDAG on real-world benchmark time-dependent road networks shows that the entire AG can be computed pretty fast, even for continental-size networks, outperforming typical baseline approaches by one to two orders of magnitude. In particular, the entire AG can be computed in less than 0.384sec for the road network of Germany, and in less than 1.24sec for that of Europe. TDAG also provides "quick-and-dirty"

results of decent quality, in about 1/300 of the above mentioned query times. To our knowledge, this is the first work achieving efficient computation of alternative routes in the more realistic setting of time-dependent road networks.

2 Preliminaries

A time-dependent *road network* can be modeled as a *directed graph* $G = (V, E)$, where each node $v \in V$ represents either intersection points along a road, or vehicle departure/arrival events with zero waiting-time; each edge $e \in E$ represents uninterrupted road segments between nodes. Let $|V| = n$, $|E| = m$. Given a time period T, and any edge $e = uv \in E$, if we consider any *departure-time* $t_u \in [0, T)$ from the tail u, then $D[uv](t_u)$ is the corresponding edge-traversal-time for uv, determined by the evaluation of a *continuous*, *piecewise-linear* (pwl) function $D[uv] : [0, T) \mapsto \mathbb{R}_{\geq 0}$. Analogously, $t_v = Arr[uv](t_u) = t_u + D[uv](t_u)$ is the corresponding function providing the *edge-arrival-time* to the head v, for different departure-times from u. We additionally make the (typical for road networks) *strict FIFO property* assumption: each edge-traversal-time function $D[uv]$ has minimum slope greater than -1. Equivalently, we assert that the edge-arrival-time functions $Arr[uv]$ are strictly increasing. This property implies that there is no reason to wait at the tail u of uv before traversing it towards the head v, provided that we are interested in earliest-arrival-times.

Given a departure-time $t \in [0, T)$, and a path $\pi = \langle x_0x_1, x_1x_2, \ldots, x_{k-1}x_k \rangle$ (as a sequence of edges), $Arr[\pi](t) = Arr[x_{k-1}x_k](Arr[x_{k-2}x_{k-1}](\cdots(Arr[x_1x_2](Arr[x_0x_1](t)))\ \cdots))$ is the *path-arrival-time* function, defined by applying function composition on the edge-arrival-time functions of π's constituent edges. In addition, $D[\pi](t) = Arr[\pi](t) - t$ is the corresponding *path-travel-time* function. Let $\mathcal{P}_{u,v}$ be the set of all uv-paths in G, i.e., originating at u and ending at v. Then, $\forall t \in [0, T)$, $Arr[u, v](t) = \min_{\pi \in \mathcal{P}_{u,v}}\{Arr[\pi](t)\}$ is the *earliest-arrival-time* function, from u to v. Analogously, $D[u, v](t) = Arr[u, v](t) - t$ is the corresponding *minimum-travel-time* (or shortest-path-length) function, and $P[u, v](t)$ is the corresponding *time-dependent-shortest-path* function, providing the minimum-travel-time paths w.r.t. the departure time t from u. For $\epsilon > 0$ and $\forall t \in [0, T)$, a function $\overline{D}[u, v](t)$ such that $D[u, v](t) \leq \overline{D}[u, v](t) \leq (1 + \epsilon) \cdot D[u, v](t)$ is called a $(1 + \epsilon)$ *upper-approximation* for $D[u, v]$.

Our main goal is to obtain fundamentally different (but not necessarily disjoint) alternative-paths with optimal or near-optimal travel-times, from an origin-node o to a destination-node d in G, and departure-time t_o from o. The aggregation of the computed alternative od-paths is materialized by the concept of the *Alternative Graph (AG)*, a notion first introduced in [2]. We shall now proceed with the adaptation of the AG concept to the time-dependent context.

Formally, an alternative graph $H = (V', E')$ is the induced subgraph by the edges of several od-paths in G. Let $D_G[u, v](t) \equiv D[u, v](t)$ and $D_H[u, v](t)$ denote the minimum-travel-time functions w.r.t. G and H, respectively. Similarly, $Arr_G[u, v](t) \equiv Arr[u, v](t)$ and $Arr_H[u, v](t)$ denote the earliest-arrival-time functions w.r.t. G and H, respectively. Succinctly representing the produced alternative paths with AG is reasonable, because the alternative paths may share common nodes (including o and d) and edges. Furthermore, their subpaths may be combined to form even more alternative paths, possibly better than the ones that determined AG. In general, there can be too many alternative od-paths and the problem is to find a way to select only a meaningful subset of them. Hence, there is a need for filtering and ranking the alternative od-paths, based on certain quality criteria.

