Karlsruhe Institute of Technology

Cooperative Route Planning On Time-Dependent Road Networks

Master Thesis of

Nils Werner

At the Department of Informatics
Institute of Theoretical Informatics

Reviewers:	PD Dr. Torsten Ueckerdt
	Prof. Dr. Peter Sanders
Advisor:	Tim Zeitz

Time Period: 1st October 2021 – 1st April 2022

Statement of Authorship

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, March 31, 2022

Abstract

We study efficient algorithms for route planning in road networks. Cooperative routing applications consider the previously assigned routes of all road users in the current request. The aim is to distribute traffic evenly over the entire road network. Besides that, another goal is to detect and avoid impending traffic bottlenecks at an early stage.

In this thesis, we present a simple model to implement cooperative route planning. For this purpose, the underlying road network is modeled as a graph with time-dependent edge weights. Each request causes a modification of the edge weights along the proposed route.

We apply the A* algorithm to solve the shortest path queries efficiently. This goal-directed approach uses a heuristic (also called potential) to skip irrelevant nodes during the search. However, existing heuristics are often inaccurate on time-dependent graphs. To improve the performance of the A* algorithm, we introduce time-dependent A* potentials as a novel approach. The focus is on the design of these time-dependent potentials as well as their implementation and evaluation.

Our experiments show that the cooperative model needs further adaptation for use in a real-world application. However, we also observe that our time-dependent A* potentials further improve the efficiency of the A* algorithm. Compared to existing heuristics, the running time is improved by up to an order of magnitude. This makes time-dependent A* potentials interesting for other application areas, too.

Deutsche Zusammenfassung

Wir untersuchen effiziente Algorithmen zur Routenplanung in Straßennetzwerken. Kooperative Anwendungen zeichnen sich dadurch aus, dass sie die Routen aller Verkehrsteilnehmer in der aktuellen Anfrage berücksichtigen. Dabei wird eine gleichmäßige Verteilung des Verkehrs auf das gesamte Straßennetzwerk angestrebt. Ein weiteres Ziel besteht darin, drohende Verkehrsengpässe frühzeitig zu erkennen und zu vermeiden.

In dieser Arbeit stellen wir ein einfaches Modell zur Umsetzung kooperativer Routenplanung vor. Hierfür wird das zugrundeliegende Straßennetz als Graph mit zeitabhängigen Kantengewichten modelliert. Ferner werden durch jede gestellte Anfrage die Kantengewichte entlang der jeweils vorgeschlagenen Route verändert.

Um die Kürzeste-Wege-Anfragen schnell zu verarbeiten, nutzen wir den A* Algorithmus. Dieser zielgerichtete Ansatz verwendet eine Heuristik (auch Potential genannt), um irrelevante Knoten während der Suche frühzeitig auszuschließen. Auf zeitabhängigen Graphen erweisen sich bestehende Heuristiken jedoch als ungenau. Um die Performance des A* Algorithmus weiter zu verbessern, stellen wir mit zeitabhängigen A*-Potentialen einen neuen Ansatz vor. Der Fokus liegt dabei auf dem Entwurf zeitabhängiger Potentiale sowie deren Implementierung und Evaluation.

Unsere Experimente zeigen, dass das kooperative Modell weitere Anpassungen zum Einsatz als Endnutzeranwendung benötigt. Allerdings stellen wir auch fest, dass unsere zeitabhängigen Potentiale die Effizienz des A*-Algorithmus weiter verbessern. Im Vergleich zu bestehenden Heuristiken lässt sich eine Verbesserung der Laufzeit um bis zu einer Größenordnung beobachten. Dies macht zeitabhängige A*-Potentiale auch für andere Anwendungsgebiete interessant.

Contents

1. Introduction

Modern route planning systems for road networks have become an integral part of our lives. In the past decade, its usage has mostly shifted from separate navigation devices to mobile applications. Nowadays, a large proportion of the requests are processed by a few providers, such as *Google Maps*[1] or *TomTom*[2]. The relevance of centralized route planning applications will continue to increase with the usage of autonomous vehicles. As these vehicles rely on routing applications, the given requests could represent the entire traffic flow. This would enable comprehensive control and optimization of traffic flows by a centralized authority. However, popular route planning applications mostly suggest routes in a *selfish* manner. Regardless of the route choices by other road users, these applications assign the currently best route to each driver. Especially in unusual traffic situations (e. g. congested roads), following this approach leads to sub-optimal traffic distributions.

The idea of *cooperative routing* represents a counter design to the selfish routing approaches which have dominated to date. Its main goal is to enforce cooperation between the road users and distribute traffic evenly in the road network. By this, cooperative route planning aims to achieve a *social optimum* state in which the total travel time of all users is minimized. For this reason, it is important to predict future traffic conditions based on the given set of user requests. Consequently, the routing algorithm cannot solely base its calculations on current traffic data. It must also consider the planned route information derived from the other drivers.

Implementing cooperative routing requires careful engineering. Existing (selfish) route planning applications apply graph theory to model the underlying road network. A shortest path algorithm is used to determine the fastest route between two locations in the network. These algorithms must solve the given queries quickly (i. e. within milliseconds) to be suitable for real-time routing applications. In addition to that, cooperative routing introduces further complexities. Particularly, each request must be solved with respect to all previously assigned routes. This motivates our research for efficient algorithms for cooperative route planning.

1.1 Related Work

There has been lots of research on designing efficient algorithms for route planning in the past decade. Bast et al. [BDG+16] provide an extensive overview. Throughout these

[1] Google Maps: `https://www.google.com/maps/`
[2] TomTom: `https://www.tomtom.com`

works, the underlying road network is modeled as a weighted graph. Solving point-to-point queries corresponds to the *Shortest-Path Problem*, for which *Dijkstra's* algorithm [Dij59] is a well-known solution. On continental-sized graphs, however, solving a point-to-point query with Dijkstra takes several seconds. This makes the algorithm infeasible for many real-time applications. Therefore, several speed-up techniques have been proposed to accelerate the basic algorithm by several orders of magnitude. These techniques aim to reduce the *search space size*, i. e. the number of visited nodes during a query, by exploiting auxiliary data. For example, *goal-directed* approaches like the *A* algorithm* [HNR68] apply a pre-calculated heuristic that estimates the remaining distance to the given target node. Depending on the precision of the estimates, the goal-directed search only considers a fraction of the nodes visited by a standard Dijkstra run. Apart from the A* algorithm, many other speed-up techniques consist of two stages. In the initial *preprocessing* step, the graph is extended with auxiliary data. As this step is run infrequently, it may take several hours or even days. Subsequently, the auxiliary data can be exploited to solve the queries faster. One of the most popular techniques are *Contraction Hierarchies* (CH) [GSSD08]. In this *hierarchical* approach, the preprocessing step classifies the nodes by their *importance* and produces additional *shortcut* edges. These shortcuts are later used to skip unimportant nodes during the query. More recently, *Customizable Contraction Hierarchies* (CCH) [DSW14] has been proposed as a CH extension. Inspired by *Customizable Route Planning* [DGPW17], the preprocessing step is divided into a metric-independent preprocessing and a metric-dependent *customization* step. In the latter stage, the updated edge weights are applied to the shortcuts within a few seconds. This allows regular adjustments to the current traffic situation. Both CH and CCH accelerate Dijkstra's algorithm significantly. On continental-sized graph instances, their query times are less than a millisecond. Nevertheless, there are even faster speed-up techniques, for example *Hub Labeling* [ADGW11] and *Transit Node Routing* [ALS13].

All of the aforementioned approaches model the road network with scalar edge weights. In reality, however, the travel time along a segment strongly depends on the time of the day. For example, the travel time during peak hours in the morning and afternoon is usually higher than around midnight. In *time-dependent route planning*, the travel time of each road segment is modeled as a function that depends on the current daytime. In 1966, Cooke and Halsey modified Dijkstra's algorithm to handle time-dependent edges [CH66]. Recently, the research interest in time-dependent route planning has also increased. For instance, some A* heuristics have been extended to cover time-dependence [DW09, SZ21]. Their estimates for each node are constant over the entire day, though. Besides A*, several other speed-up techniques have also been modified for time-dependent route planning [BGSV13, SWZ21]. Due to the increased complexity of the edge weights, preprocessing time and memory consumption are significantly higher. Some storage-intensive techniques (e. g. Hub Labeling or Transit Node Routing) are therefore not easily applicable to time-dependence. To reduce the memory consumption, some approaches [BGSV13] approximate travel time profiles during the preprocessing stage. Unfortunately, this leads to either inexact results or additional effort during the query phase. Currently, *CATCHUp* [SWZ21] offers the best trade-off between storage consumption and query time. The algorithm extends CCH and enables a customization within a few minutes. Experiments have shown that CATCHUp requires less than ten milliseconds for queries on continental-sized graphs [SWZ21]. The authors also provide a detailed running time comparison with other time-dependent speed-up techniques.

In a cooperative setting, we consider the behavior of other road users and take into account their route choices. In the past 60 years, the *Traffic Assignment* problem has been studied in multiple variants (see [FH95] for a comprehensive overview). Its goal is to determine how traffic is distributed over an underlying transportation network. We emphasize that the

results are not primarily used for real-time navigation. Instead, the expected behavior of all drivers provides important insights for urban transportation planning [PERW15]. The traffic demand is typically given as *origin-destination* (OD) matrices. Each matrix entry represents the quantity of traffic that has to be routed between the corresponding origin and destination node. In other studies, a set of concrete OD-pairs is provided [BSW19b]. Assuming that each driver aims to minimize the travel time, the system converges to an equilibrium state. In this state, no road user can improve the currently taken route.

Beckmann et al. [BMW56] have formalized the traffic assignment problem as a mathematical program. Its solution corresponds to an equilibrium flow pattern [BSW19b]. Over the decades, several algorithms have been proposed to solve the traffic assignment problem. The *Frank-Wolfe* algorithm [FW56] is one of the most famous approaches. Some other methods are outlined in [PERW15]. Up to this point, the mentioned approaches do not consider time-varying traffic flows and are hence considered as *static*. For an overview about *Dynamic Traffic Assignment*, we refer to a survey by Peeta et al. [PZ01].

The scope of traffic assignment algorithms is often restricted to cities or metropolitan areas. In most related work, the limited performance only allows considering graphs with a few thousand nodes and edges. For example, the graph instances used in a recent survey [PERW15] are bound to 15 000 nodes and 40 000 edges. The shortest path calculations are often the main performance bottleneck of these algorithms. Surprisingly, we are only aware of three studies that apply speed-up techniques from route planning. The first known approach from Luxen and Sanders [LS11] is based on Contraction Hierarchies. A more recent work by Buchhold et al. [BSW19b] applies Customizable Contraction Hierarchies to utilize the customization step between each iteration of the traffic assignment algorithm. Both approaches consider OD-pairs, but there is also an adaption to OD-matrices for the latter work [SN20]. All of the mentioned approaches provide significant speedups compared to earlier work. However, time-dependence has not been taken into account in these works.

Traffic assignment algorithms require that all queries are known in advance. In route planning applications, however, a real-time assignment has to be made at the time of a user's request. Furthermore, the future inputs are unknown at this time. Despite the prominence of cooperative route planning, few works incorporate the behavior of other road users into efficient route planning algorithms. Existing *dynamic route planning* approaches either predict future traffic conditions with stochastic models based on the current traffic situation [LPBM17] or combined with historical data [HMT17, ZMCS+19, WVLM11]. However, none of these works utilize efficient speed-up techniques for time-dependent route planning.

1.2 Contribution and Outline

We study a model of cooperative route planning in which the queries are processed *on-line*, i. e. by ascending departure time and unaware of upcoming requests. Similar to a real-world scenario, the queries are received one after another and must be resolved in real-time. The given optimization problem is to minimize the aggregated travel time of all road users. Our proposition is that the best results are achieved by considering all available route information from previous requests in the current query.

Due to the on-line query processing, the applied shortest path algorithm cannot predict the impact of upcoming requests. To incorporate future traffic conditions more precisely, subsequent adjustments to the routes are necessary. For simplicity, however, our model is limited to one-time route assignments. We are aware that this approach yields suboptimal results in practice, especially for the queries processed first. Implementing a complex

traffic distribution algorithm is beyond the scope of our work, though. We are primarily interested in providing a proof of concept for the plausibility of cooperative route planning. Nevertheless, integrating a cooperative model into real-world route planning applications would present an interesting extension of our work.

The main focus of this work is designing efficient shortest path algorithms for the mentioned model of cooperative route planning. To the best of our knowledge, this is the first work that addresses time-dependent graphs with permanently changing edge weights. To consider all available information about future traffic conditions, we must process the requests on the current edge weights. This poses further challenges related to the usage of speed-up techniques. In particular, it is too time-consuming to perform the preprocessing or customization step before each query. Despite that, A* potentials are still suitable for this scenario because their inaccuracy does not imply sub-optimal routes. The applied heuristics deliver correct results as long as their estimates provide a lower bound of the actual travel times. In our model, the travel times only increase with a rising number of queries. Therefore, A* potentials remain valid throughout all requests, although the estimation quality continuously degrades. We recommend adjusting the potential estimates frequently. This separate step can be executed in the background, similar to the customization stage of the CCH algorithm.

Our results include a novel approach of time-dependent A* potentials. More precisely, the potential estimates depend on the node as well as the departure time at the origin. Although cooperative route planning is the primary motivation for this study, we point out that time-dependent potentials have other applications beyond the scope of this work. Our baseline heuristic is the current state-of-the-art *CCH-Potential* [SZ21]. This heuristic provides static lower-bound estimates over the entire day and is therefore time-independent. Despite that, the time-dependent A* potentials proposed in our work are largely derived from the CCH-Potential algorithm. Our first approach, the *Multi-Metric-Potential*, maintains several weight functions (*metrics*) to cover overlapping time intervals throughout the day. In each query, the algorithm selects a time interval that covers the entire trip. Then, it suffices to use the lower bound of each travel time function within the given time frame as an estimate. The second approach (*Corridor-Lowerbound-Potential*) is more fine-grained and determines the relevant time interval for each edge individually. While its preprocessing step is time-consuming, this approach provides an even more precise heuristic for the A* algorithm.

We conduct several experiments to evaluate the performance of the time-dependent A* potentials as well as the storage consumption and plausibility of our cooperative model. Our experiments show that both the *Multi-Metric-Potential* and *Corridor-Lowerbound-Potential* outperform the *CCH-Potential*. On graphs with real-world traffic predictions, speedups of up to a factor of 10 are achieved. We also evaluate our cooperative distribution algorithm and conclude that further improvements are necessary for practical usage.

This thesis is organized as follows. First, basic definitions and algorithms are introduced in Chapter 2. Chapter 3 covers the main aspects of cooperative routing and explains our traffic distribution algorithm. In Chapter 4, we present our time-dependent A* potentials and prove their correctness. A comprehensive experimental evaluation of the time-dependent potentials as well as our cooperative distribution algorithm is provided in Chapter 5. Finally, Chapter 6 summarizes our main observations and discusses future work related to cooperative route planning.

2. Preliminaries

In this chapter, we define the basic notation used throughout the thesis. We also describe a model commonly used for time-dependent route planning and introduce some efficient algorithms for the Shortest Path Problem.

2.1 Problem Statement

We want to design and implement efficient shortest path algorithms for cooperative route planning. In our model, the queries are given on-line, sorted by ascending departure time. Each query should be solved with respect to all route information available up to this point. Therefore, the algorithm must operate on the current edge weights to obtain exact results. Overall, we want to find out whether predicting future traffic conditions from previously assigned routes helps to reduce the total travel time of all road users.

The research focus is on developing speed-up techniques to accelerate the time-dependent shortest path computations. Our goal is to provide algorithms that can handle permanently changing edge weights. We primarily focus on A* potentials for this reason. Existing competitors, such as *CCH-Potentials* [SZ21], should be outperformed in terms of query time. Furthermore, the algorithm should allow fast estimate adjustments to the current traffic situation.

2.2 Basic Definitions and Models

To solve the given problems, we first describe how to model the underlying road network using graph theory. A *graph* is a tuple $G = (V, E, len)$ consisting of a set of *nodes* V, a set of *edges* $E \subseteq V \times V$, and a *weight function* $len : E \to \mathbb{R}_+$. Edges represent links between two nodes. Moreover, the weight function assigns a non-negative weight to each edge. An underlying road network can be modeled by providing an edge for each road segment. Consequently, the nodes either represent connecting points (e. g. intersections) or dead ends. To enable one-way roads in our model, we use *directed* edges. A segment between two adjacent nodes $v, w \in V$ is traversable in both directions if $\{(v, w), (w, v)\} \subseteq E$. Furthermore, the weight function typically represents the expected travel time along each segment. Alternatively, it is also possible to apply other criteria such as travel distance or fuel consumption. Custom preferences, for example avoiding highways/toll roads, can be incorporated by assigning an infinite weight to the affected edges. In the following, we also denote the weight function as *metric*. However, we do not require any of the properties

associated with the mathematical term of a metric. We also point out that the terms *node* and *vertex* as well as *edge* and *arc* are used interchangeably in literature.

Finding the fastest route in a road network corresponds to the *Shortest Path Problem* (SPP) in our modeled graph. For now, we focus on *point-to-point* queries between a *source* node $s \in V$ and a *target* node $t \in V$.

Definition 2.1 (Shortest Path Problem)**.** *Let $G = (V, E, len)$ be a graph and let $s, t \in V$ be the source and target node of the current query. The Shortest Path Problem asks for a path $P = (s, \dots, t)$ such that the sum of edge weights along P is minimized.*

The expected travel time between the source and the target node is given by the sum of edge weights along the shortest path P. In the following, we will use the notation of the *shortest distance function* $d(\cdot)$ to express the fastest travel time between *any* pair of nodes.

Definition 2.2 (Shortest Distance Function)**.** *Consider the graph $G = (V, E, len)$ and the shortest path $P = (u, v_1, \dots, v_k, w)$ between two arbitrary nodes $u, w \in V$. The shortest distance function $d : (V \times V) \to \mathbb{R}_+$ is defined as*

$$d(u, w) = len((u, v_1)) + \cdots + len((v_k, w))$$

for all nodes pairs u, w.