The main idea of the AG approach is to rank the paths w.r.t. some quality criteria and discard the ones that have poor scores. We use the quality indicators proposed in [2] for static instances. These indicators are defined on the single-edge level and then they are extended

to the edge-set level. We provide at this point the definition of these quality criteria, adapted to time-dependent networks. Let $H = (V', E')$ be an AG of G, and let $uv \in E'$. Then:

$$
\begin{aligned}
W[uv](t) &:= D[uv](Arr_H[o,u](t)) \\
share[uv](t) &:= \frac{W[uv](t)}{D_H[o,u](t) + W[uv](t) + D_H[v,d](Arr_H[o,v](t))} \\
totalDistance(t) &:= \sum_{uv \in E'} share[uv](t) \qquad (path\ non\text{-}overlappingness) \\
stretch[uv](t) &:= \frac{W[uv](t)}{D_G[o,d](t) \cdot totalDistance(t)} \\
averageDistance(t) &:= \sum_{uv \in E'} stretch[uv](t) \qquad (path\ stretch) \\
decisionEdges &:= \sum_{v \in V' \setminus \{d\}} (outdegree(v) - 1) \qquad (AG\ size)
\end{aligned}
$$

The criterion *decisionEdges* quantifies the size-complexity of AG, as the number of the alternative paths in AG is directly dependent on the number of the "decision" edge branches in AG. For this reason, the higher the value of *decisionEdges*, the more confusion is created to a typical traveler, when having to choose a route among the alternatives. Therefore, it should be limited. The criterion *totalDistance* captures the extent to which the paths in AG are non-overlapping. Its maximum value is *decisionEdges*+1 and can be as large as the number of all *od*-paths in AG, e.g. when all of them are edge-disjoint. Its minimum value is 1, corresponding to the case where the AG has only one *od*-path. The criterion *averageDistance* measures the average path-travel-time of the alternative paths w.r.t. the shortest one. Its minimum value is 1, e.g., when every *od*-path in AG has the minimum-travel-time.

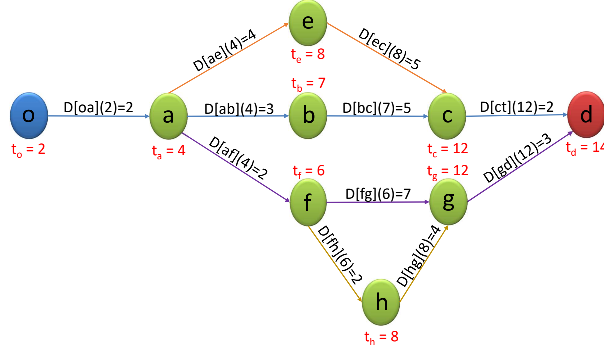

Figure 1 Evaluation of the quality criteria for an alternative graph. For each node x, $t_x = Arr[o,x](2)$ is the earliest arrival-time at x, for departure time $t_o = 2$.

Figure 1 provides an example AG H whose quality indicators are computed as follows, for a given departure-time $t = 2$ from o.

$$
\begin{aligned}
totalDistance(2) &= \frac{(4+5)}{2+(4+5)+2} + \frac{2+3+5+2}{2+3+5+2} \\
&+ \frac{2+7}{2+(2+7)+3)} + \frac{3}{2+2+2+4+3} + \frac{2+4}{2+2+(2+4)+3} \\
&= 0.692 + 1 + 0.643 + 0.231 + 0.462 = 3.028
\end{aligned}
$$

$$
averageDistance(2) = \frac{2+2+2+2+3+3+4+4+5+5+7}{12 \cdot 3.028} = 1.073
$$

$$
decisionEdges = |E'| - (|V'| - 1) = 11 - 8 = 3
$$

In order to construct a high-quality alternative graph, one should aim for high *totalDistance* and low *averageDistance*. In practice, however, achieving low *averageDistance* may take away the ability of collecting high-degree disjoint (non-overlapping) paths and gaining high *totalDistance*, as these criteria can be contradicting with each other. In any case, the target function can be any linear combination of *totalDistance* and *averageDistance*. Similar to [2, 21], we adopt as our target function the quantity $totalDistance + 1 - averageDistance$.

2.1 Computing Time-dependent Shortest Paths

In this section we review some fundamental techniques for computing time-dependent shortest paths, which are used throughout the paper.

Time-dependent Dijkstra. The time-dependent variant of Dijstra's algorithm (TDD) [8] is a straightforward extension of the classical algorithm that computes earliest-arrival-times "on the fly" when scanning (relaxing) the outgoing edges from a node. TDD grows a shortest-path tree rooted at an origin o, for a given departure-time t_o from it. Analogously to the static case, TDD performs a breadth-first search (BFS) exploration of the graph, settling the nodes in increasing order of their tentative labels (representing earliest-arrival-times from o, given the departure-time t_o from it), until the priority queue becomes empty, or a given destination d is settled. During the settling of a node, all the outgoing edges are relaxed, implying new evaluations of the corresponding edge-traversal-time functions. Note that the resulting shortest-path tree may vary for different departure-time choices $t_o \in [0, T)$.

Reversed Time-dependent Dijkstra. The reversed version of TDD (RTDD) grows a full shortest path tree rooted at a node d for a given arrival time t_d. The differences from the original (forward) TDD are the following: (a) the edge relaxations are performed for the incoming edges of each explored node; and (b) the algorithm computes latest-departure-times at edge tails "on the fly" during an edge relaxation, by evaluating the inverse of the edge-arrival-time function (which is strictly increasing, due to the strict FIFO property).

CFLAT. The CFLAT time-dependent oracle [15] precomputes approximate minimum-travel-time functions $\overline{D}$ (travel-time summaries) from each element of a small set of *landmark* nodes, towards all reachable destinations from it. These travel-time summaries are succinctly represented as a collection of time-stamped minimum-travel-time trees. Their careful construction ensures both $(1+\epsilon)$ approximation guarantees (for any $\epsilon > 0$) for the landmark-to-destination travel-time functions $\overline{D}$, and efficient (subquadratic) space requirements.

2.2 Computing Alternative Graphs in Static Road Networks

In this section, we briefly review some approaches used for computing alternative graphs in time-independent (static) graphs.

k-Shortest Paths. The k-shortest path routing algorithm [10, 23] finds k shortest paths in order of increasing cost. The disadvantage of this approach is that the computed alternative paths may share too many edges, making it difficult for a human to actually distinguish them and eventually make his/her own selection of a route. In order for really meaningful alternatives to be revealed, one should compute k-shortest paths for very large values of k, at the expense of a rather prohibitive computational cost.

Pareto. The Pareto algorithm [6, 11, 20] computes an AG by iteratively finding *Pareto-optimal* paths on a suitably defined objective cost vector. The idea is to use as first edge-cost vector the one of the single-criterion problem, while the second edge-cost is defined as follows: all edges belonging to AG (initially the AG is the shortest od-path) set their second cost function to their initial edge cost. All edges not belonging to AG, set their second cost function to zero. This approach also produces too many alternatives with small deviations. Relaxing the domination criteria and fine-tuning the bounds is non-trivial and time consuming.

Plateau. The Plateau algorithm [4] provides alternative od-paths by constructing "plateaus" that connect shortest subpaths. For a shortest-path tree T_f from o and a reverse shortest-path tree T_b from d, a *uv-plateau* is a uv-path that is a shortest subpath both in T_f and T_b. The candidate paths via plateaus are constructed by running Dijkstra's algorithm from o and its reverse version from d, to produce respectively the trees T_f and T_b. Then, for each uv-plateau in T_f and T_b, the shortest ou-path in T_f and the shortest vd-path in T_b are connected at the endpoints of the uv-plateau, in order to form a complete od-path. The candidate od-paths are of high quality, but they are too many, requiring a size decreasing filtration.

Penalty. The Penalty method [5] provides alternative paths by iteratively running shortest-path queries and adjusting the weight of the edges on the resulting path. Initially, a shortest-path query is performed. The resulting shortest path $\pi_{o,d}$ is penalized, by increasing the weight of all its edges. Then, a new od-query is executed in the graph with the new weights. The resulting shortest path $\pi'_{o,d}$ is again penalized and, if it is short and different enough from the previously discovered od-paths, it is added to the solution set, otherwise it is ignored. This process is repeated until a sufficient number of alternative paths (with desired characteristics) is discovered, or the weight adjustments of od-paths bring no better results. For a suitable penalty scheme, the resulting set of od-paths can be of high quality.

3 The TDAG Algorithm

In this section we present our new algorithm, TDAG, which, given a time-dependent road network $G = (V, E)$ with a small set $L \subset V$ of landmark nodes, and an arbitrary query (o, d, t_o) of an origin $o \in V$, a destination node $d \in V$ and a departure-time $t_o \in [0, T)$ from o, computes a collection of meaningful (short and essentially non-overlapping) alternative od-routes. The solution is succinctly represented by an alternative graph H, i.e., the subgraph of G induced by the chosen od-routes. Of course, within H there may exist even better combinations of od-routes for the query (o, d, t_o), which are also considered as part of the solution. The input arguments of TDAG are: (i) the number $N \in \mathcal{O}(1)$ of nearby landmarks that will be settled by TDD in the origin's neighborhood; (ii) the upper bounds $maxAverageDistance$ for the $averageDistance$ criterion, $maxDecisionEdges$ for the $decisionEdges$ criterion, and $maxStretch$ for the maximum stretch of each accepted od-path in H compared to the minimum-travel-time $D_H[od](t_o)$ in the alternative graph H[1]. All these input parameters directly affect the size, the quality and the computation time for constructing H. TDAG consists of two parts (preprocessing and query) that will be presented in the rest of this section.

[1] Since TDAG essentially mimics the preprocessing of the CFLAT oracle [15], one can easily deduce that $D_H[od](t_o)$ is a very good approximation of $D_G[od](t_o)$, for all possible departure times from the origin.

3.1 TDAG Preprocessing

Initially, TDAG chooses a small subset of nodes in G to constitute the landmark set L. There are various ways for the selection of L, either randomly, or according to some properties of the underlying graph (e.g., some balanced partition of the graph, or the ranking of the graph nodes according to a centrality measure such as betweeness-centrality) [15]. In this work we choose one of the most successful methods for landmark selection, called *Sparse-Random* (SR), according to which the landmarks are selected sequentially. Each new landmark ℓ is chosen uniformly at random from the remaining nodes and, after its selection, a small neighborhood of nodes around ℓ is also excluded from future landmark selections. TDAG proceeds with the computation and succinct storage of timestamped shortest-path trees, from each landmark $\ell \in L$ towards all reachable destinations $v \in V$. These trees comprise the travel-time summaries stored by the preprocessing phase.

The algorithmic part (the computation of the shortest-path trees from landmarks) is based on the preprocessing phase of CFLAT [15]. The preprocessing-time requirements are subquadratic in the graph size. As for the required space (also of subquadratic size [15]), in order to be able to efficiently handle continental-size time-dependent instances, we had to significantly improve the succinct representation of CFLAT, especially how the preprocessed travel-time summaries of the landmarks are stored.