The *Floyd-Warshall Algorithm* [Flo62] can be used to calculate all entries of $d(\cdot)$. For point-to-point queries between two nodes $s, t \in V$, however, it is sufficient to obtain $d(s, t)$ and its corresponding path. *Dijkstra's Algorithm* [Dij59] is a more efficient solution to this problem. It iteratively explores the graph from the source node. The algorithm maintains a distance label $dist : V \to \mathbb{R}_+$ for the tentative distances from s to all other nodes and a predecessor label $pred : V \to V$ for the path retrieval. Additionally, a priority queue Q is used to order the recently explored nodes by ascending distance. Initially, $dist$ and $pred$ are unset for all nodes but the source ($dist[s] = 0$ and $pred[s] = s$, respectively). In each iteration, the Dijkstra search extracts the node $v \in Q$ with the smallest distance label $dist[v]$ and scans its outgoing edges. If $dist[v] + len(v, w) < dist[w]$ for an edge $(v, w) \in E$, the label of w is updated and w is re-inserted into the priority queue. Moreover, the predecessor of w is set to v. The algorithm terminates as soon as the target node t is extracted from Q and returns $dist[t]$. As $pred$ induces a reversed shortest-path tree from s, we can recursively follow the references from t to obtain the asked path between s and t.

Dijkstra's algorithm is classified as *label-setting* [BDG+16, AMO93] for non-negative edge weights. This means that once a node $v \in V$ is extracted from the queue, its label $dist[v]$ becomes final and will not be improved in any following iteration. After the queue extraction of the target node, $dist[t] = d(s, t)$ holds. The algorithm thus obtains correct results. From a theoretical point of view, the queue operations dominate the running time of Dijkstra's algorithm. Using a common *Binary Heap* results in a worst-case time of $\mathcal{O}((|V| + |E|) \cdot \log |V|)$. Although the usage of other queue implementations improves this bound in theory [BDG+16], these approaches often lack efficiency in practice.

Up to this point, we have assumed that the travel time along a road segment is constant throughout the day. This gives us a simple model and allows fast shortest path calculations. In reality, however, there is often a strong correlation between the daytime and the observed travel time. For example, there is significantly less traffic around midnight than during rush hours in the morning and afternoon. *Time-Dependent Route Planning* incorporates this aspect by using *travel time profiles* instead of scalar edge weights. These profiles are periodic and bound to a time interval $T = [0, p]$ with period $p \in \mathbb{R}_+$.

Definition 2.3 (Travel Time Profiles)**.** *Let $T = [0, p]$ be a time interval with period $p \in \mathbb{R}_+$. A time-dependent graph $G_T = (V, E, len)$ consists of a node set V, edges $E \subseteq V \times V$ and a weight function $len : (E \times T) \to \mathbb{R}_+$. The travel time profile $f_e : T \to \mathbb{R}_+$ of an edge $e \in E$ is given by $f_e(\tau) = len(e, \tau)$ for all $\tau \in T$.*

Hence, time-dependent graphs can be seen as a generalization of the graph model we have examined before. We can transform each time-independent graph into a time-dependent graph by using constant travel time functions for each edge. To approximate the travel time profiles by scalar edge weights, we can e. g. consider their lower and upper bounds.

Definition 2.4 (Lower/Upper-Bound Weights)**.** *Let $G_T = (V, E, len)$ be a time-dependent graph. The range of the travel time functions is limited by the lower bound $\underline{len} : E \to \mathbb{R}_+$ with*

$$\underline{len}(e) = \min_{\tau} len(e, \tau)$$

for all edges $e \in E$ and the upper bound $\overline{len} : E \to \mathbb{R}_+$ such that

$$\overline{len}(e) = \max_{\tau} len(e, \tau).$$

Compared to their time-independent counterparts, time-dependent graphs can utilize significantly more memory. In particular, we have to store complex travel time profiles instead of scalar edge weights. A compact memory layout is necessary to keep the storage overhead as low as possible. *Periodic Piecewise Linear Functions* are a commonly used solution to this problem. They are represented by a set of *breakpoints* $B \subseteq T \times \mathbb{R}_+$.

Definition 2.5 (Periodic Piecewise Linear Function, Breakpoints)**.** *Let $T = [0, p]$ be a time interval and let $f : T \to \mathbb{R}_+$ be a travel time profile. We call f a Periodic Piecewise Linear Function with breakpoints $B = \{b_0, b_1, \ldots, b_k\} \subseteq T \times \mathbb{R}_+$ ($b_i = (\tau_i, w_i)$), if the following conditions hold:*

- *Periodicity: $\tau_0 = 0$, $\tau_k = p$ and $w_0 = w_k$.*
- *Ordered by timestamp: $\tau_i < \tau_j$ for all $0 \leq i < j \leq k$.*
- *Equivalence: $f(\tau_i) = w_i$ for all $0 \leq i \leq k$ and the function curve is linear between all breakpoints.*

Periodic Piecewise Linear Functions are evaluated by performing a linear interpolation. To determine the function value $f(\tau)$ for an arbitrary timestamp $\tau \in T$, we first have to find the neighboring breakpoints $b_i, b_{i+1} \in B$ such that $\tau_i \leq \tau \leq \tau_{i+1}$. A binary search achieves this in $\mathcal{O}(\log |B|)$ time. The subsequent interpolation between b_i and b_{i+1} is a constant-time operation.

Extending Dijkstra's algorithm to time-dependent edge weights has first been studied by Cooke and Halsey [CH66]. It turns out that some minor modifications to the algorithm suffice. First of all, the query now also contains a departure timestamp τ^{dep}. Moreover, the travel time profile of each edge $e = (v, w) \in E$ must be evaluated at the arrival time $\tau^{dep} + dist[v]$ at v. Although the evaluation of the weights becomes more complex, the queue operations still dominate the overall running time. The performance losses caused by the time-dependent model are therefore manageable.

In addition to queries with a fixed departure τ^{dep}, we can also calculate the shortest distance for any departure time. These *profile queries* are particularly relevant for the preprocessing of speed-up techniques, as τ^{dep} is unknown in advance. Performing a separate Dijkstra run

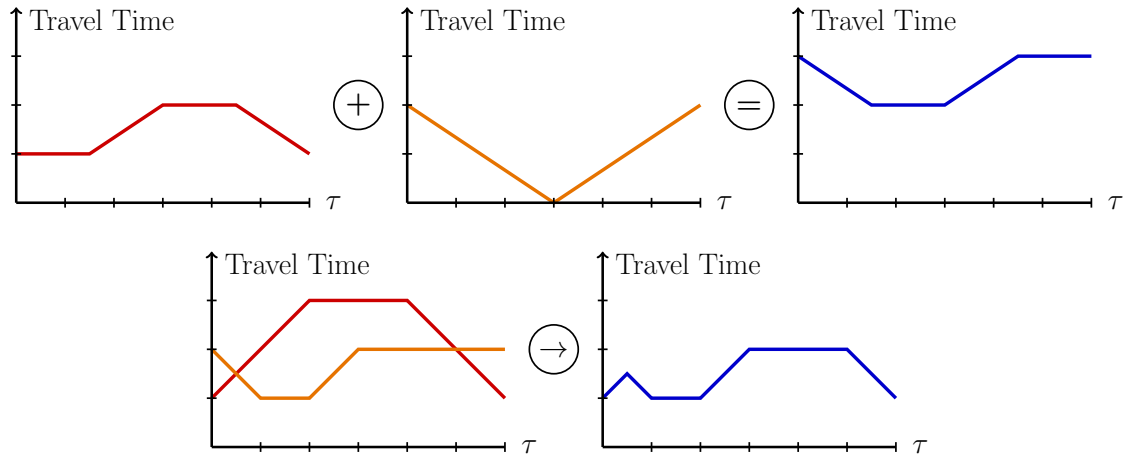


Figure 2.1: Examples of *linking* (above) and *merging* (below) travel time functions.

for each possible $\tau \in T$ is infeasible. To enable Dijkstra's algorithm to process profile queries, we maintain entire travel time functions as distance labels. The queue keys are derived from the lower-bound weights $\underline{len}$. Unfortunately, neither the composition of edges with distance labels (*linking*) nor the comparison between labels (*merging*) is straightforward for profile queries. In the time-independent setting, linking an edge weight $len((v,w))$ onto a distance label $dist[v]$ is done by a simple addition. After that, the merge operation updates the label at node w according to $\min\{dist[w], dist[v] + len(v,w)\}$. Both operations have a complexity of $\mathcal{O}(1)$ for scalar edge weights. Before discussing the complexity of linking and merging for profile queries, we define the general procedure.

Definition 2.6 (Linking and Merging Travel Time Profiles)**.** *Let $G_T = (V, E, len)$ be a time-dependent graph with time interval T. Furthermore, let $d_u, d_v : T \to \mathbb{R}_+$ be the corresponding distance labels of two nodes $u, v \in V$ and f_e be the travel time profile at edge $e = (u,v)$. Then, the linking step $\ell = d_u \oplus f_e$ composes d_u and f_e such that*

$$\ell(\tau) = d_u(\tau) + f_e(\tau + d_u(\tau))$$

for all $\tau \in T$. In the subsequent merging operation, the minimum $h = \min(d_v, \ell)$ is given by

$$h(\tau) = \min(d_v(\tau), \ell(\tau)).$$

The linking and merging operations on travel time functions are sketched in Figure 2.1. We omit further algorithmic details and refer to a work by Delling [Del09]. Both linking and merging run in linear time $\mathcal{O}(|B_f| + |B_g|)$, with respect to the number of *breakpoints* in the input functions f, g. Moreover, the linked function $f \oplus g$ has at most $|B_f| + |B_g|$ breakpoints. Due to additional intersection points in between, the merging step returns profiles with at most $2 \cdot (|B_f| + |B_g|)$ breakpoints. The complexity remains linear, though. Although these operations seem straightforward to implement, a practical implementation is non-trivial. First of all, both linking and merging can produce breakpoints at arbitrary positions. Therefore, we must not use integer values to represent travel time functions. Moreover, several edge cases related to inexact floating-point arithmetics must be addressed. For further details, we refer to the *CATCHUp* [SWZ21] implementation, which we have partially used for our experimental evaluation.

Profile queries are usually several orders of magnitude slower than a time-dependent Dijkstra search with a fixed departure time τ^{dep}. An important reason is that the queue

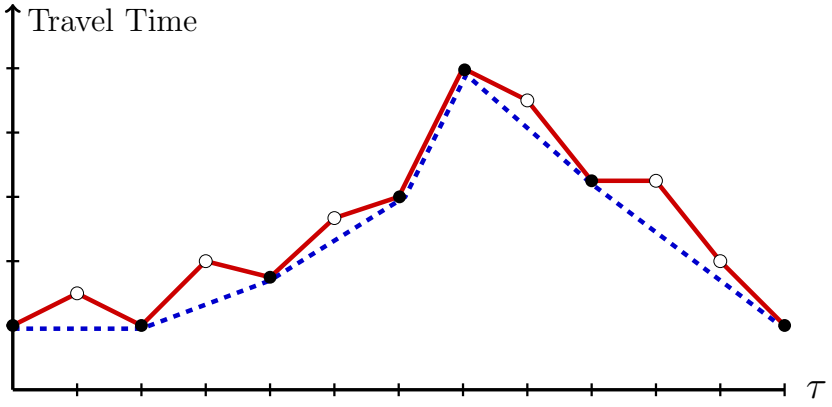


Figure 2.2: Lower-bound approximation of a complex travel time profile. By accepting a slight deviation from the original function, the number of breakpoints can be halved.

operations no longer dominate the running time. Instead, the linear-time linking and merging steps turn out to be the performance bottleneck. Unfortunately, the number of breakpoints at the distance labels increases with every step. With a growing distance from the source node, the performance of these procedures degrades continuously.

Another issue is that Dijkstra profile queries are not classified as *label-setting*, but *label-correcting* [AMO93]. Using a lower bound of the distance label as a queue key does not necessarily imply that a node is settled after its queue extraction. Instead, a node has to be re-inserted whenever its distance label is improved at *any* timestamp $\tau \in T$. This circumstance increases the total number of linking and merging steps and thus leads to further performance losses. We can only consider a node $v \in V$ settled if its upper-bound distance $\max_{\tau \in T} dist[v](\tau)$ falls below the minimum queue key. Therefore, we use the upper-bound distance to the target node as a threshold to terminate the query as early as possible.

There has been some research on the maximal complexity of travel time functions. A conjecture by Dean [Dea04] states that the complexity may grow superpolynomially. Later, Foschini et al. [FHS11] have proven this conjecture. For an aggregated number of k breakpoints over all edge weight functions, the maximal complexity of a profile is bounded by $k \cdot n^{\mathcal{O}(\log(n))}$ where $n = |V|$. If k is polynomial in the number of nodes, this formula simplifies to $n^{\Theta(\log n)}$. Although this does not resemble the average case, the rapid growth in complexity of the distance labels still poses a major performance issue.

To facilitate the linking and merging operations, we apply approximation techniques to the travel time functions. At the cost of inexact results, profiles are approximated whenever the number of breakpoints exceeds a given threshold. *Douglas-Peucker* [DP73] and *Imai-Iri* [II88] are two of the most popular algorithms to approximate complex functions. These methods reduce the number of breakpoints by tolerating a slight deviation $(1 + \epsilon)$ from the original function. The error tolerance $\epsilon > 0$ is given as an additional parameter. Depending on the use case, we can also provide an approximated lower or upper-bound function, as depicted in Figure 2.2.

We must also address another modeling issue of time-dependent edges. The *First-In-First-Out* (FIFO) property states that departing at a later time does not result in an earlier arrival time. This implies that the slope of a travel time function not fall below -1 at any time.

Definition 2.7 (FIFO Property for Travel Time Functions)**.** *Let $f : T \to \mathbb{R}_+$ be a travel time function in the interval $T = [0, p]$. The FIFO property is satisfied for f if*

$$f(x) + x \leq f(y) + y$$

holds for any timestamps $0 \leq x < y \leq p$.

Allowing non-FIFO travel time profiles has severe impacts on the performance of our algorithms. In particular, the time-dependent SPP gets $\mathcal{NP}$-hard and might even be unsolvable by Dijkstra's algorithm [Ord89]. Therefore, it is crucial to assert the FIFO property after each linking, merging, and approximation step.

We can conclude that time-dependent route planning is associated with several issues that do not exist in the time-independent model. However, addressing these problems allows us to use a more realistic model of the underlying road network.

2.3 Algorithms for Route Planning

In the past two decades, several speed-up techniques have been proposed to accelerate Dijkstra's algorithm. A common feature of these methods is to reduce the *search space size*, i. e. the number of visited nodes during a query. We introduce the most relevant research results for our work in this section. Section 2.3.1 describes the *goal-directed* search, also known as the *A* Algorithm* [HNR68]. In Section 2.3.2, we discuss two *hierarchical* speed-up techniques. *Contraction Hierarchies* [GSSD08] usually outperform Dijkstra by several orders of magnitude. More recently, an extension has been proposed to incorporate frequent traffic changes [DSW14]. Finally, an approach to combine A* and hierarchical techniques is presented in Section 2.3.3. We describe all algorithms in their original time-independent graph model.

2.3.1 The A* Algorithm

Extending Dijkstra's algorithm with a *goal-directed* heuristic can help to skip irrelevant nodes during the query phase. The heuristic is applied to estimate the remaining distances to the target node. Using the sum of distance labels and heuristic estimates as queue keys changes the processing order of the nodes. In the case of perfect estimates, the search will only visit nodes along the shortest path.

Goal-directed heuristics have originally been used in the area of artificial intelligence [GH05]. The A* algorithm [HNR68] is widely applied to reduce the effort of finding admissible solutions in huge data sets. Pohl [Poh71] has studied the relation between Dijkstra's algorithm and A*. For shortest-path algorithms, the heuristic is given as a *potential function* $\pi_t : V \to \mathbb{R}_+$ which estimates the distances to a fixed target node $t \in V$. The actual A* algorithm can be considered as a generalization of Dijkstra. As outlined in Algorithm 2.1, the heuristic π_t is applied to change the order in which the nodes are processed. In particular, the queue key of each node $v \in V$ is given by $dist[v] + \pi_t[v]$ (lines 12–15). All other steps are equivalent to Dijkstra. Therefore, using the *Zero-Potential* $\pi_t(\cdot) = 0$ restores the original search order. Aside from that, modifying the A* algorithm to handle time-dependent edge weights is analogously done as discussed in Section 2.2.

To ensure the correctness of the A* algorithm, some conditions must hold for the potential functions. We define the corresponding properties as *feasibility* [GH05] and *lower-bound* property.

Algorithm 2.1: GOAL-DIRECTED SEARCH (A* ALGORITHM)

Input: Graph $G = (V, E, len)$, Potential Function $\pi\colon V \to \mathbb{R}_+$
Input: Source node $s \in V$, target node $t \in V$
Data: Priority Queue Q
Output: Shortest distance $\mathsf{dist}(t)$ from s to t (∞ if not reachable)

```
1  forall v ∈ V do
2      dist(v) ← ∞
3  Q.INSERT(s, 0)
4  dist(s) ← 0
5  while Q is not empty do
6      u ← Q.DELETEMIN()
7      if u = t then
8          return dist(t)
9      forall (u, v) ∈ E do
10         if dist(u) + len((u, v)) < dist(v) then
11             dist(v) ← dist(u) + len((u, v))
12             if Q.CONTAINS(v) then
13                 Q.DECREASEKEY(v, dist(v) + π(v))
14             else
15                 Q.INSERT(v, dist(v) + π(v))
```

Definition 2.8 (Feasibility Property of A* Potentials [GH05])**.** *Consider the potential function $\pi_t : V \to \mathbb{R}_+$ for a given graph $G = (V, E, len)$. We call π_t feasible, if*

$$len((u, v)) + \pi_t(v) - \pi_t(u) \geq 0$$

holds for all edges $(u, v) \in E$.

Definition 2.9 (Lower-Bound Property of A* Potentials)**.** *The potential π_t satisfies the lower-bound property if*

$$\pi_t(v) \leq d(v, t),$$

i. e. the estimates do not exceed the actual shortest distance $d(v, t)$ between any node $v \in V$ and the target node.

Pohl [Poh71] has shown that running A* with a *feasible* potential π_t corresponds to a Dijkstra search with *reduced* edge weights

$$len'((u, v)) = len((u, v)) + \pi_t(v) - \pi_t(u)$$

for all $(u, v) \in E$. The *feasibility* of π_t implies a non-negative weight function len' and thus preserves the correctness of Dijkstra's algorithm. If the heuristic provides exact estimates, all edges along the shortest path have a reduced cost of zero. Moreover, we show that *feasibility* is a sufficient criterion for the *lower-bound* property if $\pi_t(t) = 0$ (see Theorem 2.11). An inductive argument is applied to prove this relation. In the induction step, we exploit the *sub-path* property. Consistent with *Bellman's Principle of Optimality* [Bel57], subsequences of shortest paths are also shortest paths.

Lemma 2.10 (Sub-path property of shortest paths)**.** *Let $G = (V, E, len)$ be a graph with non-negative edge weights $len : E \to \mathbb{R}_+$. Moreover, let $P = (v_1, \ldots, v_k)$ be the shortest path between $v_1, v_k \in V$. For any $1 \leq i < j \leq k$, the subsequence $P' = (v_i, \ldots, v_j)$ of P is a shortest path between v_i and v_j.*

Suppose for contradiction that the shortest path between some v_i, v_j deviates from P'. Then, we can replace the subsequence P' in P with the shorter path. However, this is a contradiction to the assumption that P is the shortest path. Next, we prove the relation between the A* correctness properties.

Theorem 2.11. *Let $\pi_t : V \to \mathbb{R}_+$ be a feasible potential with $\pi_t(t) = 0$. Then π_t also satisfies the lower-bound property, i. e. $\pi_v(t) \leq d(v,t)$ for all $v \in V$.*

Proof. We give a proof by induction on the depth k in the (reversed) shortest path tree induced by a fixed target node $t \in V$.

Initial Case: $k = 0$ is only satisfied for the target node t. As $d(v,v) = 0$ for all $v \in V$, we require $\pi_t(t) = 0$ as given by assumption.

Induction Step: For a fixed $k \geq 0$, we show that if $\pi_t(v) \leq d(v,t)$ holds for all nodes v with tree depth k, it also holds for all nodes u with depth $k+1$.

Let $u \in V$ be an arbitrary node at depth $k+1$ and let $(u, v, \ldots, t)$ be the corresponding path in the tree. The sub-path property (Lemma 2.10) of shortest paths implies $d(u,v) = len(u,v)$ as well as $d(u,t) = d(u,v) + d(v,t)$. Moreover, $(v, \ldots, t)$ is a shortest path from v to t with length k. Hence, $\pi_t(v) \leq d(v,t)$ follows by induction hypothesis. The feasibility property

$$len(u,v) + \pi_t(v) - \pi_t(u) \geq 0$$

can thus be rewritten as

$$\begin{aligned} \pi_t(u) &\leq len(u,v) + \pi_t(v) \\ &\leq d(u,v) + d(v,t) \\ &= d(u,t) \end{aligned}$$

which implies that all feasible potentials π_t with $\pi_t(t) = 0$ also satisfy the lower-bound property. □

We have already mentioned that *feasible* potentials guarantee the correctness of the A* algorithm. However, it is also possible to obtain correct results from heuristics that only satisfy the *lower-bound* property. Although some reduced edge weights may be smaller than zero, we can terminate the A* algorithm upon the target node extraction from the queue.

Theorem 2.12. *The A* algorithm provides correct results if the used potential function π_t provides lower-bound estimates for all nodes and $\pi_t(t) = 0$.*

Proof. It suffices to show that the distance labels of all nodes along the shortest path are final when the queue extraction of t terminates the algorithm. We give a proof by contradiction.

Let $tent_t = dist[t]$ be the tentative distance after the first queue extraction of t. Suppose we could improve $dist[t]$ in a following iteration. Then, at least one node $v \in V$ along the shortest path between s and t must remain in the queue or has not been discovered yet. Hence, $dist[v] + \pi_t(v) \geq tent_t$. We also require $dist[v] + d(v,t) < tent_t$ as we assume that the target distance can still be improved. This implies

$$dist[v] + \pi_t(v) \geq tent_t > dist[v] + d(v,t),$$

which can be simplified to $\pi_t(v) > d(v,t)$. As this contradicts to the lower-bound property of π_t, such a node v does not exist. Therefore, $dist[t] = d(s,t)$ holds at the termination of the A* algorithm. □

Most A* heuristics found in literature (e. g. [GH05, DW09, SZ21]) are *feasible*. Despite that, Theorem 2.12 proves that the A* algorithm also obtains correct results for potentials which satisfy the *lower-bound* property, but not *feasibility*. In this case, however, negative reduced weights are possible. Some nodes might therefore be visited several times. From a theoretical perspective, the worst-case running time of these *label-correcting* shortest path algorithms is significantly higher compared to *label-setting* algorithms [AMO93]. Fortunately, the observed performance impacts are often manageable. As we will see later in Chapter 4, using such heuristics may even improve the overall running time of the A* algorithm. An important reason is that *feasible* potentials constrain the algorithmic design space and prohibit some performance optimizations. As a side note, it is also possible to obtain correct results with potentials violating the *lower-bound* property. However, such modifications like *Anytime A** [LGT03] cause significant performance losses and are thus unlikely able to compete with the A* potentials described in the following chapters.

2.3.2 Customizable Contraction Hierarchies

Hierarchical speed-up techniques such as *Contraction Hierarchies* (CH) [GSSD08] augment the graph by adding additional *shortcut* edges during preprocessing. In the query phase, these shortcuts can be used to skip unimportant nodes and thus reduce the search space to a fraction of the original graph. To transform the result into a valid route, we translate the shortcuts back to their corresponding set of original edges in the input graph. Initially, the function $rank : V \to \{1, \dots, |V|\}$ is determined to order the nodes by their *importance*. For instance, nodes along highway segments are more important than dead ends in residential areas. Throughout this work, we depict the most important nodes at the top and the least important nodes at the bottom. We emphasize that choosing a suitable node order is crucial to gain significant speed-ups over Dijkstra's algorithm. Further details on obtaining good orders are found in [GSSD08].

During the preprocessing step, the nodes are iteratively *contracted* in ascending order. By contracting a node $v \in V$, additional *shortcut* edges are created between all incoming and outgoing *upward* neighbors of v. As shown in Figure 2.3, the contraction of the node v produces an additional shortcut edge (u, m). Its weight corresponds to the original weights along the path (u, v, m). The contraction of all nodes results in an augmented graph $G' = (V, E \cup E')$ with the shortcut edges E'. Multi-edges in G' are aggregated to a single edge with the smallest weight. For the subsequent query stage, we construct a *forward* (*upward*) graph

$$\overrightarrow{G} = (V,\ \{(v, w) : (v, w) \in E \cup E' \wedge rank(v) < rank(w)\})$$

as well as a *backward* (*downward*) graph

$$\overleftarrow{G} = (V,\ \{(w, v) : (v, w) \in E \cup E' \wedge rank(v) > rank(w)\})$$

with reversed edge directions. We denote the edges in $\overrightarrow{G}$ as *forward* or *upward* edges in the following. Analogously, the edges of $\overleftarrow{G}$ are classified as *backward* or *downward* edges.

The query phase exploits the added shortcuts to skip unimportant parts of the graph. A *bidirectional* variant of Dijkstra's algorithm solves the given query. The forward search explores $\overrightarrow{G}$ from the source node s. Similarly, the backward search operates on the reversed graph $\overleftarrow{G}$ and begins at the target node t. Both search directions only consider edges from less to more important nodes. When the searches meet at an intermediate node $m \in V$, we have found an *up-down path* between source and target (see Figure 2.3). However, the first meeting node found is not necessarily part of the shortest path. Both searches must continue until their respective minimum queue keys exceed the tentative distance between s

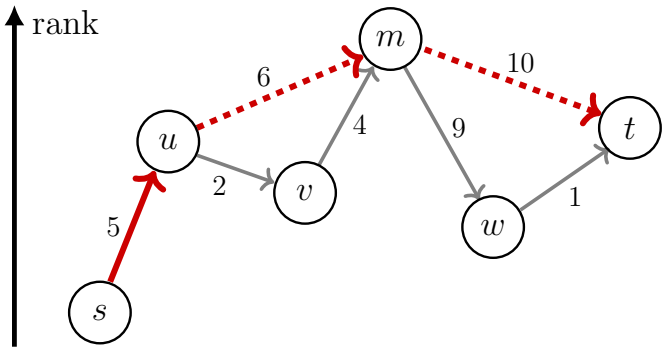


Figure 2.3: Visualization of the CH contraction and query step. Contracting the nodes v, w augments the graph by two shortcut edges (dashed). During the subsequent query from s to t, several original edges can be skipped by using these shortcuts. The thick edges mark the shortest *up-down path* with meeting node m.

and t. Although this seems rather conservative, it is necessary to ensure the correctness of the algorithm. It is guaranteed that the shortest path between any pair of nodes is found via an up-down path [GSSD08].

Contraction Hierarchies can be applied to solve queries on continental-sized graphs within milliseconds [GSSD08]. However, finding a good node order can take several hours, depending on the size of the graph and the applied algorithm. Real-world applications must regularly apply traffic updates within seconds. Re-running the preprocessing step for each traffic update is thus not applicable. *Customizable Contraction Hierarchies* (CCH) [DSW14] has been proposed to solve this issue. In this speed-up technique, the CH preprocessing is split into two sub-phases. The *metric-independent* stage obtains a node order and augments the given graph with shortcuts accordingly. We only run this step whenever the topology of the road network changes. The *metric-dependent* phase (also denoted as *customization*) applies the updated traffic data to the augmented CCH graph. In contrast to the metric-independent stage, the customization is highly optimized and runs within seconds [DSW14]. Hence, CCH is more suitable for usage in real-world applications. In the following, we briefly describe the three CCH stages (preprocessing, customization, queries).

Preprocessing. Metric-independent node orders can be calculated with a *nested dissection* (ND) [BCRW16, Geo73] algorithm. Initially, a small separator set $S \subseteq V$ is selected to split the remaining graph $G \setminus S$ into two disjoint partitions of balanced size. While the nodes of the separator set are classified as the most important in G, recursion is applied to determine the order within the partitioned subgraphs. Further details, including graph partitioning algorithms, can be found in [DSW14]. The obtained node order is then used to contract the nodes iteratively and create additional shortcut edges. Due to *metric-independent* calculations, the resulting shortcut graph is *unweighted*. The direction of the edges is also omitted at this point. As a result, augmented CCH graphs are usually larger than CH graphs [DSW14]. The main reason is that the CH algorithm is able to conduct metric-based optimizations [GSSD08]. Nevertheless, the resulting performance impact on the query times is negligible. Furthermore, computing a ND-based node order allows us to extract an *elimination tree* of low height from the augmented CCH graph. The root of this tree corresponds to the most important node in the graph. All other nodes are child nodes of their respective lowest-ranked upward neighbor. An example of this is given in Figure 2.4. We will explain later how to exploit the elimination tree in the query. Besides that, we must be aware that obtaining an optimal ND-based node order as well as constructing elimination trees with minimal height is $\mathcal{NP}$-hard [DSW14, Pot88]. The preprocessing step thus relies on heuristic approaches which provide good results in practice.

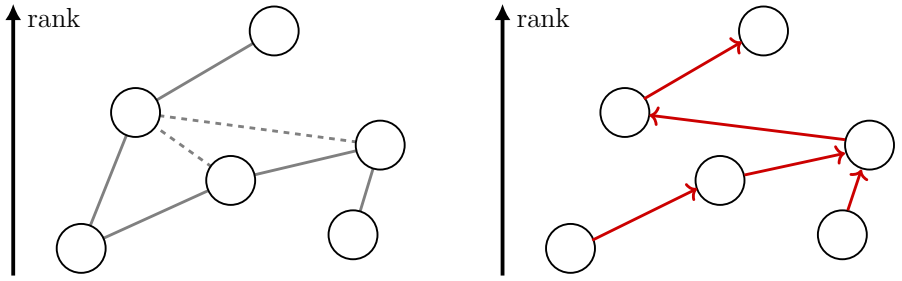


Figure 2.4: Constructing an elimination tree (right) from an unweighted graph with a given node order (left). Added shortcut edges are dashed.

Customization. During the subsequent customization phase, the shortcut weights are initialized according to the given metric $len : E \to \mathbb{R}_+$. In contrast to the metric-independent preprocessing step, the direction of the edges is relevant from now on. Initially, an infinite weight is assigned to all shortcut edges. After that, the algorithm enumerates all *lower triangles* $\{u, v, w\}$ $(rank(u) < rank(v) < rank(w))$ in the augmented CCH graph. It is checked whether the weight at $(v, w) \in E \cup E'$ can be improved by traversing the path (v, u, w). This procedure is repeated for the downward edge (w, v). Processing the triangles in ascending node order is necessary to ensure the correctness of the algorithm. In particular, the weights of intermediate shortcuts must be initialized when needed for further calculations. After completion of the triangle enumeration, it is possible to apply further metric-dependent optimizations. Similar to the CH preprocessing, we can remove unnecessary shortcut edges to reduce the size of the graph (*perfect customization*). Further details are explained in [DSW14].

Query. The CCH query algorithm applies a bidirectional search to find the shortest path between a given source $s \in V$ and a target node t. Each search direction employs an *elimination tree query* instead of Dijkstra's algorithm. The forward search starts exploring the outgoing edges of the source node in $\overrightarrow{G}$ and recursively follows the parent references from s up to the tree root. Analogously, the backward search starts at t and proceeds on $\overleftarrow{G}$. Bauer et al. [BCRW16] have proven the correctness of the bidirectional elimination tree query. Compared to a Dijkstra-based CH search, the entire upward (reversed downward) search space of s (t) must be explored. Fortunately, we can do this without maintaining the tentative distances in a priority queue. Therefore, elimination tree queries perform well on *long-range* queries, where large parts of the graph are scanned. Still, both algorithms yield query times within milliseconds [DSW14]. Lastly, we point out that only metric-independent node orders provide theoretical boundaries on the height of the elimination tree. We cannot use elimination tree queries in Contraction Hierarchies for this reason.

Both CH and CCH are extendable to time-dependent graphs [BGSV13, SWZ21]. However, their preprocessing times are significantly higher. This is due to the additional effort of processing travel time functions instead of scalar edge weights. Moreover, queries take an order of magnitude longer [SWZ21]. We must also consider that designing a bidirectional search with an initially unknown arrival time at the target node is non-trivial. This poses additional challenges for the implementation of the query algorithm. Time-dependent Contraction Hierarchies have been studied in [BGSV13]. More recently, Strasser et al. [SWZ21] have reworked the CCH algorithm to solve time-dependent queries. As expected, the time-dependent customization takes much longer than in the time-independent scenario. The authors have reported times of up to 15 minutes on continental-sized graphs with several million nodes and edges [SWZ21]. For some applications, the customization takes too long to incorporate traffic updates frequently. Fortunately, it is possible to

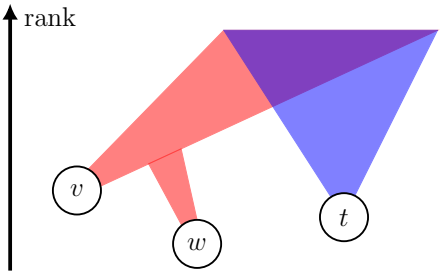


Figure 2.5: Lazy RPHAST with multiple source nodes. As the forward search spaces of v and w overlap with each other, only few computations are required to obtain the distance between w to t.

decouple the CCH data from the graph used for the queries. We provide more details in the next section.

2.3.3 Lazy RPHAST: CCH-Potentials

Several speed-up techniques augment the graph with additional shortcut edges to accelerate the queries. We have mentioned that this can be problematic if the edge weights are subject to frequent changes, especially on time-dependent graphs. *CH-Potentials* [SZ21] are a solution to this problem. They combine hierarchical speed-up techniques and the A* algorithm. The authors also describe a general scheme of decoupling the speed-up technique from the underlying graph structure. *CH-Potentials* provide fast and tight heuristic estimates which are derived from a CH or CCH. The queries are solved with the A* algorithm that operates on the unmodified input graph. Although the A* algorithm is generally slower than hierarchical methods, we can apply traffic updates without any additional effort. The correctness properties of the applied A* heuristics must hold, though.

Computing valid potential estimates for all nodes $v \in V$ corresponds to solving the *All-To-One* SPP on the augmented shortcut graph. More precisely, we are interested in computing estimates $\pi_t(v)$ of the shortest distance $d(v, t)$ between all nodes v and the target node t. *PHAST* [DGNW13] is a CH extension and an efficient solution to the All-To-One problem. The algorithm is divided into two phases. During the first stage, the search space of t is explored on the *reversed downward* graph $\overleftarrow{G}$. In the second step, the entire forward graph $\overrightarrow{G}$ is inspected in reversed order, i. e. starting with the most important node. The previously obtained distance labels are used to combine the edge weights of $\overrightarrow{G}$ and $\overleftarrow{G}$. This is necessary to ensure that all *up-down paths* (see Section 2.3.2) are considered. By construction of the node order, the backward distance of the highest-ranked node must be set after the exploration of $\overleftarrow{G}$. Then, processing the nodes in reverse order guarantees that all upward neighbors of the currently scanned node have a final distance label.

Depending on the accuracy of the potential estimates, the A* search is often restricted to small parts of the graph. Therefore, PHAST might perform unnecessary calculations for many nodes. To provide potentials for a smaller set of nodes, we are interested in solving the *Many-To-One* SPP (multiple sources, one target node) efficiently. Restricted PHAST (*RPHAST*) [DGW11] has been proposed as a modification of PHAST and is more suitable for the given problem. For a given set of source nodes, the algorithm applies an additional *selection* step. It extracts all nodes and edges in the forward graph $\overrightarrow{G}$ which are reachable from these nodes. The second stage of PHAST is then conducted on the restricted graph.

Although RPHAST solves the Many-To-One SPP efficiently, one issue remains to be solved to make it suitable for providing A* potentials. While the target node is known in advance,

Algorithm 2.2: Lazy RPHAST: CCH-Potential

Data: Forward Graph $\overrightarrow{G} = (V, \overrightarrow{E}, \overrightarrow{len})$, Backward Graph $\overleftarrow{G} = (V, \overleftarrow{E}, \overleftarrow{len})$
Data: Elimination Tree `tree`: $V \to V$
Data: Tentative distances $B(\cdot)$ to the target node, initially ∞
Data: Potential estimates $D(\cdot)$ for each node $v \in V$, initially $\bot$

```
1  function init(t):
2      u ← t
3      while u ≠ ⊥ do
           // init backward-upward search space of t
4          forall e ← (u, v) ∈ E← do
5              B[v] ← min{B[v], B[u] + len←(e)}
6          u ← tree(u)
7  function potential(v):
8      u ← v, stack ← Stack()
       // determine non-explored forward-upward search space
9      while u ≠ ⊥ && D[u] = ⊥ do
10         stack.push(u), u ← tree(u)
       // descend search space in reverse order
11     while stack is not empty do
12         u ← stack.pop()
13         forall e ← (u, x) ∈ E→ do
14             B[u] ← min{B[u], B[x] + len→(e)}
15         D[u] ← B[u]
16     return D[v]
```

the sources are given one after another by the A* algorithm. In addition, we must provide a target distance estimate for the current node before the A* algorithm proceeds with the next node. To solve this problem, the second RPHAST step is executed *lazily* [SZ21]. As depicted in Figure 2.5, each potential request extends the previously explored search space in $\overrightarrow{G}$. This avoids redundant computations and enables processing the sources one after another. The entire *CH-Potential* [SZ21] algorithm is based on *Lazy RPHAST*. As we only consider customizable speed-up techniques in the following, we also denote this approach as *CCH-Potential*.