The main intervention of this work is a lossless sparse matrix compression methodology for the succinct and space-efficient representation of the timestamped shortest-path trees from landmarks, avoiding a considerable increase in the access time for the preprocessed information. In particular, the preprocessed data conceptually contain, for each $\ell \in L$, a collection $T_\ell = \{T_\ell(t_\ell^1), \dots, T_\ell(t_\ell^{\lambda_\ell})\}$ of timestamped shortest-path trees rooted at ℓ, which are optimal for the carefully selected departure times from ℓ, $\{t_\ell^1, \dots t_\ell^{\lambda_\ell}\} \subset [0, T)$. The selection of the sampled departure-times was such that, for all possible departure-times $t \in [0, T)$, T_ℓ contains some trees providing, in worst case, an $(1 + \epsilon)$ approximation for $D[\ell, v](t)$.

Data Structure For Timestamped Predecessors. The novelty of our representation is the following: Rather than keeping a collection of trees per landmark, we maintain for each pair $(\ell, v) \in L \times V$ two sequences of the same length: (i) $\Delta_{\ell,v}$ for the sampled departure-times from ℓ (in increasing order), and (ii) $\Pi_{\ell,v}$ for the predecessors of v in the corresponding (ℓ, v)-paths. The departure-times in $\Delta_{\ell,v}$ are integers from $\{0, 1, \dots, 86399\}$ (considering an accuracy of seconds). Rather than using 3 bytes per cell, we exploit the fact that most departure times are smaller than $2^{16} = 65536$sec. Therefore, we keep an index $h_{\ell,v}$ of the latest departure-time in $\Delta_{\ell,v}$ that is smaller than 2^{16}. The first $h_{\ell,v}$ cells in $\Delta_{\ell,v}$ store exact departure-times, but the remaining cells only store the difference of the actual departure times from the offset 2^{16}. This way, all the cells in $\Delta_{\ell,v}$ require exactly 2 bytes. As for $\Pi_{\ell,v}$, every cell is the relative position of the predecessor of v, in its in-neighborhood list. 1 byte per cell is sufficient for real-world instances whose maximum in-degree is a small constant.

A first observation of an initial implementation of this data structure, was that a lot of space was consumed for storing duplicates of exactly the same pairs of sequences, for different landmark-destination pairs. For example, in the the continental-size EU instance with $18,010,173$ nodes, for $16,000$ landmarks one would need to store $576,325,536,000$ sequences. Nevertheless, we observed that there were only $1,632,168,375$ *distinct* sequences ($1,623,701,331$ departure-time sequences and $8,467,044$ predecessor sequences). To avoid this waste of space, we chose to store only pairs of (4-byte) pointers to sequences, among all landmark-destination pairs. After implementing this variant as well, we also observed that, in many cases, the same destination v^* had many repetitions of the same pairs of (4-byte)

pointers to sequences, over all the landmarks. Indeed, this is quite reasonable for landmarks located towards the same direction and roughly at the same distance from v^*. In order to avoid these repetitions of pairs of long pointers (8 bytes in total), we proceeded as follows (cf. Figure 2): We maintained a landmark-indexed dictionary $Ldict_{v^*}$, whose value for a key ℓ^* is a pointer s^* to the cell of an array S_{v^*} containing a *unique* pair (p^*, d^*) of pointers to the sequences Δ_{ℓ^*,v^*} and Π_{ℓ^*,v^*}.

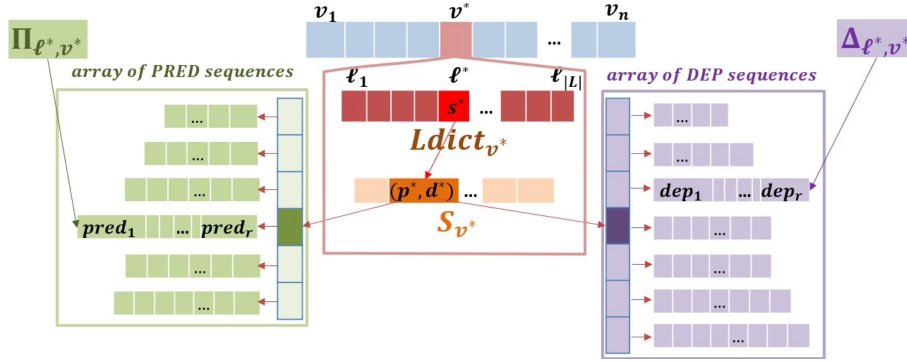

Figure 2 Data structure for the succinct representation of preprocessed information of TDAG.

The size $|S_{v^*}|$ is exactly the number of *distinct* pairs of pointers (to sequences) involving v^*, among all landmarks, and on average is significantly smaller than $|L|$. Each cell of S_{v^*} requires 8 bytes. On the other hand, for $Ldict_{v^*}$ we use bit-level representation of the stored values, with each cell consuming only $\log_2(|S_{v^*}|+indeg(v^*))$ bits. Even for 16,000 landmarks this is at most 14 bits per cell.

Finally, we observed from real-world instances that, more often than not, a node v^* might have always the same predecessor, independently of landmarks and departure times. In those cases, rather storing $Ldict_{v^*}$ and S_{v^*}, we simply stored this unique predecessor for v^*.