Lazy RPHAST is outlined in Algorithm 2.2. During the *Initialization* step, the upward search space of the target node t is explored on the reversed backward graph $\overleftarrow{G}$. As the shortest distances in a CCH can only be captured by considering *up-down paths*, all distance labels are tentative and become final once the respective nodes are scanned in forward direction. The second stage is invoked whenever the potential estimate for a node $v \in V$ is requested by the A* algorithm. We ascend in the elimination tree until we either reach the root or find a node that has been settled in a previous request (lines 8–10). Storing the parent nodes along this path in a stack allows us to descend back to v in reversed order. After exploring the outgoing edges of each node u in the stack, the distance estimate $\pi_t(u) = D[u]$ becomes final (lines 11–15). As we keep the results of previous requests in $D[\cdot]$, the running time for a single potential request varies. While the earliest queries likely have to explore their entire upward search space, some intermediate requests

might require no work at all. The more potential requests by the A* algorithm, the higher the probability that large parts of the CCH graph are already explored. Unlike PHAST and RPHAST, Lazy RPHAST thus performs efficiently for both small and large sets of source nodes.

Compared to a Dijkstra search, *CCH-Potentials* can reduce the search space size of a query by around three orders of magnitude [SZ21]. Although solving queries with a CH or CCH yields even faster response times, we can exploit the flexibility of A* to incorporate further modeling details (e. g. time-dependence) more efficiently. We have already discussed that applying a time-dependent CCH leads to high memory consumption and a slow customization phase. In contrast, using CCH-Potentials completely avoids the need for a time-dependent speed-up technique. Instead, it suffices to obtain a *feasible* A* heuristic from the lower-bound weights $\underline{len}$ of the actual travel time functions. Further efficient algorithms for time-dependent route planning emerge from this observation. We will discuss these in more detail in Chapter 4.

3. Cooperative Route Planning

This chapter covers the basic concepts of *cooperative route planning*. Possible advantages over existing *selfish* routing algorithms are discussed in Section 3.1. In Section 3.2, we describe our proposed model of cooperative route planning. Finally, Section 3.3 points out some issues related to applying speed-up techniques to the given model.

3.1 Basic Concept

In most route planning applications, the incoming routing requests are solved independently from each other. These applications propose the currently best-known route to the users without considering other requests. Although this *selfish* approach is comparatively easy to implement, it does not provide an accurate prediction of future traffic conditions. In particular, a set of similar requests is likely being routed along the same roads. This might lead to additional and avoidable traffic jams. It would be smarter to apply a more sophisticated approach that distributes traffic evenly across the road network. We give two real-world examples why cooperation between road users can be advantageous.

Firstly, selfish routing systems often suggest alternative routes during traffic disruptions (e. g. accidents or roadworks). Occasionally, the routing application proposes the same alternative route to all drivers. As a result, the detour congests while the traffic along the main route eases. The other way around, it is also possible that all road users attempt to traverse the main roads and ignore all detours. In a cooperative environment, however, the traffic would be distributed evenly.

The other example is related to major events, such as sports matches. As many road users aim to arrive simultaneously, the main roads towards the venue are likely congested. Extending the arrival time of road users over a longer time window and evenly distributing traffic over the access roads would be reasonable countermeasures. Both examples demonstrate that predicting and balancing future traffic flows can help to reduce congestion.

In addition to these real-world observations, theoretical studies show that selfish routing negatively affects the aggregated travel time of all road users. In 1952, John Wardrop introduced two famous principles related to traffic assignment [War52]. These principles distinguish between a *user-optimal* and a *system-optimal* environment [KZT20]. The first principle describes a *user equilibrium* state in which the selfishly taken route of each user takes less time than any alternative. Conversely, this also means that a user will immediately choose an alternative route if it promises an earlier arrival time at the desired

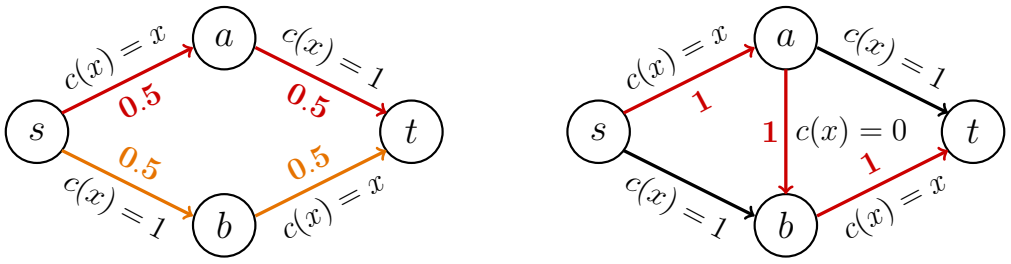


Figure 3.1: Braess paradox: An additional road segment can slow down the entire traffic, even if its usage is free of cost [RT02]. In this example, one unit of traffic has to be routed from s to t. If all drivers selfishly minimize their cost, we will obtain equilibrium states with costs of 1.5 (left) and 2 (right).

destination. According to the second principle, the average travel time of each user is minimized in the *social optimum* state. Achieving this system-optimal state requires full cooperation between all road users. Therefore, user equilibrium and social optimum state usually differ from each other. Although some assigned routes might be sub-optimal in the latter setting, there are cases in which the social optimum can be advantageous for all road users. We present a well-known paradox to underline this claim.

The *Braess paradox* [BNW05] has been published by the mathematician Dietrich Braess in 1968. It demonstrates that the user equilibrium state can worsen when considering an additional road segment. An example taken from [RT02] is visualized in Figure 3.1. In this example, the cost of traversing an edge is given by the function $c(x)$. The costs (e. g. congestion or traffic delays) depend on the road conditions and the traffic load x. Suppose that we are routing one unit of traffic from the leftmost node s to the rightmost node t. In the first scenario, the target node is reachable via the paths (s, a, t) and (s, b, t). Traversing these paths is associated with the cost $c(x) = 1 + x$. Both the *user-optimal* and *system-optimal* environment yield an equilibrium state in which the traffic is distributed evenly over the lower and upper route. Hence, the cost for all road users is 1.5 in both cases. Next, the road network is expanded by an edge (a, b) that can be traversed free of cost. Even though this additional edge provides a possibility to bypass the most expensive edges (s, b) and (a, t), we suddenly observe an equilibrium cost of 2 in the *user-optimal* setting. An important reason is that the previous equilibrium state is no longer valid. In this state, a single driver at node a could improve the remaining travel time by taking the detour via the new edge. Assuming that all drivers selfishly follow this detour, the edge (a, t) will not be used at all. Following the same argumentation, the edge (s, b) will not be used in an equilibrium state, either. Therefore, all traffic is routed along the path (s, a, b, t) with a total cost of 2. In the system-optimum setting, no driver traverses the new road at all. The system-optimum equilibrium cost remains unchanged at 1.5.

Both practical examples and theoretical results underline that cooperation between road users can reduce the aggregated travel time. We refer to *cooperative route planning* as a centralized approach to predict future traffic conditions and avoid congested roads at an early stage. Moreover, we study a cooperative routing model in which a central authority assigns routes in real-time and considers the impact of previously made queries in the current request. To predict future traffic conditions, each route assignment increases the expected travel times along the affected road segments. This allows us to assign similar requests to different routes and reduce the congestion of a few main roads. We point out that this is not a solution to unforeseen events, such as temporary road closures. However, routing fewer vehicles along the same road reduces the risk of both traffic jams and accidents.

Cooperative route planning can thus be a useful measure to reduce congestion in urban areas.

Unfortunately, applying traffic assignment algorithms in real-time route planning applications is infeasible for several reasons. Contrary to our problem setting, the Traffic Assignment Problem requires all origin-destination pairs to be known in advance. Moreover, adapting to this scenario by continuously recalculating an equilibrium state for the currently known requests is not practicable. The performance of traffic assignment algorithms is limited, particularly on large graph instances. Considering time-dependence in the assignment step would degrade the performance even further. Hence, we restrict our model to provide routes in real-time, considering *all available* information derived from *previous* requests.

In general, an *on-line* distribution of the given requests over the road network will not yield a *social optimum* state. This would require an *off-line* algorithm that is aware of the entire input sequence in advance. Nonetheless, the studied model enables cooperation between the road users in real-time. In particular, we enforce cooperation by predicting future traffic flows based on previous requests. Our approach requires further adjustments to be usable in real-world applications, though. For example, subsequent route adjustments could incorporate additional traffic information. However, we stress that our main focus is on designing efficient shortest-path algorithms for cooperative route planning. For this, it is sufficient to use a simple model with one-time route assignments. In the next section, we will formally describe our considered model. We also present a graph structure that enables predictions of future traffic conditions.

3.2 Graph Model

First, we present a graph structure that captures future traffic flows. For this purpose, the route information of all previous requests must be stored. Following modeling techniques of traffic assignment algorithms, we store the traffic loads along each road segment [BSW19b]. To account for time-dependent traffic flows, we maintain k *buckets* for each edge. These buckets cover equally-sized time intervals over the entire period T and store the number of vehicles passing the edge in the respective time frame. Additionally, we define the *capacity* of each edge as the number of vehicles that can traverse the segment during each interval without causing significant congestion. In the following, we denote the resulting graph structure as *Capacity-Graph*.

Definition 3.1 (Capacity-Graph)**.** *Let $T = [0, p]$ be a time interval with period $p \in \mathbb{R}_+$. Moreover, let $k \in \mathbb{N}$ and let $G = (V, E)$ be a graph representing the topology of the road network. The Capacity-Graph $G_{T,k} = (V, E, C, C_{max})$ is a tuple of*

- *the set of nodes V and edges E given by G,*
- *the function $C : (E \times [0, k]) \to \mathbb{N}$ representing the edge buckets. The entry $C(e, i)$ stores the number of vehicles passing the edge $e \in E$ within the interval $\left[i \cdot \frac{p}{k}, (i + 1) \cdot \frac{p}{k}\right)$,*
- *and the capacity function $C_{max} : E \to \mathbb{N}$.*

The parameter k in $G_{T,k}$ controls the degree of time-dependence. Using $k = 1$ corresponds to the time-independent case and only considers the total traffic on each road segment. In contrast, a large k offers a more detailed overview of current and future traffic conditions. Depending on k, we must be aware that maintaining $\mathcal{O}(k \cdot |E|)$ edge buckets can lead to considerable memory consumption. Apart from that, k also affects the capacity C_{max} of each edge bucket. Shortening the time intervals implies that fewer vehicles can traverse the

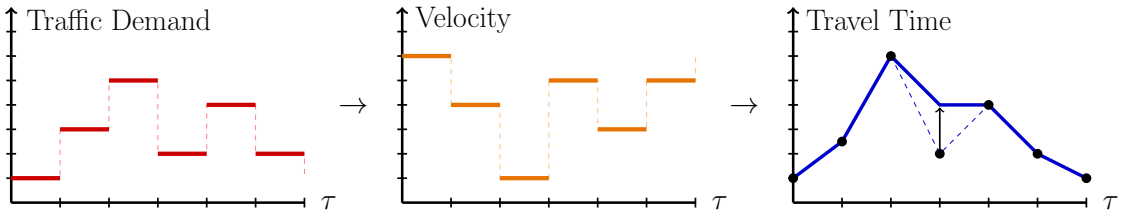


Figure 3.2: Two-step procedure to obtain travel time profiles from traffic loads stored in the edge buckets. After deriving velocities from the traffic loads, we convert the resulting speed profile into the desired travel time profile. Compared to directly converting traffic loads into travel times, the breakpoints are arranged to be FIFO-consistent.

road during each interval. The capacity of each edge also depends on the road type. While highways usually enable high traffic loads before congesting, many roads in residential areas are not designated for transit traffic. This has to be considered in the model, too.

Next, we discuss how to derive travel times from traffic loads. This problem has been studied in the context of traffic assignment [BSW19b]. A commonly used solution is the *Bureau of Public Roads* (BPR) function [BoPR64]. This function requires three parameters: the initial *free-flow* time t_0 along the road segment as well as the traffic load c and the capacity c_{max} of the edge. Then, the travel time is determined according to

$$t(t_0, c, c_{max}) = t_0 * \left(1 + \alpha * \left(\frac{c}{c_{max}}\right)^{\beta}\right), \tag{3.1}$$

where α and β are additional calibration parameters to take into account further modeling aspects. Although several different parameterizations can be found in literature (see e. g. [BSW19b, MM14]), $\alpha \leq 1$ and $\beta \geq 2$ are common modeling choices. This implies a polynomial growth of the BPR functions, consistent with real-world observations: Few vehicles on the road do not interfere with each other. However, as soon as the road starts to congest (i. e. $c \geq c_{max}$), the slowdown is worsened with every additional vehicle.

The BPR function enables the conversion from traffic loads into travel times. In a time-dependent scenario, however, the function must be applied to all buckets of the considered edge. We must ensure that combining the resulting set of breakpoints produces a FIFO-consistent travel time profile (see Section 2.2). To solve this issue, we first convert the bucket loads into *piecewise constant speed functions*. The function

$$v(s, t_0, c, c_{max}) = \frac{s}{t\left(t_0, c, c_{max}\right)} = \frac{s}{t_0 * \left(1 + \alpha * \left(\frac{c}{c_{max}}\right)^{\beta}\right)} \tag{3.2}$$

extends Equation (3.1) to derive velocities from traffic loads. Note that this conversion step also considers the travel distance s along the respective edge. The other parameters of the BPR function remain identical. Finally, the piecewise constant speed profiles are converted into piecewise linear travel time profiles. For this purpose, we apply a method from [SZ21] in our implementation. The adapted procedure assumes that the velocity of all drivers changes instantly and simultaneously at the breakpoints of the speed profiles. Consequently, no “overtaking“ takes place at any road segment. This preserves the FIFO property. We refrain from further details at this point. The entire conversion step is depicted in Figure 3.2 and takes linear time $\mathcal{O}(|B|)$ in the number of breakpoints of the speed profile.

Algorithm 3.1: Cooperative Traffic Distribution

Input: Capacity-Graph $G_{T,k} = (V,\ E,\ C,\ C_{max})$
Input: User Requests $R =$ `listOf(`$s,\ t,\ \tau^{dep}$`)`, ordered by departure τ^{dep}
Output: `listOf(`shortest path $P = (s, \ldots, t)$`)`

```
1  for i from 0 to |R| do
       // determine the shortest path for the current request
2      (s, t, τ^dep) ← R[i]
3      P ← shortestPath(G_{T,k}, s, t, τ^dep)
       // update the edge weights along P, return result
4      updateWeights(G_{T,k}, P, τ^dep)
5      return P to user i
6  function updateWeights(G_{T,k}, P, τ^dep):
7      forall edges e = (v, w) ∈ P do
           // get timestamp at current node, determine bucket
8          τ_v ← departureAtNode(v, τ^dep)
9          b ← roundToBucketId(T, k, τ_v)
           // increment traffic load, update travel time profile
10         incrementEdgeBucket(G_{T,k}, e, b)
11         updateTravelTimeFunction(G_{T,k}, e)
```

Up to this point, we have demonstrated how to store traffic loads and how to convert these into FIFO-consistent travel time functions. This allows us to extract a time-dependent graph $G_T = (V, E, len)$ with weight function $len : (E \times T) \to \mathbb{R}_+$ from a given Capacity-Graph $G_{T,k} = (V, E, C, C_{max})$. Next, we describe our approach of distributing the given requests evenly over the road network. As already mentioned, the incoming requests are sorted by departure time and must be processed sequentially. Our traffic distribution strategy is outlined in Algorithm 3.1. For each request, we apply a shortest path algorithm to retrieve the path P. In the subsequent *update* phase (lines 6–11), the travel times along P are slightly incremented at the affected timestamps. First, the arrival time τ_v at each node $v \in P$ is determined by traversing P. We then round τ_v to the corresponding bucket id $b = \lfloor \frac{\tau_v}{p} \cdot k \rfloor$ with respect to the period p of T. Finally, the traffic load $C(e, b)$ of the considered edge $e \in E$ is incremented by one unit, and the travel time function gets updated accordingly. The next request will then be solved on the updated graph to incorporate all available traffic information.

To reduce the memory consumption of the Capacity-Graph, we refrain from storing an array of k bucket entries for each edge. Doing so would dominate the required storage for large k. Moreover, a significant part of the edges might be irrelevant for the queries. For example, we will unlikely consider dirt tracks in rural areas. Besides that, only a fraction of all buckets are utilized after processing a small set of queries. Our implementation provides a more compact storage layout instead. We only store the used buckets of each edge and maintain these sorted by bucket ids. Hence, updating and inserting new bucket entries takes logarithmic time. Compared to keeping an array of fixed sized k for each edge, our approach reduces the overall memory consumption if less than 50% of all edge buckets are used.

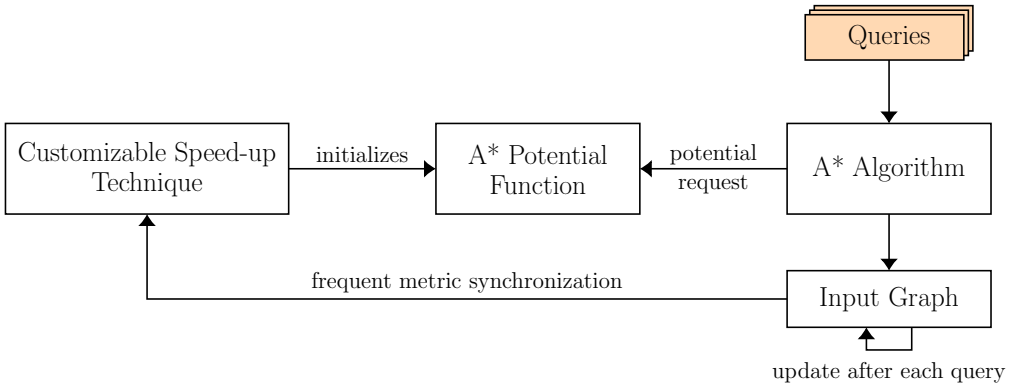


Figure 3.3: Scheme of decoupling the customizable speed-up technique from the input graph on which the queries are processed. This procedure is advantageous on dynamic graphs, assuming that minor edge weight adjustment do not violate the correctness of the A* potential.

3.3 Applying Speedup Techniques

In this section, we discuss issues related to the design of efficient shortest path algorithms for cooperative route planning. Most importantly, the applied algorithm must be able to adapt to permanently changing edge weights. Unfortunately, the majority of known (time-dependent) speed-up techniques [BDG+16, Del09] assume that the edge weights do not change while processing the queries. Particularly techniques with long preprocessing times are therefore infeasible for this dynamic setting. We are even unable to leverage customizable approaches like CCH. As our goal is to incorporate all available traffic information, we would have to run the customization step after each query.

Of all mentioned speed-up techniques, goal-directed heuristics (A* potentials) seem to be the most suitable for dynamic graphs with permanently changing weights. Theorem 2.12 (see Section 2.3.1) proves that the A* algorithm is guaranteed to find the optimal solution as long as the applied heuristic provides lower-bound estimates. At this point, we can exploit the simplicity of our cooperative model. Due to the one-time route assignments, the traffic load of an edge bucket is never decreased. Hence, it is possible to design A* potentials which remain valid after processing an arbitrary amount of queries. The estimation quality will continuously degrade, though.

As already seen in Section 2.3.3, it is possible to combine speed-up techniques with the A* algorithm to provide fast and tight heuristics. Furthermore, the speed-up technique can be decoupled from the input graph. Updates after every step are thus obsolete unless the heuristic provides invalid estimates. Inspired by *CCH-Potentials* [SZ21], Figure 3.3 shows a generalized scheme of a speed-up technique for cooperative route planning. The queries are solved with the time-dependent A* algorithm on the input graph, which gets updated after each query. Moreover, the applied heuristic is initialized by a customizable speed-up technique. This technique, in turn, is frequently customized with the current edge weights to keep providing tight estimates.

In the next chapter, we will analyze the presented scheme in more detail. Based on the *CCH-Potential* algorithm, we will design time-dependent A* potentials whose estimates also depend on the given daytime.

4. Time-Dependent A* Potentials

In this chapter, we focus on designing time-dependent A* potentials to provide efficient algorithms for cooperative route planning. A general scheme of exploiting the A* algorithm to decouple speed-up techniques from the input graph has been presented in Section 3.3. As we refrain from subsequent route adjustments in our cooperative model, the stored traffic loads never decrease along any segment. Hence, the *lower-bound* property of A* potentials is preserved after updating edge weights. However, the estimation quality degrades with each additional query. Therefore, the underlying speed-up technique should be customized frequently with the adjusted travel time profiles of the input graph.

So far, most studies related to A* potentials have only considered time-independent graphs. Although a few heuristics also cover time-dependence [DW09, SZ21], their estimates for each node are constant throughout the day. In the following, we introduce a novel approach of *time-dependent A* potentials*. Contrary to common A* heuristics, these also consider the time component τ.

Definition 4.1 (Time-Dependent A* Potential Function)**.** *Consider the time-dependent graph $G_T = (V, E, len)$. A time-dependent A* potential is a function*

$$\pi_{s,t,\tau^{dep}} : (V \times T) \to \mathbb{R}_+, \tag{4.1}$$

which estimates the travel time between each node $v \in V$ and the target $t \in V$ when departing v at time $\tau \in T$. During its computation, the source node $s \in V$, the target node t, and the departure time $\tau^{dep} \in T$ can be taken into account.

Due to the additional parameter of the potential function, it is necessary to redefine the A* correctness properties from Section 2.3.1. Fortunately, the relation between the *feasibility* and the *lower-bound* property (see Theorem 2.11) also holds in the time-dependent scenario. A *feasible* potential $\pi_{s,t,\tau^{dep}}$ with $\pi_{s,t,\tau^{dep}}(t, \cdot) = 0$ hence provides *lower-bound* estimates.

Definition 4.2 (Feasibility Property of Time-Dependent A* Potentials)**.** *A time-dependent A* potential $\pi_{s,t,\tau}$ on the graph $G_T = (V, E, len)$ is feasible if*

$$len(e, \tau) + \pi_{s,t,\tau^{dep}}(v, \tau + len(e, \tau)) - \pi_{s,t,\tau^{dep}}(u, \tau) \geq 0 \tag{4.2}$$

for all edges $e = (u, v) \in E$ and timestamps $\tau \in T$.

Definition 4.3 (Lower-Bound Property of Time-Dependent A* Potentials)**.** *Consider the time-dependent graph $G_T = (V, E, len)$. The time-dependent potential function $\pi_{s,t,\tau^{dep}}$ satisfies the lower-bound property if*

$$\pi_{s,t,\tau^{dep}}(v, \tau) \leq d(v, t, \tau)$$

with respect to the shortest distance $d(v, t, \tau)$ between any node $v \in V$ and the target $t \in T$ at departure time $\tau \in T$.

In the following sections of this chapter, we examine time-dependent A* potentials in more detail. Section 4.1 contains an overview of different design approaches and points out the most promising ideas. After that, we propose two time-dependent A* potentials. We also discuss key aspects of an efficient implementation of these potentials. The *Multi-Metric-Potential* introduced in Section 4.2 applies a time-independent CCH which handles multiple weight functions simultaneously. At the cost of longer preprocessing times, the *Corridor-Lowerbound-Potential* (Section 4.3) uses a time-dependent CCH to allow tighter estimates. Both approaches provide time-dependent estimates based on the source node, the target node, and the departure time.

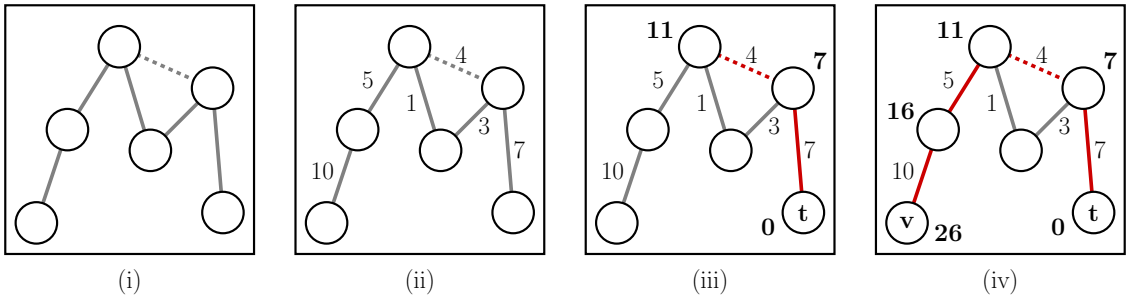
4.1 Engineering Time-Dependent Potentials

A naive method to obtain a time-dependent potential function is to perform a backward *Dijkstra profile query* as described in Section 2.2. Before executing the A* algorithm, we initialize the heuristic by conducting a *One-To-All* profile search that starts at the target node and explores the entire reversed graph. The resulting distance labels are then used as estimates in the query step. It suffices to evaluate the labels of the requested nodes at the given timestamps. As discussed in Section 2.2, we can employ approximation algorithms to reduce the computation effort of the *linking* and *merging* operations. Applying a lower-bound approximation to the travel time profiles preserves the *lower-bound* property and thus the correctness of the resulting A* heuristic.

Using profile queries to compute time-dependent A* heuristics has two advantages. Firstly, no preprocessing has to take place in advance. Secondly, we always consider the updated travel time functions of the input graph in our calculations. Aside from the applied approximations, the resulting potential estimates are therefore precise. This leads to fast query times of the A* algorithm. Unfortunately, conducting profile queries is too time-consuming and thus not applicable. While (approximated) profile queries take several minutes on city-sized graphs, the time-dependent Dijkstra modification solves the actual request within a second. To compete with *CCH-Potentials* (see Section 2.3.3), the initialization of the time-dependent A* heuristic must even occur within milliseconds.

As mentioned before, the linking and merging operations constitute the major performance bottleneck of profile queries. Their negative impact on the running time does not justify the usage of a tight heuristic in the subsequent A* execution. For this reason, we must avoid these operations during the potential initialization. Consequently, maintaining travel time profiles is not feasible here. We can still incorporate time-dependence by initializing the A* heuristic with the known parameters $s, t \in V$ and $\tau \in T$. However, using scalar edge weights implies that the provided estimates are constant, regardless of the current time at the visited node.

To obtain time-dependent potentials efficiently, we take a step back and reconsider the *CCH-Potential* algorithm from Section 2.3.3. The heuristic can be adapted to time-dependent graphs by applying the lower-bound metric $\underline{len}$ to compute the estimates. It is trivial to show that the CCH-Potential is *feasible*. CCH-Potentials are also fast to compute. The



(i) (ii) (iii) (iv)

Figure 4.1: Overview of the different stages of an A* potential based on Lazy RPHAST: (i) Metric-Independent Preprocessing, (ii) Metric-Dependent Preprocessing, (iii) Potential initialization, (iv) Potential request.

precision of the estimates can be improved, though. While $\underline{len}$ provides tight estimates at daytimes with low traffic (e. g. around midnight), there is little accuracy during traffic peaks in the morning and afternoon. Hence, $\underline{len}$ is not representative of traffic flows throughout the day.

Despite this problem, *CCH-Potentials* are still a good starting point to design time-dependent A* potentials. Both of our heuristics use CCH as underlying speed-up technique. Moreover, the potentials are based on the scheme depicted in Figure 4.1. This generic scheme divides the algorithm into four stages. The first two stages are related to the CCH preprocessing (see Section 2.3.2). In the initial *metric-independent* preprocessing step, the nodes are ranked by their importance. Based on this order, the nodes are contracted to construct a CCH graph with additional shortcut edges and its corresponding elimination tree. The second preprocessing step is frequently executed and applies the current edge weights to the CCH graph. We intentionally call this step *preprocessing* and not *customization* because the latter is only a sub-routine. Both proposed potentials will perform further operations before and after the customization. The queries are processed in the third (*initialization*) and fourth (*potential request*) stage. These stages are processed as described for *Lazy RPHAST* in Section 2.3.3. After initializing the backward search space from the target node, the forward graph is lazily explored during the potential requests. Our proposed heuristics must implement these two stages efficiently. To outperform *CCH-Potentials*, we have to provide time-dependent estimates without causing significant computation overhead.

4.2 Multi-Metric Potential

Our first proposed time-dependent A* heuristic is the *Multi-Metric-Potential*. It extends the *CCH-Potential* by applying multiple metrics simultaneously. Each metric $len_{\mathcal{I}} : E \to \mathbb{R}_+$ provides lower-bound travel times within a given time interval $\mathcal{I} \subseteq T$. Analogously to *CCH-Potentials*, we apply Lazy RPHAST to compute the potentials. We instantiate the routine with a metric $len_{\mathcal{I}}$ whose corresponding time interval covers the departure time $\tau^{dep} \in T$ as well as the arrival time τ^{max} at the target node. This is necessary to ensure the *feasibility* property of the resulting A* heuristic.

We perform a separate *elimination tree query* on the upper-bound travel times $\overline{len}$ to determine the latest arrival time τ^{max} at the target node. Although this bound on τ^{max} is rather conservative, it can be computed fast. In our cooperative setting, we must also be aware that $\overline{len}$ is not static. Each modification of the travel time functions may violate the correctness of the customized upper-bound metric. Whenever such a violation is detected, we have to re-run the customization before continuing with the following queries.

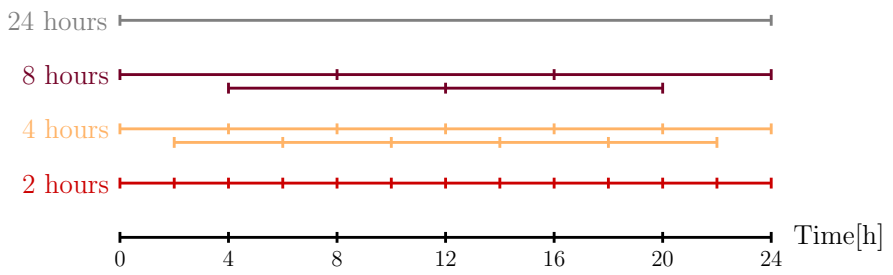


Figure 4.2: The *Multi-Metric-Potential* is initialized with a set of time intervals that cover different ranges. A separate metric with lower-bound weights is extracted for each given interval.

Preprocessing. We initialize the preprocessing with a set of time intervals $I = \{\mathcal{I}_1, \dots\}$. Based on these intervals, the metrics $\mathcal{M} = \{len_{\mathcal{I}_1}, \dots\}$ are extracted in advance. Each metric $len_{\mathcal{I}} \in \mathcal{M}$ provides lower-bound weights within its corresponding interval, i. e.

$$len_{\mathcal{I}}(e) = \min_{\tau \in \mathcal{I}} len(e, \tau)$$

for all edges $e \in E$ of the input graph. To deliver tight estimates during the query phase, the selectable metrics should be based on a variety of overlapping time intervals I. As depicted in Figure 4.2, we recommend to cover the entire day by intervals of different lengths. Since the lower-bound metric $\underline{len}$ can be applied for all possible inputs $[\tau^{dep}, \tau^{max}]$, we require $T \in I$ such that $\underline{len} \in \mathcal{M}$. During the query step, we can fall back to this metric if no other candidates qualify.

After the metric extraction, the CCH graph is customized with these metrics. To process all metrics at once, the basic CCH algorithm is slightly adjusted. Although we consider different time intervals, the entire customization step processes time-independent weights. For simplification, we omit the *perfect customization* step which removes unnecessary edges subsequently (see [DSW14] for further details). After that, we reorder the edge weights into a flat array, grouped by each metric. The purpose of this reordering is to improve the cache performance. If the outgoing edges of a node are laid out consecutively in memory, several entries will be fetched with a single data access. Compared to grouping the weights by edges, this results in significant performance gains.

Initialization and Potential Request. These stages work mostly similar to Algorithm 2.2 outlined in Section 2.3.3. Therefore, we only point out the adjustments to the basic Lazy RPHAST routine. At the beginning of the initialization, an *elimination tree query* is run to determine the latest arrival time τ^{max} at the target node. After that, we pick a suitable metric $len_{\mathcal{I}} \in \mathcal{M}$ such that $[\tau^{dep}, \tau^{max}] \subseteq \mathcal{I}$. If multiple metrics qualify, we pick the metric with the shortest corresponding time interval. Figure 4.3 shows that choosing smaller intervals can improve the precision of the heuristic. Some visualized interval minima are significantly tighter than the trivial lower bound over the entire day. Afterward, we initialize Lazy RPHAST with $len_{\mathcal{I}}$ to compute the A* heuristic. Additional *pruning* can be applied to speed up the potential computations. In particular, we exploit that each node with a distance label greater than $\tau^{max} - \tau^{dep}$ is not relevant for the current query. The estimates for these nodes can be set to infinity without violating the correctness. During the execution of the A* algorithm, the affected nodes will not be visited at all. All other operations run analogously to the *CCH-Potential* algorithm.

Metric reduction. Applying several metrics simultaneously improves the accuracy of the potential estimates. However, it also leads to high memory consumption and additional computation effort during customization. Our approach of reducing the number of metrics

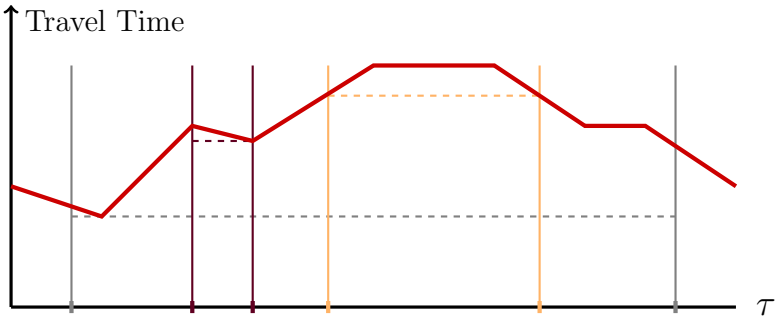


Figure 4.3: Extraction of several interval minima from a travel time function. The determined lower-bound weights (dashed) strongly depend on the given interval boundaries (marked in different colors).

is based on the observation that some metrics have similar edge weights. This particularly affects the metrics that cover time intervals with low traffic volumes. During the *metric reduction* step, we *merge* pairs of similar metrics to a single weight function. The input parameter $k \in \mathbb{N}$ controls how many metrics remain after the reduction. It is important to mention that this procedure does not restrict the set of usable time intervals during the query. Instead, the affected intervals point to the same metric after a merging step. The merged metric must therefore provide lower-bound weights for both of these intervals.

The metric reduction procedure is described in Algorithm 4.1. Initially, each pre-defined time interval $\mathcal{I}$ references a custom metric $len_{\mathcal{I}} \in \mathcal{M}$ as defined above. Then, all metrics are compared with each other. We quantify the similarity between two metrics len_i, len_j by the *square sum* of their weight differences. More precisely, we compute

$$\sum_{e \in E} (len_i(e) - len_j(e))^2$$

over all edges $e \in E$. This approach particularly emphasizes large variations between edge weights. Therefore, similar metrics are characterized by predominantly small deviations. After the comparison, we iteratively merge the two most similar metrics. To enable fast retrieval of these metrics len_i, len_j, we maintain all pairs in a priority queue (ordered by their squared differences). When merging len_i and len_j, we obtain the function len' such that

$$len'(e) = \min\{len_i(e), len_j(e)\}.$$

Hence, len' provides a lower bound for all intervals that have previously referenced len_i or len_j. To conduct the reduction in-place, we implicitly set $len_i = len'$. For this reason, all intervals which have previously been pointed at len_j must afterward reference len_i (see line 8). Moreover, all metric comparisons related to len_i must be repeated (lines 9–10). All queue entries that refer to len_j, on the other hand, are obsolete and may be removed.

Overall, the proposed metric reduction algorithm runs in time $\mathcal{O}(|\mathcal{M}|^2 \cdot |E| \cdot \log(|\mathcal{M}|))$. The quadratic number of metric comparisons (lines 3–4 and 9–10) constitutes the main bottleneck. Especially on large graphs, the procedure can be time-consuming. This underlines the need to constrain the initial set of intervals without compromising the accuracy of the resulting A* potentials too much. An important measure is to omit intervals outside of expected traffic peak hours. Additional performance improvements can be achieved by parallelizing the initial metric comparisons. Another optimization is to *pre-merge* similar metric pairs with little differences before performing further comparisons.