Lookup Procedure for Timestamped Predecessors. The lookup operation of preprocessed information, in order to get a time-dependent predecessor per landmark-node pair, is a procedure that is repeatedly used in the path-collection phase of the TDAG query algorithm (cf. PHASE-2 in Subsection 3.2). Briefly, the lookup operation takes as input a triple (ℓ, v, t_ℓ) of a landmark $\ell \in L$, a current node $v \in V$ and a departure-time from ℓ $t_\ell \in [0, T)$. The lookup procedure starts by locating $\Delta(\ell, v)$, and then conducts a binary search in it, to locate the closest sampled departure times $dep_i \leq t_\ell < dep_{i+1}$, in time $\mathcal{O}(\log(|\Delta(\ell, v)|))$. Consequently, the corresponding predecessors of v are located at positions i and $i+1$ of $\Pi(\ell, v)$, and thus are retrieved in $\mathcal{O}(1)$ time. Since the number of sampled departure times only partitions the period $[0, T)$ in small time intervals, it is independent of the network size (e.g., for the EU instance the maximum length of a sequence is 4407). Therefore, the entire lookup procedure takes $\mathcal{O}(1)$ time (e.g., at most 13 comparisons even for EU).

Because of this novel methodology for the succinct representation of the preprocessed data, preprocessing a large number of landmarks is now possible, even for continental-size datasets. In performance terms, this novel storage scheme provides a cache-friendly gain which beats the overhead of the bit-field and bit-mask extraction operations. This in turn leads to higher quality results and significantly lowers the observed relative error.

3.2 TDAG Query

The TDAG query algorithm executes three phases for serving a query $(o, d, t_o) \in V \times V \times [0, T)$:

PHASE 1: *Landmark Settling.* A forward TDD tree $T_f(t_o)$ is grown from (o, t_o), until either d or a set $F \subset L$ of the N closest landmarks are settled. Subsequently, a reverse BFS from d is executed, exploring the neighborhood around d in a backward fashion. The growth of the reverse BFS tree T_r is stopped when its size becomes equal to $|T_r| = c \cdot |T_f(t_o)|$, for some constant $c \geq 1$ (our experimental analysis showed that $c = 1.2$ is an appropriate choice). It should be mentioned that we experimented also with growing a reverse TDD tree towards d, but this approach was more time-consuming and the resulting AG, to be constructed in the next phases, was eventually similar to the one constructed using the reverse BFS tree towards d.

PHASE 2: *Path Collection.* Using the preprocessed data, our next task is to construct a subgraph of shortest paths from the N landmarks of $T_f(t_o)$, with their own departure-times which have been already computed in PHASE 1, towards each leaf node of T_r. This is done as follows: starting from each leaf node of the reverse BFS tree T_r, we recursively move backwards towards each $\ell \in F$, by looking-up predecessors in the timestamped shortest paths originating at the landmarks of $T_f(t_o)$. All the edges that participate in these paths connecting the landmarks in F to the leaf nodes of T_r, become marked. The initial alternative graph H consists of the union of the two trees $T_f(t_o)$ and T_r of PHASE 1, plus the marked edges of PHASE 2. We continue expanding the forward TDD tree of PHASE 1 towards d, by working only on H, until all nodes in H are settled. This allows us to obtain a tentative arrival-time $\tilde{t}_d$ at d: $\tilde{t}_d = t_o + D_H[o, d](t_o) \leq t_o + \min_{\ell \in T_f(t_o) \cap L}\{D[o, \ell](t_o) + \overline{D}[\ell, d](t_o + D[o, \ell](t_o))\}$. Clearly $\tilde{t}_d$ is an upper-bound of the earliest-arrival-time $t_d = t_o + D[o, d](t_o)$. The quality of this upper bound depends on the choice of the precision ϵ of the preprocessed information (cf. the analysis of CFLAT [15] for further details), the number N of settled landmarks within $T_f(t_o)$, and the size of the reverse BFS tree T_r.

PHASE 3: *Path Pruning.* The graph H produced by PHASE 2 is already smaller than G. Nevertheless, it is further pruned so as to meet the three quality criteria for an alternative graph: small path overlapping, stretch, and size. This is done in three steps.

Step 3.1 We first aim at a loose pruning over H, in order to obtain a subgraph containing a smaller number of candidate *od*-paths with reasonable travel-times. In particular, any node u in H whose shortest travel-time from o to d via u is greater than the targeted upper-bound, i.e., $D_H[o, u](t_o) + D_H[u, d](t_o + D_H[o, u](t_o)) > maxStretch \cdot D_H[o, d](t_o)$, is removed.

Step 3.2 For further reducing the number of the candidate *od*-paths, we use initially the Plateau method [4, 21] by running, within H, TDD from (o, t_o), and RTDD from $(d, t_o + D_H[o, d](t_o))$. Any edge not belonging to the resulting Plateau candidate *od*-paths, is removed from H. The Penalty pruning method [5, 21] is then applied, to prune further the subgraph H. At each Penalty iteration, TDD runs on H, computing a new time-dependent shortest path $\pi_{o,d}$, which is marked and is added to the solution set E_s. Additionally, the edges in E_s and the incoming edges incident to the nodes in $\pi_{o,d}$ are penalized with an increasing penalty factor $p(e)$ and $r(e)$, respectively, initially set to 0. For each edge $e = uv \in E_s$, its travel-time is increased to $D[e](t)^{(penalized)} = (p(e)^{(current)} + 1)D[e](t)^{(original)}$; otherwise, if u or $v \in \pi_{o,d}$ and $e \notin \pi_{o,d} \cup E_s$, its travel-time is increased to $D[e](t)^{(penalized)} = (r(e)^{(current)} + 1)D[e](t)^{(original)}$. The penalty factors of the affected edges are increased at the end of each step to $p(e)^{(new)} = p(e)^{(old)} + p_c$ and $r(e)^{(new)} = r(e)^{(old)} + r_c$, where $p_c > 0$ and $r_c > 0$ are constants. The process is repeated

until a sufficient number of alternative paths is found, or the travel time adjustments of $\pi_{o,d}$ paths bring no better results. At the end, the unmarked edges are removed. In order to speedup the Penalty approach at path computation, the time-dependent variant of A^* [7, 12] can be used in place of TDD. For each node of H, its distance towards d which is already computed from RTDD during the Plateau phase, can be used as a lower bound for the time-dependent A^* heuristic.