Algorithm 4.1: Generic Metric Reduction

Input: Metric reduction threshold $k \in \mathbb{N}$
Input: Set of metrics $\mathcal{M} = \{len_1, \ldots, len_n\}$ and intervals $\mathcal{I} = \{\mathcal{I}_1, \ldots, \mathcal{I}_n\}$
Output: Reduced set of metrics $\mathcal{M}$

```
1  function reduceMetrics(M, I, k):
       // init queue - insert all pairs
2      Q ← priorityQueue()
3      forall (len_i, len_j) ∈ M^2: i < j do
4          Q.insert({i, j}, metricDiff(len_i, len_j))
5      while more than k metrics remaining do
6          {i, j} ← Q.pop() // wlog i < j
           // merge metrics, update interval data
7          mergeMetrics(len_i, len_j)
8          updateIntervalReferences(I, len_j → len_i)
9          forall {k, i} ∈ Q do
10             Q.update({k, i}, metricDiff(len_k, len_i)) // update entries with i
11         forall {k, j} ∈ Q do
12             Q.remove({k, j}) // remove entries with j
13     return M
```

Up to this point, we have only claimed that the *Multi-Metric-Potential* is *feasible*. We will now formally prove this claim. As the potential estimate for the target node is always zero, *feasibility* also implies the *lower-bound* property.

Theorem 4.4. *The Multi-Metric-Potential is feasible if its upper-bound metric $\overline{len}$ is valid.*

Proof. Consider the time-dependent graph $G_T = (V, E, len)$ with upper-bound weights $\overline{len}$ and let $\pi_{s,t,\tau^{dep}}$ be a Multi-Metric-Potential. $\pi_{s,t,\tau^{dep}}$ is initialized with the source node $s \in V$, the target node $t \in V$, and the departure time $\tau^{dep} \in T$. We assume that the estimates of $\pi_{s,t,\tau^{dep}}$ are based on the metric $M : E \to \mathbb{R}_+$. Moreover, let $d_M : (V \times V) \to \mathbb{R}_+$ be the shortest distance function between any pair of nodes by using the weights of M. Then, $d_M(v,t) = \pi_{s,t,\tau^{dep}}(v, \cdot)$ follows for all $v \in V$. Hence, we get

$$\begin{aligned} & len(e,\tau) + \pi_{s,t,\tau^{dep}}(v, \tau + len(e,\tau)) - \pi_{s,t,\tau^{dep}}(u,\tau') \\ &= len((u,v),\tau) + d_M(v,t) - d_M(u,t) \\ &\geq d_M(u,v) + d_M(v,t) - d_M(u,t) \end{aligned} \tag{4.3}$$

for each edge $e = (u,v) \in E$ and timestamp $\tau \in T$. We can now apply the triangle inequality

$$d_M(u,t) \leq d_M(u,v) + d_M(v,t) \tag{4.4}$$

which states that $d_M(u,t)$ must not be shortened by visiting v along the path. Thus,

$$len(e,\tau) + \pi_{s,t,\tau^{dep}}(v, \tau + len(e,\tau)) - \pi_{s,t,\tau^{dep}}(u,\tau) \geq 0 \tag{4.5}$$

which was to be proven. □

To conclude, the *Multi-Metric-Potential* represents an extension of the *CCH-Potential* that provides time-dependent potential estimates. By applying several pre-calculated metrics for different time intervals, we leverage time-independent CCH to provide time-dependent A* heuristics efficiently. However, *long-range* queries pose an issue for the *Multi-Metric-Potential*. If $[\tau^{dep}, \tau^{max}]$ covers large parts of the day, the approach will fall back to the metric $\underline{len}$. The estimates of *Multi-Metric-Potentials* and *CCH-Potentials* are identical in this case and do not justify the additional initialization effort. We present another time-dependent A* heuristic in the next section to solve this issue.

4.3 Corridor-Lowerbound Potential

As discussed in the last section, the *Multi-Metric-Potential* is not optimized for long-range queries. For large time intervals $[\tau^{dep}, \tau^{max}] \subseteq T$, the approach falls back to the lower-bound metric $\underline{len}$ to obtain a *feasible* heuristic. To compute tighter estimates, we consider shorter time intervals for each edge in the following. Although it can take several hours to traverse a route, each road segment is only relevant within a small time window.

The *Corridor-Lowerbound-Potential* determines individual time intervals for each edge. Similar to the *Multi-Metric-Potential*, this approach represents a modification of Lazy RPHAST to compute time-dependent A* heuristics. Instead of using pre-defined and time-independent metrics, we must employ a time-dependent CCH to retrieve tight lower-bound weights within arbitrary intervals. During each query, the relevant interval $\mathcal{I}(v)$ is determined for all nodes $v \in V$ visited by the A* algorithm. For each outgoing edge $(v, w) \in E'$ in the time-dependent CCH graph $G'_T = (V, E', len')$, we use the lower-bound weight

$$\min_{\tau \in \mathcal{I}(v)} len'((v, w), \tau)$$

within the interval $\mathcal{I}(v)$. This step is embedded into a Lazy RPHAST routine, which in turn computes the A* heuristic.

The resulting potential estimates can be seen as lower-bound weights obtained from an implicit subgraph, also denoted as *corridor*. This subgraph contains the source and target node $s, t \in V$ and at least all relevant nodes for the shortest path retrieval between s and t. Moreover, it restricts the domain of each travel time function $len((v, w), \cdot)$ to the corresponding time interval $\mathcal{I}(v)$ of the node $v \in V$.

Definition 4.5 (Corridor subgraph)**.** *Consider the time-dependent graph $G_T = (V, E, len)$ and the shortest distance functions $\underline{d}, \overline{d} : (V \times V) \to \mathbb{R}_+$ related to the lower and upper-bound metrics $\underline{len}$, $\overline{len}$. Moreover, the time interval function $\mathcal{I}(\cdot)$ assigns*

$$\mathcal{I}(v) = [\tau^{dep} + \underline{d}(s, v), \tau^{dep} + \overline{d}(s, v)] \subseteq T$$

to each node $v \in V$, with respect to the departure time τ^{dep} at the source node $s \in V$. Then, the corridor $C = (V_C, E_C, len_C)$ is a subgraph of G_T with partial travel time profiles. It consists of

- *the nodes $V_C = \{v \in V : \underline{d}(s, v) \leq \overline{d}(s, t)\}$,*
- *the edges $E_C = \{(v, w) \in E : v \in V_C \wedge w \in V_C\}$,*
- *and partially defined edge weights $len_C : (E \times T) \to \mathbb{R}_+$ such that*

$$len_C((v, w), \tau) = \begin{cases} len((v, w), \tau) & \tau \in \mathcal{I}(v) \\ \bot & otherwise \end{cases}$$

for each edge $(v, w) \in E_C$.

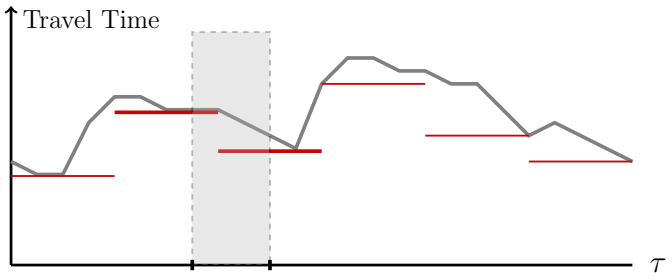


Figure 4.4: Approximating a complex travel time function by piecewise constant segments. The segments (straight red lines) cover time intervals of equal sizes and mark their respective minimum travel time. To obtain a lower-bound weight within $\mathcal{I}(v)$ (gray rectangle), it is sufficient to compare the two intersecting segments.

Intuitively, the corridor C contains a superset of all relevant nodes for the current query. Applying a *Corridor-Lowerbound-Potential* on G_T thus roughly corresponds to using the *CCH-Potential* on the corridor C. We emphasize that C is an implicit structure that we will not extract separately. The performance of our approach depends on the size of the corridor, though. In the worst-case, the A* algorithm requires estimates for all nodes $v \in V_C$.

To compute the heuristic efficiently, the time intervals $\mathcal{I}(\cdot)$ must be determined fast. We perform a separate Lazy RPHAST run for these calculations. The subroutine processes lower and upper-bound distances and starts at the given source node. Some minor adjustments to Algorithm 2.2 presented in Section 2.3.3 suffice. Most importantly, the search direction is reversed. In the initialization step, the forward search space of the source node is explored. The distances labels are then lazily expanded on the reversed downward graph. We must be aware that obtaining $\mathcal{I}(v)$ for each inspected node $v \in V$ already requires as much computation effort as the entire *CCH-Potential* algorithm. Therefore, the resulting estimates must be precise enough to balance this negative performance impact during the execution of the A* algorithm. Next, we describe the different stages of the *Corridor-Lowerbound-Potential* and discuss several optimizations.

Preprocessing. Based on the input graph, we first conduct a time-dependent CCH customization as described in the CATCHUp paper [SWZ21]. After the customization, we approximate the resulting travel time functions by $k \in \mathbb{N}$ constant segments which cover equally sized time intervals. Figure 4.4 depicts the procedure. Using this rather unusual approximation scheme has several advantages. Instead of keeping complex travel time functions with thousands of breakpoints, we can adjust k to control the memory utilization. Storing the same number of entries for each edge can also improve the cache performance, as we will discuss later. Besides that, the extraction of the minimum edge weight within a given interval $\mathcal{I}(v)$ is not as complex as evaluating piecewise linear travel time functions. As shown in the figure, it suffices to compare all approximated segments which intersect with $\mathcal{I}(v)$.

Initialization. Algorithm 4.2 describes the procedure during the initialization and the potential requests. Similar to the *CCH-Potential*, we explore the search space of the target node t on the reversed downward graph $\overleftarrow{G}$ in the initialization step. However, we must be aware that linking an edge $e = (v, w) \in \overleftarrow{G}$ occurs in backward direction. As all edges are traversed in forward direction during the query, we use the lower-bound weight of e within the time interval $\mathcal{I}(w)$ assigned to node w. After that, the backward distance labels are updated accordingly (lines 6–8).

Algorithm 4.2: CORRIDOR-LOWERBOUND POTENTIAL

Input: CCH forward/backward graphs $\overrightarrow{G} = (V, \overrightarrow{E}, \overrightarrow{len})$, $\overleftarrow{G} = (V, \overleftarrow{E}, \overleftarrow{len})$
Input: CCH elimination tree `tree`: $V \to V$
Data: Tentative distances $B(\cdot)$ to the target node, initially ∞
Output: Potential estimates $D(\cdot)$ for each node $v \in V$, initially $\perp$

1 **function** `init`(s, t, τ^{dep})**:**
2 u $\leftarrow t$
3 **while** u $\neq \perp$ **do**
  `// init search space of` t `with respect to` τ^{dep}
4 **forall** e $\leftarrow$ (u, v) $\in \overleftarrow{E}$ **do**
5 $\mathcal{I}(v) \leftarrow$ `relevantInterval`(v, τ^{dep})
6 intervalMin $\leftarrow \min_{\tau \in \mathcal{I}(v)} \overleftarrow{len}(\mathrm{e}, \tau)$
7 B[v] $\leftarrow \min\{B[\mathrm{v}], B[\mathrm{u}] + \text{intervalMin}\}$
8 u $\leftarrow$ `tree`(u)

9 **function** `potential`(v)**:**
10 u $\leftarrow v$, stack $\leftarrow$ `Stack()`
  `// determine non-explored forward-upward search space`
11 **while** u $\neq \perp$ `&&` D[u] $= \perp$ **do**
12 stack.push(u), u $\leftarrow$ `tree`(u)
  `// descend search space in reverse order`
13 **while** stack *is not empty* **do**
14 u $\leftarrow$ stack.pop()
15 $\mathcal{I}(u) \leftarrow$ `relevantInterval`(u, τ^{dep}) `// same for all out-edges`
16 **forall** e $\leftarrow$ (u, x) $\in \overrightarrow{E}$ **do**
17 intervalMin $\leftarrow \min_{\tau \in \mathcal{I}(u)} \overrightarrow{len}(\mathrm{e}, \tau)$
18 B[u] $\leftarrow \min\{B[\mathrm{u}], B[\mathrm{x}] + \text{intervalMin}\}$
19 D[u] $\leftarrow B$[u]
20 **return** $D[v]$

Potential request. The forward graph $\overrightarrow{G}$ is explored lazily during the potential retrieval step. Analogously to the Lazy RPHAST algorithm, all undiscovered upward nodes are collected and processed in reversed order (lines 10–12). Contrary to the initialization, linking edges now occurs in forward direction. Therefore, the relevant time interval $\mathcal{I}(v)$ is identical for all outgoing edges of the node v. Updating the distance labels with the corresponding interval minimum (lines 18–19) works similar to the initialization phase.

As mentioned before, applying another Lazy RPHAST routine to determine the time intervals $\mathcal{I}$ doubles the required work to compute the potentials. Obtaining the minimum edge weights results in additional computation effort, particularly if the intervals cover several approximated segments. Therefore, thorough engineering is necessary to achieve significant speedups over the *CCH-Potential*. Similarly to the *Multi-Metric-Potential*, we can perform *pruning* and skip all nodes whose distance label exceeds the upper-bound distance $\overline{\tau_t} - \tau^{dep}$ between source and target node. $\overline{\tau_t}$ can be obtained by running a separate interval request $\mathcal{I}(t)$ for the target node.

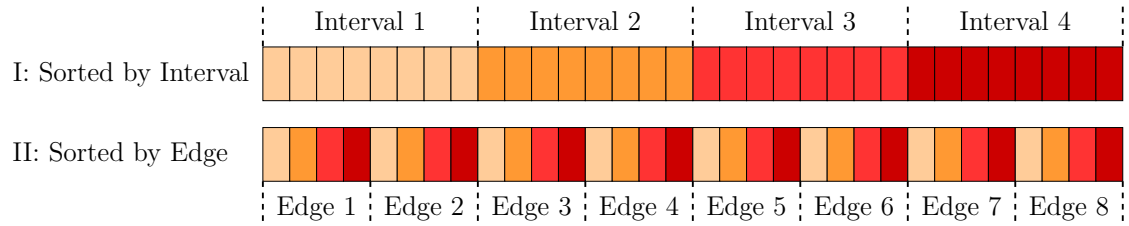


Figure 4.5: Possible storage layouts of the edge intervals in a single-dimensional array. The intervals are colored to enable a better distinction between the approaches.

Approximating complex travel time functions into equally sized time intervals is another measure to reduce memory consumption. Here, we can exploit that all approximated edge functions follow the same structure, i. e. their breakpoints have identical timestamps. Therefore, the weights can be stored in a single-dimensional array and accessed in constant time by their edge id and the rounded timestamp. The procedure works as described in Section 4.2. Figure 4.5 depicts the two possible storage layouts. We have observed that grouping the array by interval timestamp (approach I) yields better cache performance. Assuming that the outgoing edges of a node are laid out consecutively in memory, they will not be flushed out of the cache if the interval $\mathcal{I}(v)$ of the corresponding node $v \in V$ only spans over a few entries. By grouping the weights by edge instead, we would not be able to reuse previous cache entries when proceeding with the next edge. The main reason is that the number k of breakpoints per edge function is usually too high to fit all associated segments into a single cache line. Apart from that, the choice of k has another impact on the running time of the query phase. While large intervals (small k) reduce the memory consumption, they also worsen the accuracy of the potential estimates. We recommend choosing $k \in [48, 96]$ breakpoints per edge to achieve a reasonable trade-off. If the edge profiles cover a single day, the corresponding segments are bound to a range between 15 and 30 minutes. Assuming that $\mathcal{I}(v)$ covers 15 minutes on average, we have to carry out at most two comparisons per edge in expectation and still obtain tight potentials.

Last but not least, we formally prove the correctness of the *Corridor-Lowerbound-Potential*. We first show that the *lower-bound* property is satisfied if the time intervals $\mathcal{I}(\cdot)$ are valid for each node. According to Theorem 2.12, this implies that the A* algorithm may terminate upon the target node extraction from the queue. We also give a counterexample to disprove the *feasibility* property.

In the following, we reuse the notations from Definition 4.5. Particularly, we consider a time-dependent graph $G_T = (V, E, len)$ with bounds $\underline{len}, \overline{len}$ and the corresponding shortest distance functions $\underline{d}, \overline{d}$. We examine a query with source and target nodes $s, t \in V$ and departure time $\tau^{dep} \in T$.

Lemma 4.6. *Let $P = (s, v_1, \ldots, v_k, t)$ be a shortest path on the graph G_T, with respect to the departure time τ^{dep}. Each node $v \in P$ is also part of the corridor $C = (V_C, E_C, len_C) \subseteq G_T$.*

Proof. Suppose by contradiction that there is a node $v_i \in P \setminus V_C$. Then, the inequality

$$d(s, v_i, \tau^{dep}) \geq \underline{d}(s, v_i) > \overline{d}(s, t)$$

holds. Moreover, we obtain

$$d(s, t, \tau^{dep}) = d(s, v_i, \tau^{dep}) + d(v_i, t, \tau^{dep} + d(s, v_i, \tau^{dep}))$$

from the *sub-path* property of shortest paths. The contradiction

$$d(s, t, \tau^{dep}) \geq d(s, v_i, \tau^{dep}) > \overline{d}(s, t)$$

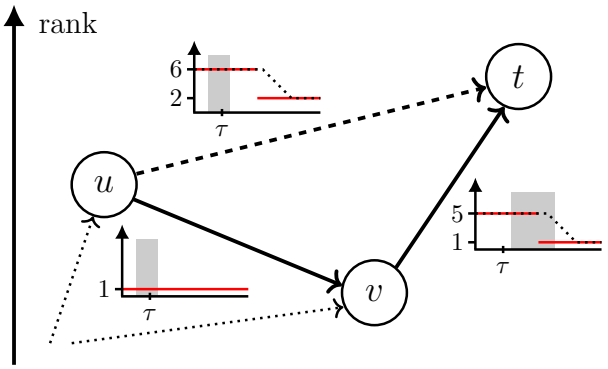


Figure 4.6: Counterexample to contradict the *feasibility* property of the *Corridor-Lowerbound-Potential*. The violation occurs at the edge (u, v).

follows directly because $d(\cdot)$ is non-negative. Therefore, such a node $v_i \in P \setminus V_C$ does not exist and all nodes $v \in P$ are part of the corridor. □

Due to this lemma, we can restrict our consideration to the corridor C. Next, we show that all nodes along the shortest path between s and t are visited within their time interval $\mathcal{I}(v)$.