Step 3.3 The final pruning of H is performed via a ranking procedure. Initially, if a path $\pi_{x,y}$ in H has $outdeg(x) \geq 2$ and $indeg(y) \geq 2$, and $\forall v \in \pi_{x,y} - \{x, y\}$ $outdeg(v) = indeg(v) = 1$ (i.e., it increases by one the *decisionEdges*), then it is considered as a *decision-path* and it is ranked by the function $rank(\pi_{x,y}, t) = \sum_{e \in \pi_{x,y}} (share[e](t) - stretch[e](t))$ that represents the contribution of π_{xy} in terms of averageDistance and totalDistance in H. The ranked decision-paths are sorted by increasing ranking order in a priority queue Q. Then an iterative procedure starts, where in each iteration a path π_{xy} is dequeued from Q. If the condition $outdeg(x) \geq 2$ and $indeg(y) \geq 2$ remains in effect, then $\pi_{x,y}$ is removed from H, leading to a decrease of the *decisionEdges* by one. After the removal of $\pi_{x,y}$, if for $v \in \{x, y\}$ it holds that $outdeg(v) = indeg(v) = 1$, then a new decision path π is revealed which has v as an internal node. π is ranked and inserted in Q, in order to be considered along with the rest of decision paths. The iterative procedure stops when $decisionEdges \leq maxDecsionEdges$.

4 Experimental Evaluation

Experimental Setup and Goal. TDAG was implemented in C++ (GNU GCC version 9.3.0). All the experiments were conducted on a AMD Ryzen Threadripper 3960X 24-Core 3.8GHz Processor, with 256GB of RAM, running Ubuntu Linux (20.04 LTS). We used 24 threads for the preprocessing phase of CFLAT [15], using as preprocessing precision $\epsilon = 0.1$.

Three typical benchmark instances for testing speedup techniques and oracles in time-dependent road networks are used in our experiments, kindly provided to us by TomTom and PTV for scientific purposes. The real-world instance of Berlin (BE) was provided by TomTom, has $473,253$ nodes and $1,126,468$ edges, and contains edge-travel-time functions taken from historical data of a typical working day (Tuesday) in a typical urban environment. The instances of Germany (GE) and Europe (EU) were provided by PTV, and contain edge-travel-time functions of a typical working day, in nation-wide and continental road networks, respectively. GE has $4,692,091$ nodes and $10,805,429$ edges, and is a real-world instance. EU has $18,010,173$ nodes and $42,188,664$ edges, and is a synthetic time-dependent benchmark instance that is typically considered in the related literature.

The TDAG query algorithm was executed on a single thread. For the sake of comparison, in all the query algorithms that have been evaluated in this work, we used the same set of $10,000$ queries chosen independently and uniformly at random without repetitions (*iuar*) from $V \times V \times [0, T)$ in each instance, for randomly selected departure-times from $[0, T)$. The static (forward-star) variant of the PGL library [19] was used for the graph representation. For Dijkstra-based algorithms, we used as priority queue Sander's implementation[2] of the sequence heap [22].

In [15], various methods were considered for the selection of the landmark set. In this work, we only consider the sparse-random (SR) method: the landmark nodes are chosen sequentially, each new landmark is chosen iuar from the remaining nodes, and excludes a free-flow neighborhood of nodes around it from future landmark selections.

[2] `http://algo2.iti.kit.edu/sanders/programs/spq/`.

The goal of our experimental evaluation was twofold:

(i) we investigated the *scalability* of TDAG, i.e., how smoothly it trades higher query times with better quality of the alternative graph H, using the value of N as our tuning parameter;

(ii) we compared TDAG's performance with the performances of straightforward time-dependent variants of existing techniques for constructing alternative graphs in static graphs [2, 21], which serve as our baseline approaches.

Moreover, the relative error *ApxErr*, defined as

$$ApxErr = \frac{D_H[o,d](t) - D_G[o,d](t)}{D_G[o,d](t)},$$

provides the approximation accuracy of H, that is, how close $D_H[o,d](t)$ is to $D_G[o,d](t)$, given that $D_H[o,d](t) \geq D_G[o,d](t)$.

Experimental Results. Our bit-level data compression technique (cf. Section 3.1) turned out to be beneficial. The byte-based approach of CFLAT [15] for the succinct representation of the travel-time summaries of 2000 landmarks chosen with SR (SR2K) consumed space of 5.2GB in Berlin, 53.6GB in Germany, and 107.2GB in Europe. Using the new profiling, bit-level based approach of TDAG, the preprocessed data for SR2K landmarks consumes space of 0.28GB in Berlin, 3.2GB in Germany, and 31.05GB in Europe. That is, the exploitation of the bit-level representation of a sparse matrix, without sacrificing the landmark and node indexing, leads to a significant reduction of about 70% in space requirements, which in turn allows for the selection of larger landmark sets, especially for continental-size instances.