Lemma 4.7. *All nodes $v \in V_C$ along the shortest path $P = (s, \dots, t)$ are visited within their corresponding time interval $\mathcal{I}(v) = [\tau^{dep} + \underline{d}(s, v), \tau^{dep} + \overline{d}(s, v)]$.*

Proof. It suffices to show that any access outside of $\mathcal{I}(v)$ is either impossible or irrelevant for the shortest path. The inequality $\underline{d}(s, v) \leq dist[v]$ follows directly from the correctness of the lower bound. Hence, visiting a node v before $\tau^{dep} + \underline{d}(s, v)$ is not possible.

Related to the upper bound case, it might be possible that v is extracted with a distance label greater than $\overline{d}(s, v)$. Assuming that $dist[v]$ will not be improved in a later iteration implies $dist[v] > \overline{d}(s, v)$. Suppose that $\overline{d}(s, v)$ is derived from the path $\overline{P} = (s, \dots, v)$. Traversing $\overline{P}$ then improves the distance between s and v, which contradicts to the *sub-path* property (Lemma 2.10) of P. As a result, v cannot be part of the shortest path between s and t. Visiting a node outside of its interval $\mathcal{I}(v)$ is either not possible or irrelevant for the shortest path retrieval, as to be proven. □

Theorem 4.8. *The Corridor-Lowerbound Potential provides lower-bound estimates for all nodes that are part of the corridor.*

Proof. According to Lemma 4.6, we can ignore all nodes which are not part of the corridor $C \subseteq G_T$. Moreover, Lemma 4.7 states that all nodes $v \in V_C$ along the shortest path $P = (s, \dots, t)$ must be visited within their given time interval $\mathcal{I}(v)$. For each edge $(v, w) \in E_C$, it is guaranteed that the extracted weight provides a lower bound within the range $\mathcal{I}(v)$. Therefore, all applied weights are lower-bound estimates of the actual distances. The resulting potential function thus satisfies the *lower-bound* property. □

Due to the intermediate approximation of the travel time functions into k constant segments, the *Corridor-Lowerbound-Potential* is not necessarily *feasibile*. A counterexample is provided in Figure 4.6. In this example, the *feasibility* property is violated for the edge (u, v). As the node u is more important than v, the edge (u, v) will not directly be considered in the potential computation. Instead, the shortcut edge (u, t) that combines (u, v) and (v, t) is used. Moreover, the time interval $\mathcal{I}(u)$ is chosen shorter than $\mathcal{I}(v)$. While $\mathcal{I}(u)$ only

covers the first segment of the approximated travel time function, both segments of the edge (v, t) must be considered. In this context, the potential of v is set too low to guarantee *feasibility*. By the given numbers in our example, we obtain

$$len((u,v),\tau) + \pi_{s,t,\tau^{dep}}(v, \tau + len((u,v),\tau)) - \pi_{s,t,\tau^{dep}}(u,\tau) = 1 + 1 - 6 = -4$$

which contradicts the *feasibility* property. As described in Section 2.3.1, the A* algorithm is thus *label-correcting* and might visit some nodes several times. This does not affect the correctness of the algorithm, though. Moreover, we have already discussed that approximating the edge functions enables further optimizations related to memory consumption and cache performance.

To sum up, both the *Multi-Metric-Potential* and *Corridor-Lowerbound-Potential* extend the *CCH-Potential* to incorporate time-dependence. Our approaches provide tighter potential estimates over the entire day. In the next chapter, we will compare the proposed heuristics in terms of their memory consumption and running times.

5. Experiments

We present the results of our experimental evaluation in this chapter. Section 5.1 and Section 5.2 describe implementation details as well as the experimental setup. After that, we examine the performance of our proposed time-dependent A* potentials in Section 5.3. Finally, Section 5.4 contains a comprehensive evaluation of the cooperative routing model. Besides evaluating memory consumption and running times, we also compare the model with common selfish routing strategies.

5.1 Experimental Setup

Our benchmark machine runs openSUSE Leap 15.3, and has 192 GiB of DDR4-2666 RAM and two Intel Xeon Gold 6144 CPUs, each of which has eight cores clocked at 3.5 Ghz and 8 × 64 KiB of L1, 8 × 1 MiB of L2, and 24.75 MiB of shared L3 cache. Unless stated otherwise, all experiments have been conducted sequentially.

The code is written in Rust and compiled with `rustc 1.58.0-nightly`. All experiments have been executed in the `release` mode with the `target-cpu=native` option. Our implementation can be found on GitHub[1]. It extends the `rust_road_router`[2] framework which provides efficient implementations for several speed-up techniques, including *CCH-Potentials* [SZ21] and *CATCHUp* [SWZ21]. We also reuse the given graph data structures. In addition to that, we run *InertialFlowCutter* [GHUW19] to obtain metric-independent node orders for the applied speed-up techniques.

Next, we describe the parameterization of our implementation. We interpret time-dependent edge weights as periodic functions whose domain covers a single day. The modulo operation is applied to map all timestamps into the domain $T = [0, p]$. Wherever possible, we represent travel time functions as integer weights with a precision of milliseconds, i. e. $p = 86.4 \cdot 10^6$. However, using integers is not possible for hierarchical time-dependent speed-up techniques. Each *linking* and *merging* step may produce breakpoints at arbitrary timestamps. Therefore, the corresponding algorithms operate on floating-point weights. To reduce the impact of inaccuracies, we use the range $T' = [0, 86.4 \cdot 10^3]$ instead.

The *Capacity-Graph* structure defined in Section 3.2 also requires further parametrization. Based on the results of our experiments, we recommend to use $k \in [50, 300]$ buckets per edge. Each bucket then represents a time interval of approximately 5–30 minutes. Additionally,

[1] Full implementation: `https://github.com/nils-we97/rust_road_router-fork`
[2] Basic framework: `https://github.com/kit-algo/rust_road_router`

Graph	#Nodes	#Edges	TD Edges [%]	#Breakpoints [$\times 10^6$]
Lux-20	58 511	123 718	71.67	2.40
Ger-06	4 688 214	10 795 826	7.25	23.79
Eur-17	25 757 978	55 503 819	27.16	455.60
Eur-20	28 510 049	60 898 831	76.28	1 011.60

Table 5.1: Statistics about the used PTV graph instances.

we have to calibrate the *BPR* function to convert traffic loads into velocities. As suggested by Buchhold et al. [BSW19b], we use the parameters $\alpha = 1$ and $\beta = 2$. Apart from that, our implemented Capacity-Graph stores both traffic loads, speeds, and travel time functions. At the cost of increased memory consumption, we achieve better performance while evaluating and updating the travel time profiles. As described in Section 3.2, we only maintain the traffic loads and velocities of used edge buckets.

For the *Corridor-Lowerbound-Potential* approach, we approximate the shortcut edges by $k = 72$ piecewise constant intervals of equal length. Consequently, each segment covers a time slice of 20 minutes. The *Multi-Metric-Potential* is instantiated with a generic pattern of time intervals, as shown in Figure 4.2 (see Section 4.2). These intervals range from one to 24 hours. For all experiments conducted in Section 5.3, we have reduced the interval density around midnight to accelerate the metric reduction step.

5.2 Graph Instances and Queries

We use both proprietary and publicly available graph instances for our evaluation. In Section 5.3, the performance of the time-dependent A* potentials is evaluated on proprietary graphs. All experiments related to the cooperative model are conducted on graph instances obtained from *OpenStreetMap*[3] (OSM). The usage of OSM data requires further preprocessing. First, we extract the road map as well as the travel times and distances of each road segment. A subroutine of the `RoutingKit`[4] framework is applied for this. As described in [ZNN11], we initialize the edge capacities according to the underlying road types. After that, falsely modeled edges with invalid capacities are removed and multi-edges are combined to a single edge. For simplification, we also restrict the graph to its largest strongly connected component. We use two different OSM instances. While the smaller graph models the road network of Berlin with 131k nodes and 287k edges, the road map of Baden-Württemberg contains 1.9M nodes and 4.4M edges.

The proprietary graphs used in our experiments are provided by the *PTV Group*[5]. Due to missing information about the edge capacities, we cannot conduct experiments related to the cooperative model on these graphs. Nonetheless, the instances provide real-world traffic predictions and are thus suitable to evaluate the quality of our proposed A* potentials. More details about the PTV instances are provided in Table 5.1. The last column (*#Breakpoints*) refers to the aggregated number of breakpoints of all travel time functions. Moreover, all edge profiles with at least two breakpoints are classified as time-dependent (third column).

Due to lacking real-world trip data, we have to use randomly generated query sets in our evaluation. We distinguish between four different query types: *uniform*, *geometric*, *population-uniform* and *population-geometric*. For *uniform* queries, both source and target node are drawn uniformly at random. Many evaluations found in literature are based on

[3]OpenStreetMap: `https://www.openstreetmap.org/about`
[4]Routing Kit: `https://github.com/RoutingKit/RoutingKit`
[5]PTV Group: `https://ptvgroup.com/`

these queries (see e. g. [GSSD08, DSW14, SZ21, SWZ21]). Although uniform queries are fast to generate, they often do not represent realistic traffic flows. Luxen and Sanders [LS11] claim that more realistic query sets can be obtained by using a geometric distribution with an expected travel distance of 40 kilometers. The *geometric* query type adopts this approach. We pick a random source node and determine the desired travel distance randomly from the geometric distribution. Then, we run Dijkstra's algorithm and set the target to the first node that exceeds the given distance threshold. Due to lacking travel distances in the PTV data, we assume an expected travel time of 45 minutes for these instances.

Even though geometric query sets are more realistic than uniform queries, the random choice of the source node still lacks plausibility. In reality, most trips start and end in densely populated areas. Following a recent work by Buchhold et al. [BSW19a], we incorporate publicly available population data. We use two different population grids with a precision of 100x100 meters for Germany[6] and 1x1 km for Europe[7]. A separate preprocessing step is applied to assign the nodes to their respective grid cells. The *population-uniform* approach picks source and target node in two separate steps. First, we select two random cells by their population density. After that, the source and target node are chosen randomly from the selected cells. Lastly, the *population-geometric* query type combines the aspects of *population-uniform* and *geometric* queries. The source node is picked randomly according to the population density. However, we refrain from choosing the target as the first node that surpasses the geometrically drawn travel distance d. Instead, we collect all cells visited in the distance range $[0.9d, 1.1d]$ and choose the target node from those cells. Again, population density is taken into account for the selection to ensure that the target node is likely located in a densely populated area.

In addition to the discussed query types, we also apply the *Dijkstra-Rank* [SS05] methodology. The Dijkstra rank of node v corresponds to the number of nodes settled by a Dijkstra run before extracting v. This way, we can evaluate short-, mid-, and long-range queries separately. Altogether, these five different query types offer a variety of query sets. Besides that, we must also generate reasonable departure times for the queries. We distinguish between a *uniform* and a *rush hour* departure distribution. While the uniform distribution approach picks the departure time uniformly at random, the latter applies a pattern from [TB10] to increase the probability of departing within a peak traffic hour. We claim that the use of *population-geometric* query sets combined with a *rush-hour* departure distribution provides a realistic approximation of real-world traffic flows.

5.3 Comparing Time-Dependent A* Potentials

First, we evaluate the memory consumption and running time of our time-dependent A* potentials discussed in Chapter 4. The corresponding experiments are conducted on the PTV graphs. To obtain a general overview of the applicability of time-dependent potentials, we refrain from adjusting the travel time profiles after each request. Instead, the focus is on evaluating the performance on time-dependent graphs with real-world traffic prediction. The *CCH-Potential* is used as a baseline heuristic in our experiments.

Preprocessing. Table 5.2 summarizes the results of evaluating the A* potentials in terms of memory consumption and preprocessing time. In all runs, the CCH customization is parallelized on 16 cores. For simplicity, we only measure the space utilization after the preprocessing and omit intermediate data structures. As expected, *CCH-Potentials* require

[6] Germany: `https://www.zensus2011.de/DE/Home/Aktuelles/DemografischeGrunddaten.html`

[7] Population grid, Europe: `https://ec.europa.eu/eurostat/web/gisco/geodata/reference-data/population-distribution-demography/geostat`

Graph	CCH-Pot		Multi-Metrics		Corridor-Lowerbound	
	space [GB]	time [s]	space [GB]	time [s]	space [GB]	time [s]
Lux-20	0.01	0.04	0.03	1.03	0.08	1.48
Ger-06	0.94	3.54	3.02	103.97	7.08	53.49
Eur-17	4.83	19.64	15.65	571.17	36.18	1220.53
Eur-20	5.26	22.34	17.04	649.07	40.60	2104.91

Table 5.2: Preprocessing time and space utilization of the different A* potentials.

the least storage utilization and preprocessing time. In addition to the graph structure, only the lower-bound metric *len* of the edge weights has to be customized and stored. By incorporating time-dependence in our A* potentials, we observe longer preprocessing times and higher memory utilizations. Depending on the parameterization, the intermediate memory utilization can be troublesome on large graphs for both *Multi-Metric-Potential* and *Corridor-Lowerbound-Potential*. We have observed temporary utilizations of up to 180GB, which is only slightly below the capacity of our benchmark machine. To avoid running out of memory, we restrict the *Multi-Metric-Potential* to 20 customized metrics and limit the degree of approximation in the *Corridor-Lowerbound-Potential* to at most 72 intervals per edge.

Further optimizations can be applied to reduce the preprocessing time of our time-dependent A* heuristics. We have observed that up to 90% of the time required to preprocess the *Multi-Metric-Potential* is caused by the *metric reduction* step. Even on the largest instance *Eur-20*, the CCH customization of 20 metrics only takes a minute. Therefore, using fewer time intervals would help to reduce the preprocessing time. For the *Corridor-Lowerbound-Potential*, the time-dependent customization (adopted from the *CATCHUp* [SWZ21] implementation) dominates the running time of the preprocessing step. We report preprocessing times of up to 30 minutes on continental-sized graph instances. At the expense of performance during the query phase, we can omit some stages of the customization [SWZ21]. Still, long preprocessing times are infeasible for some applications, particularly if the stage has to be re-run frequently.

Queries. Next, we evaluate the different potentials in terms of running time and search space size during the query stage. Apart from Dijkstra-Ranks, we consider all combinations of PTV graph instances and query types introduced in Section 5.2. While the departure time of the *uniform* query types is chosen uniformly at random, the *rush hour* departure distribution is applied for the *geometric* query sets. Table 5.3 shows the average running times and search space sizes over 10 000 queries. To provide a comparison with Dijkstra's algorithm, we have also applied a *Zero-Potential* which naively returns 0 as target distance estimate for all nodes.

Compared to Dijkstra's algorithm, the *CCH-Potential* reduces the search space size by up to two orders of magnitude. Both *Multi-Metric-Potential* and *Corridor-Lowerbound-Potential* achieve even smaller search spaces. Due to the additional computation effort to obtain time-dependent estimates, the observed running time improvements are not as significant as the reduction in search space size. Nonetheless, both time-dependent A* potentials also yield better running times than the *CCH-Potential*. In particular, applying the *Corridor-Lowerbound-Potential* results in the fastest query times throughout all runs and outperforms the *CCH-Potential* by up to an order of magnitude. The *Multi-Metric-Potential* improves the running time of the *CCH-Potential* on all geometric query sets. We observe little running time differences for uniform queries on large road networks, though.

Graph	Query Type	Zero-Pot [Dijkstra]		CCH-Pot		Multi-Metrics		Corridor-Lower	
		#nodes	time [ms]	#nodes	time [ms]	#nodes	time [ms]	#nodes	time [ms]
Lux-20	uni	28 066	4.73	1 510	0.38	647	0.21	335	0.19
	pop-uni	29 342	4.80	1 866	0.44	783	0.22	365	0.18
	geom	22 958	3.90	1 607	0.39	580	0.18	278	0.15
	pop-geom	23 797	3.98	2 021	0.48	760	0.22	324	0.16
Ger-06	uni	2 334 890	625.44	17 755	7.39	11 529	5.93	1 207	2.21
	pop-uni	2 257 570	597.48	19 807	8.22	12 735	6.39	1 228	2.21
	geom	219 624	69.07	2 054	1.26	830	1.02	264	0.68
	pop-geom	238 797	59.01	2 418	1.10	946	0.89	284	0.62
Eur-17	uni	12 499 218	3 388.94	339 051	175.86	333 303	164.05	7 317	13.32
	pop-uni	12 447 810	3 160.42	337 710	142.39	329 812	146.81	7 023	10.90
	geom	324 376	75.28	2 923	1.30	1 174	1.46	398	1.03
	pop-geom	289 995	69.24	4 082	1.78	1 675	1.70	425	1.06
Eur-20	uni	14 403 898	5 024.51	316 978	167.01	310 894	172.41	11 284	17.34
	pop-uni	13 405 330	4 623.20	316 365	231.62	308 246	243.07	10 996	18.69
	geom	326 601	105.03	5 905	2.77	2 584	2.20	520	1.26
	pop-geom	413 013	138.03	7 782	5.00	3 438	3.39	590	1.57

Table 5.3: Average running time and search space size of the A* algorithm by applying different potentials. Each query set contains 10 000 randomly generated queries.

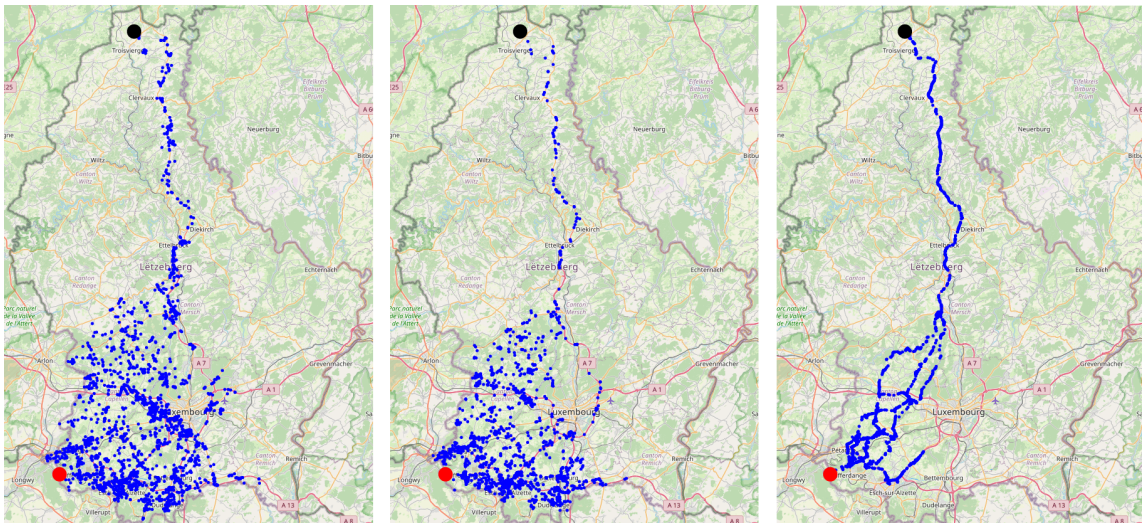


Figure 5.1: Search space visualization of a query between the red and black circle. Each blue circle represents a visited node. Left: *CCH-Potential*, mid: *Multi-Metric-Potential*, right: *Corridor-Lowerbound Potential*. Only 10% of the visited nodes are displayed for the first two approaches.