Table 1 Quality measures and execution times of TDAG.

Network	Landmark Set	N	Target Function	Total Dist	Avg Dist	Decision Edges	Apx Err (%)	Time (ms)
BE	SR4000	1	1.53	1.54	1.01	4.63	0.48	0.52
		2	1.98	2.00	1.02	7.69	0.06	0.89
		4	2.40	2.43	1.03	9.07	0.02	1.50
		10	2.97	3.02	1.04	9.68	0.01	3.11
		32	3.65	3.71	1.06	9.72	0.00	8.45
		76	3.99	4.06	1.07	9.62	0.00	18.86
		100	4.06	4.14	1.08	9.58	0.00	25.80
		250	4.22	4.30	1.08	9.46	0.00	64.44
GE	SR8000	1	1.50	1.51	1.01	8.60	0.51	1.31
		2	1.93	1.94	1.02	9.96	0.06	2.80
		8	2.77	2.81	1.04	9.93	0.00	11.38
		18	3.26	3.32	1.06	9.86	0.00	28.89
		25	3.45	3.51	1.07	9.80	0.00	43.33
		64	3.88	3.96	1.09	9.63	0.00	135.04
		100	4.02	4.11	1.09	9.54	0.00	213.05
		200	4.15	4.25	1.10	9.40	0.00	384.15
EU	SR16000	1	1.43	1.43	1.01	8.63	0.85	4.30
		6	2.07	2.09	1.02	9.95	0.55	21.95
		18	2.51	2.54	1.03	9.92	0.55	80.61
		64	3.09	3.15	1.05	9.74	0.55	330.72
		100	3.30	3.36	1.06	9.63	0.55	514.45
		150	3.47	3.54	1.07	9.51	0.55	770.48
		200	3.57	3.64	1.07	9.42	0.55	965.07
		250	3.62	3.69	1.07	9.32	0.55	1237.80

In Table 1 and 2, we report the results of our experimental evaluations of TDAG on the approximation accuracy *ApxErr* (relative error in %) and the various quality indicators[3] (cf. Section 2): *targetFunction* ($TargFun$), *totalDistance* ($TotDist$), *averageDistance* ($AvgDist$) and *decisionEdges* ($DecEdges$). Similar to [21], in order to evaluate the quality of AG, the aggregate quality indicator $TargFun$ is used, defined as follows: $TargFun = totalDistance + 1 - averageDistance$. In all cases the alternative graphs are evaluated using the following constraints: $maxStretch \leq 1.2$, $averageDistance \leq 1.1$, and $decisionEdges \leq 10$. In the path pruning step, the penalty constants were set to $p_c = 0.3$ and $r_c = 0.1$.

Table 1 demonstrates the effect of the parameter N on the execution time of TDAG, as well as on the quality of the constructed AG. As N grows, PHASE 1 becomes computationally more expensive, but the relative error rapidly drops towards 0 for BE and GE. This is due to the fact that as N increases, the expanded (forward) TDD tree gets bigger and the resulting *od*-paths increase in number, but we also get more candidate *od*-paths providing an AG of better quality. As for EU, the relative error seems to stop at 0.55, because of the steepest slopes of the earliest-arrival-time functions (which necessitate an increased number of sampled departure times during the preprocessing phase), the propagation of floating point rounding errors along the edges of long paths, and the smaller density $|L|/|V|$ of the preprocessed landmarks, compared to the instances of BE and GE. All these issues can be tackled by affording more memory for the computations.

In Table 2 we present the results of the baseline approaches and their comparison to TDAG. DPP is a combination of the Plateau and the Penalty methods [2], which collects and evaluates the candidate *od*-paths using a greedy selection approach. In our time-dependent context, Dijkstra's algorithm was replaced by its time-dependent variant, TDD. APP is again the combination of the Plateau and Penalty methods of [2, 21], which uses the ALT pruner and filtering approach [21]. Dijkstra's algorithm was again replaced by TDD, and for the lower bounds required by ALT the constant free-flow minimum-travel-time distances were used (i.e., each edge has as scalar cost corresponding to its minimum travel time). DPP does not require prepossessing, while APP requires a linear in space and super-linear in time prepossessing phase for computing the lower bounds required by ALT. Regarding the computation of AG (column q-time in Table 2), both baseline approaches have a slow path collection phase. DPP constructs a subgraph H that is huge in size, using an expensive phase of pure TDD, as there are no heuristics. APP improves the time of the path collection phase, but the lower bounds are not tight for a time-dependent metric. In both cases the achieved quality is high, at the cost of large processing times.

From Tables 1 and 2, it is clear that for all instances the configurations of TDAG, achieving analogous aggregate quality, require execution times about two orders of magnitude smaller than that of DPP. In particular, the achieved speedups are more than 102.7 for BE, 90.8 for GE and 37.9 for EU. Similarly, the configurations of TDAG achieving similar values of the target function, are faster than APP about one order of magnitude, as the instance size increases. In particular, the speedups are 2.1 for BE, 4.8 for GE and 8.3 for EU.

5 Conclusions

In this work we present TDAG, a novel algorithm for computing alternative routes in FIFO-abiding time-dependent road networks, based on succinctly stored preprocessed travel information. One of TDAG's strong features is that it can smoothly trade-off the quality

[3] To simplify notation, we omit in the rest of the paper the departure-time t notation.

Table 2 Speedups of TDAG over DPP (with TDD) and APP (with A^* and free-flow lower-bounds).

network	method	Target	q-time	speedup
BE	DPP	3.01	319.38	102.7
	TDAG vs DPP	2.97	3.11	
	APP	4.21	134.73	2.1
	TDAG vs APP	4.22	64.44	
GE	DPP	3.27	2623.36	90.8
	TDAG vs DPP	3.26	28.89	
	APP	4.17	1860.80	4.8
	TDAG vs APP	4.15	384.15	
EU	DPP	3.36	19511.93	37.9
	TDAG vs DPP	3.30	514.45	
	APP	3.89	10266.29	8.3
	TDAG vs APP	3.62	1237.80	

of the resulting AG with the required execution time, via proper choices of its parameter N. This feature provides a significant advantage over all existing approaches, which have only one solution set of *od*-paths. Our experimental evaluation on three real-world instances demonstrated that TDAG clearly outperforms both baseline approaches (DPP and APP), since it provides time-dependent alternative routes of the same quality as DPP and APP within smaller execution times. TDAG can also provide "quick-and-dirty" alternative routes with a speedup of more than 100 over both DPP and APP, but it can continue its execution until it finds alternative routes of the same quality as DPP and APP, still much faster (less than half time for BE, or one fifth of time for GE) than these two baseline approaches.

References

1. Ittai Abraham, Daniel Delling, Andrew V Goldberg, and Renato F Werneck. Alternative routes in road networks. In *Experimental Algorithms*, volume 6049 of *LNCS*, pages 23–34. Springer, 2010.
2. Roland Bader, Jonathan Dees, Robert Geisberger, and Peter Sanders. Alternative route graphs in road networks. In *Theory and Practice of Algorithms in (Computer) Systems*, volume 6595 of *LNCS*, pages 21–32. Springer, 2011.
3. G Veit Batz, Robert Geisberger, Peter Sanders, and Christian Vetter. Minimum time-dependent travel times with contraction hierarchies. *ACM Journal of Experimental Algorithmics*, 18(1.4):1–43, 2013.
4. Camvit: Choice routing, 2009. URL: `http://www.camvit.com`.
5. Yanyan Chen, Michael GH Bell, and Klaus Bogenberger. Reliable pretrip multipath planning and dynamic adaptation for a centralized road navigation system. *IEEE Transactions on Intelligent Transportation Systems*, 8(1):14–20, 2007.
6. Daniel Delling and Dorothea Wagner. Pareto paths with sharc. In *Experimental Algorithms*, volume 5526 of *LNCS*, pages 125–136. Springer, 2009.
7. James Doran. An approach to automatic problem-solving. *Machine Intelligence*, 1:105–127, 1967.
8. Stuart E Dreyfus. An appraisal of some shortest-path algorithms. *Operations Research*, 17(3):395–412, 1969.
9. eCOMPASS project, 2011-2014. URL: `http://www.ecompass-project.eu`.

10 David Eppstein. Finding the k-shortest paths. *SIAM Journal on Computing*, 28(2):652–673, 1998.

11 Pierre Hansen. Bicriterion path problems. In *Multiple criteria decision making theory and application*, pages 109–127. Springer, 1980.

12 Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

13 Moritz Kobitzsch. An alternative approach to alternative routes: Hidar. In *Algorithms*, volume 8125 of *LNCS*, pages 613–624. Springer, 2013.

14 Felix Koenig. Future challenges in real-life routing. In *Workshop on New Prospects in Car Navigation*, February 2012.

15 Spyros Kontogiannis, Georgia Papastavrou, Andreas Paraskevopoulos, Dorothea Wagner, and Christos Zaroliagis. Improved oracles for time-dependent road networks. In *Algorithmic Approaches for Transportation Modelling, Optimization, and Systems*, volume 59 of *OASIcs*, pages 4:1–4:17. Dagstuhl Publishing, 2017.

16 Spyros Kontogiannis, Dorothea Wagner, and Christos Zaroliagis. Hierarchical time-dependent oracles. In *Algorithms and Computation*, volume 64 of *LIPIcs*, pages 47:1–47:13. Dagstuhl Publishing, 2016.

17 Spyros Kontogiannis and Christos Zaroliagis. Distance oracles for time-dependent networks. *Algorithmica*, 74(4):1404–1434, 2016.

18 Dennis Luxen and Dennis Schieferdecker. Candidate sets for alternative routes in road networks. In *Experimental Algorithms*, volume 7276 of *LNCS*, pages 260–270. Springer, 2012.

19 Georgia Mali, Panagiotis Michail, Andreas Paraskevopoulos, and Christos Zaroliagis. A new dynamic graph structure for large-scale transportation networks. In *Algorithms and Complexity*, volume 7878 of *LNCS*, pages 312–323. Springer, 2013.

20 Ernesto Queiros Vieira Martins. On a multicriteria shortest path problem. *European Journal of Operational Research*, 16(2):236–245, 1984.

21 Andreas Paraskevopoulos and Christos Zaroliagis. Improved Alternative Route Planning. In *Algorithmic Approaches for Transportation Modelling, Optimization, and Systems*, volume 33 of *OASIcs*, pages 108–122. Dagstuhl Publishing, 2013.

22 Peter Sanders. Fast priority queues for cached memory. *ACM Journal of Experimental Algorithmics*, 5:1–25, 2000.

23 Jin Y Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17(11):712–716, 1971.