Figure 5.1 visualizes the search space of a query on the road network of Luxembourg. For better representation, only 10% of the nodes scanned by the A* search are displayed in the left and middle images. To obtain the fastest route between the nodes marked in red and black, the *CCH-Potential* yields a search space size of around 21 000 nodes. Applying the *Multi-Metric-Potential* reduces the number of visited nodes to 14 000. By far the smallest search space size is achieved by using the *Corridor-Lowerbound-Potential*. In this run, only 2 000 nodes have been visited by the A* algorithm, 90% less compared to the CCH-Potential.

We have also examined the accuracy of the potential estimates. The relative deviation $dev(\cdot)$ between the actual distance $d(s, t, \tau^{dep})$ and the potential estimate $\pi_{s,t,\tau^{dep}}(s, \tau^{dep})$ is given by the equation

$$dev(s, t, \tau^{dep}) = 1 - \frac{\pi_{s,t,\tau^{dep}}(s, \tau^{dep})}{d(s, t, \tau^{dep})}.$$

The deviations have been measured for the *uniform* and *geometric* query sets on the *Eur-20* graph. As expected, the largest deviation between the heuristic estimate and the actual distance is observed for the *CCH-Potential*. It averages 3.64% for the *uniform* and 8.97% for the *geometric* queries. In particular, the estimates are less precise for trips during rush hours. The estimates of the *Multi-Metric-Potential* deviate by 3.43% for both query types. The balanced result is a consequence of instantiating many metrics that cover time intervals in expected peak traffic hours. By far the tightest estimates are provided by the *Corridor-Lowerbound-Potentials*. In this case, the average deviations range from 0.37% on the *uniform* to 0.57% on the *geometric* query set. We can conclude that approximating the customized edge weights into k equally sized time intervals does not pose a risk of inaccurate predictions. Instead, it enables significant performance optimization for the potential retrieval.

Dijkstra-Ranks. The results in Table 5.3 provide average running times over 10 000 runs. Therefore, they do not highlight the difference between short-range and long-range queries. To distinguish between these types, we also evaluate the potentials on *Dijkstra-Rank* queries. The experiment has been conducted on the *Eur-20* instance. We have processed 1 000 queries for each rank $r \in \{2^8, 2^9, \ldots, 2^{24}\}$. Both source nodes and departure times of these

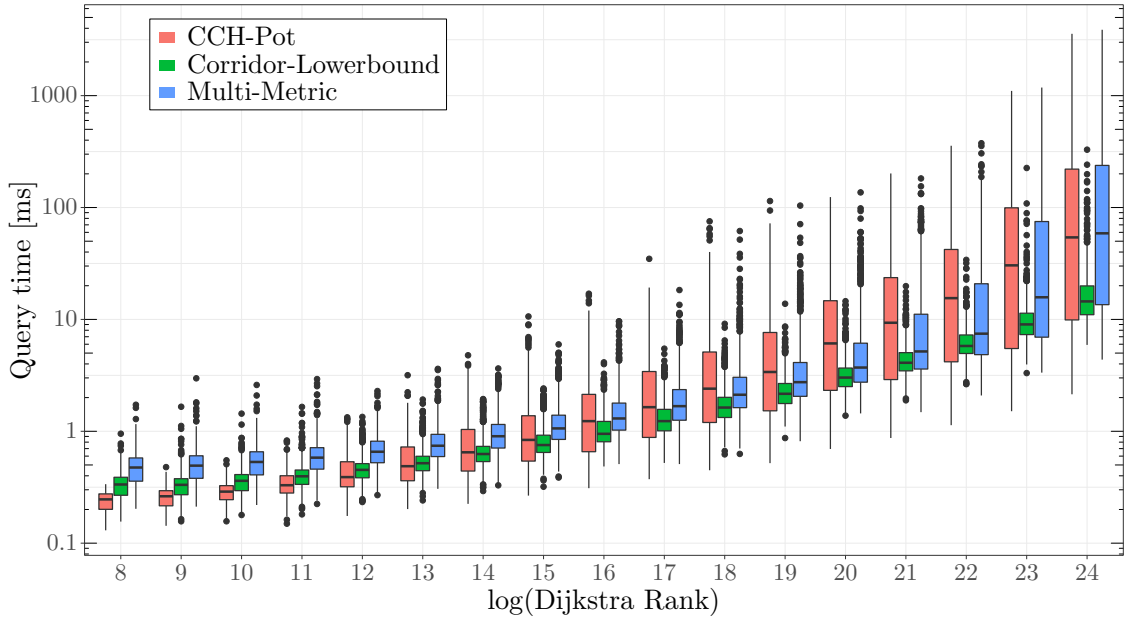


Figure 5.2: Query time as a function of the *Dijkstra Rank*.

queries are drawn randomly. The running times are presented in Figure 5.2, grouped by A* potential and Dijkstra rank. Each colored box represents the *interquartile range* (IQR). An additional mark is set for the median running time. The whisker boundaries outside of the boxes represent $1.5 \times IQR$ intervals. Furthermore, outliers below and above the whiskers are marked as separate dots.

For all ranks up to 2^{14}, the *CCH-Potential* has the lowest reported median running times. Especially for short-range queries, we can deduce that the comparably low initialization effort compensates for the larger search space of the A* algorithm. As the distance between source and target increases, the advantages of time-dependent A* potentials come into play. From rank 2^{14} onward, the *Corridor-Lowerbound-Potential* provides the fastest running times. Simultaneously, the outliers of the *CCH-Potential* and *Multi-Metric-Potential* have a considerable impact on their average query times. For long-range queries, the *Corridor-Lowerbound-Potential* outperforms the competitors significantly. The largest differences between the median query times are reported at the last rank 2^{24}. Throughout all ranks, we can also observe that the variance of the running times is relatively low for the *Corridor-Lowerbound-Potential*. Hence, this heuristic provides good estimates for all types of queries. Apart from that, the *Multi-Metric-Potential* outperforms the *CCH-Potential* between the ranks $[2^{17}, 2^{23}]$. At rank 2^{24}, the running times of these approaches converge with each other. The reason is that the *Multi-Metric-Potential* has to fall back to the lower-bound metric $\underline{len}$ for many long-range queries. Therefore, the heuristics provide identical estimates. In general, however, the *Multi-Metric-Potential* provides tighter estimates and makes the A* algorithm less prone to variations in the running time.

Up to this point, we can conclude that the *Multi-Metric-Potential* and *Corridor-Lowerbound-Potential* incorporate time-dependence efficiently. Throughout different query sets, both approaches outperform the baseline *CCH-Potential* on road networks with real-world traffic predictions. However, especially the *Corridor-Lowerbound-Potential* requires long preprocessing times and causes a considerable memory utilization. This can be troublesome if the application requires frequent adjustments to the current traffic situation. Furthermore, we have observed that the *Multi-Metric-Potential* offers a trade-off between the other approaches. A manageable preprocessing time is sufficient to enable relatively tight

Graph	#Queries	Single-Bucket		50 Buckets		200 Buckets	
		Usage [%]	Space [MB]	Usage [%]	Space [MB]	Usage [%]	Space [MB]
Berlin	500k	77.09	25.17	35.27	226.69	22.82	547.46
	1.5M	84.26	25.23	50.57	301.55	36.10	828.03
	3M	87.81	25.40	61.00	348.41	47.21	1 041.49
BaWü	500k	37.88	400.96	11.31	2 105.83	7.23	4 873.83
	1M	45.56	408.66	14.81	2 632.09	10.16	6 569.35
	2M	53.48	416.80	19.39	3 320.48	14.00	8 811.98

Table 5.4: Space consumption of multiple Capacity-Graphs with a varying bucket count. The relative usage refers to the share of edge buckets with a traffic load greater than zero.

potential estimates. Therefore, the choice of the most suitable A* heuristic depends on the application area. In the following series of experiments, we evaluate the applicability of our time-dependent A* potentials for cooperative route planning.

5.4 Evaluating Cooperative Route Planning

We finally evaluate our model of cooperative route planning. In the first experiment, we determine the memory consumption of several Capacity-Graphs initialized with different bucket counts. Then, the performance of our A* potentials is examined on the Capacity-Graphs. We also evaluate the solution quality of the on-line distribution algorithm described in Section 3.2. For this purpose, we provide *selfish* routing algorithms for comparison.

To model realistic traffic loads on the Capacity-Graphs, it is necessary to generate and process millions of requests. This can be problematic for two reasons. First, creating large sets of *geometric* queries is time-consuming because every single request involves a Dijkstra run. Moreover, it also takes several hours to solve these queries for each heuristic separately. As a workaround, we use *uniform* queries instead. Although these are less realistic, uniform queries are fast to generate. They also cover longer distances, whereby congestion on the main roads can be modeled with fewer requests. Apart from this, we must ensure that the A* heuristics provide *lower-bound* estimates throughout all runs. In the given cooperative model, the edge weights are slightly increased after each query but never decreased. For both of our time-dependent heuristics, it has to be guaranteed that the extracted upper-bound metric $\overline{len}$ remains valid. We can exploit that it is sufficient to update $\overline{len}$ upon a bound violation. The reason is that the potentials satisfy the *lower-bound* property as long as the bounds are valid. To avoid too frequent adjustments, we increase all entries in $\overline{len}$ by 50%. The main preprocessing routine is still run regularly to enable tight potential estimates.

Memory Consumption. In the first experiment related to cooperative route planning, we evaluate the memory consumption of several Capacity-Graphs. On each OSM road network, we compare graphs with 1, 50, and 200 buckets per edge. The results are shown in Table 5.4. We have captured the memory usage as well as the share of *used* edge buckets, i. e. those with a non-zero traffic load. For the single-bucket case, the relative usage corresponds to the share of road segments used in at least one route. Besides that, it should be noted that applying a speed-up technique causes additional storage overhead. Our measurements do not take into account such auxiliary data.

We have processed three million queries on the OSM graph of Berlin. Although common traffic flows within the city consist of fewer trips [FG19], this scenario poses a challenge for both the distribution algorithm and the underlying A* heuristic. We observe that 500k

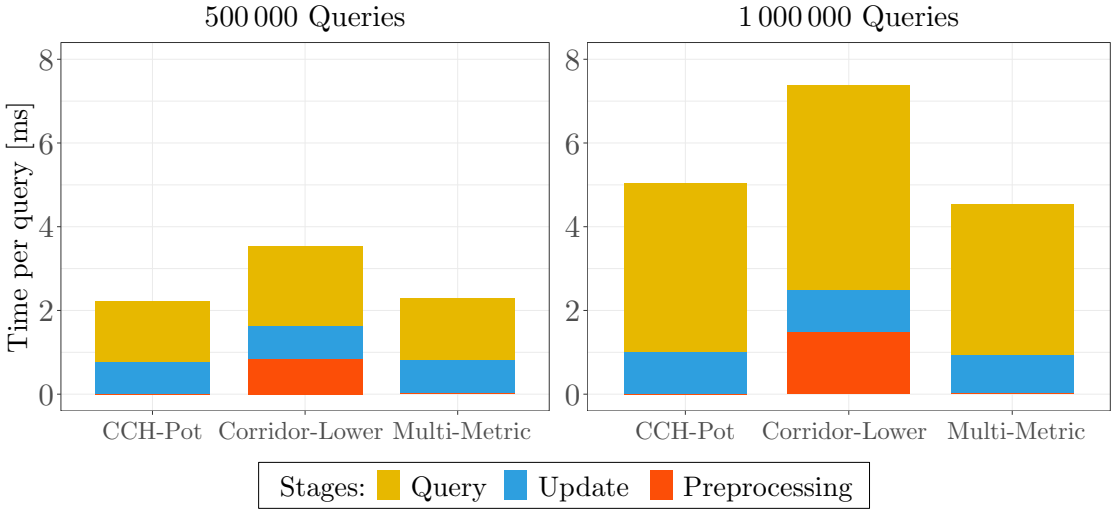


Figure 5.3: Comparison of the running times of the A* potentials on the instance of Berlin. The bars are divided into the different stages *Preprocessing*, *Query* and *Update*.

uniform queries suffice to cover large parts of the graph. After three million requests, most of the road segments are considered in at least one route. We assume that the remaining edges refer to irrelevant road segments, such as dirt tracks or falsely modeled roads. Besides, we can see that the bucket structure dominates the overall memory consumption. Compared to the single-bucket Capacity-Graph, maintaining up to 200 buckets per edge can increase the required storage by a factor of 40. Nonetheless, memory usage grows sublinearly with the applied bucket count. We can conclude that storing only used edge buckets (as described in Section 3.2) is advantageous in practice.

On the graph of Baden-Württemberg, two million requests have been run. Due to the choice of *uniform* queries, most trips cover large distances and thus mainly congest the highways and federal roads. Compared to the measurements on the road network of Berlin, the share of used edges is therefore lower. As we have seen before, a high degree of time-dependence implies considerable memory consumption. Here, the bucket structures alone account for up to 8GB of used memory. This underlines the importance of choosing the bucket count k reasonably. Especially for continental-sized graphs, a large k could be troublesome. It is necessary to achieve a good trade-off between the accuracy of the time-dependent graph model on the one hand and the memory consumption associated to the edge buckets on the other.

Query Times. Next, we compare *CCH-Potential*, *Multi-Metric-Potential*, and *Corridor-Lowerbound-Potential* in a cooperative setting. Contrary to the experiment conducted in Section 5.3, we now apply an *update* step which adjusts the edge weights after each route assignment. Throughout all runs, we use Capacity-Graphs with 100 buckets per edge. To ensure the correctness of the time-dependent A* heuristics, the upper-bound metric $\overline{len}$ must be validated in each step. Moreover, we re-run the preprocessing step regularly for both *Multi-Metric-Potential* and *Corridor-Lowerbound-Potential*. This enables frequent adjustments to the current traffic situation and therefore improves the accuracy of the estimates.

We have measured the running times of the A* heuristics on the graph of Berlin. In this experiment, two *uniform* query sets with half a million and a million requests are used. The preprocessing step of the *Multi-Metric-Potential* and the *Corridor-Lowerbound-Potential* is run after 50 000 queries each. Additional adjustments to the upper-bound metrics are applied whenever needed. Figure 5.3 presents the average query times. In the chart,

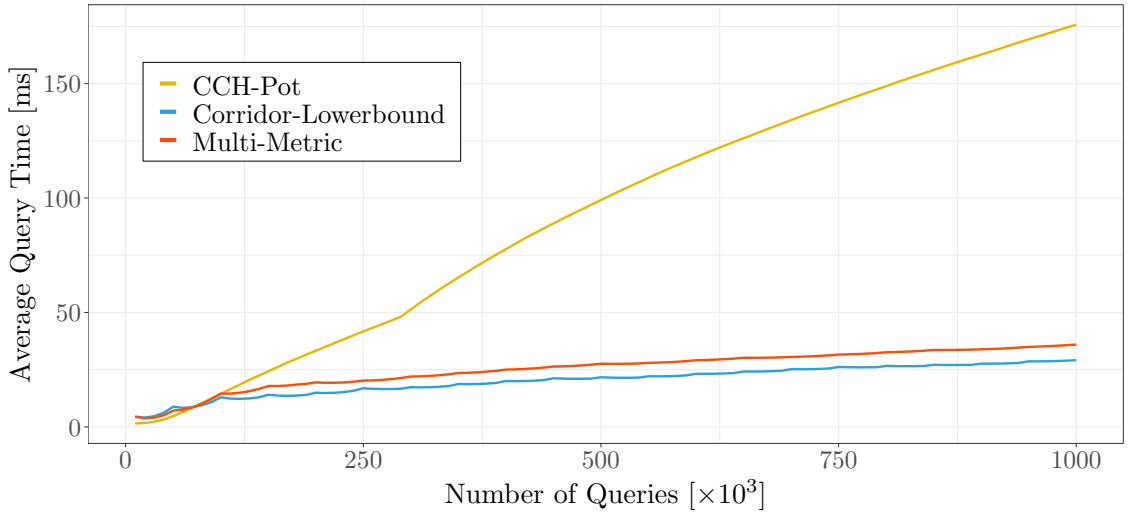


Figure 5.4: Running times on *unsorted* query sets, measured on the graph of Baden-Württemberg.

we subdivide the bars by the different query stages. Besides query and update time for each request, we also consider the amortized running time of the preprocessing phase. As expected, the update time is consistent for all approaches. Moreover, we obtain significant variations in the preprocessing times. Due to the small size of the graph, performing a time-independent CCH customization takes less than a second. Therefore, the amortized preprocessing effort is negligible for both *CCH-Potential* and *Multi-Metric-Potential*. In contrast, each preprocessing step of the *Corridor-Lowerbound-Potential* can take up to a minute.

It seems surprising that all A* potentials solve the queries equally fast. The *Multi-Metric-Potential* yields the fastest query times, followed by the *CCH-Potential* and the *Corridor-Lowerbound-Potential*. However, none of our time-dependent heuristics provide significant speedups as measured before. This observation reveals a fundamental issue of the studied cooperative model: If we process the queries by ascending departure time, only a fraction of the available information is related to future traffic flows. The preprocessing step mainly processes past traffic data instead. As a result, the estimates of our time-dependent A* potentials differ only marginally from the *CCH-Potential*. Unfortunately, there are no simple measures to deal with this problem. Running the preprocessing step more frequently to incorporate additional traffic information is not practical for large graphs as it will further degrade the overall performance.

To demonstrate that time-dependent A* potentials can still outperform the *CCH-Potential* in a cooperative setting with permanently changing edge weights, we repeat the experiment with *unsorted* query sets. We explicitly do not order the queries by departure time. Although this setup does not resemble a real-world use case, it allows us to incorporate more information about future traffic conditions. Figure 5.4 visualizes the running times after processing one million queries on the road map of Baden-Württemberg. This time, we aggregate the running times of preprocessing, query, and update phase. The measurements take place after 10 000 requests each. While the average running time of the *CCH-Potential* grows linearly with the number of processed queries, the time-dependent A* potentials provide tighter estimates throughout the entire run. We also observe that intermediate updates of the bounds do not pose a performance risk. Therefore, we can conclude that time-dependent A* potentials are also applicable for cooperative route planning. However, sufficient information on future traffic flows must be available.

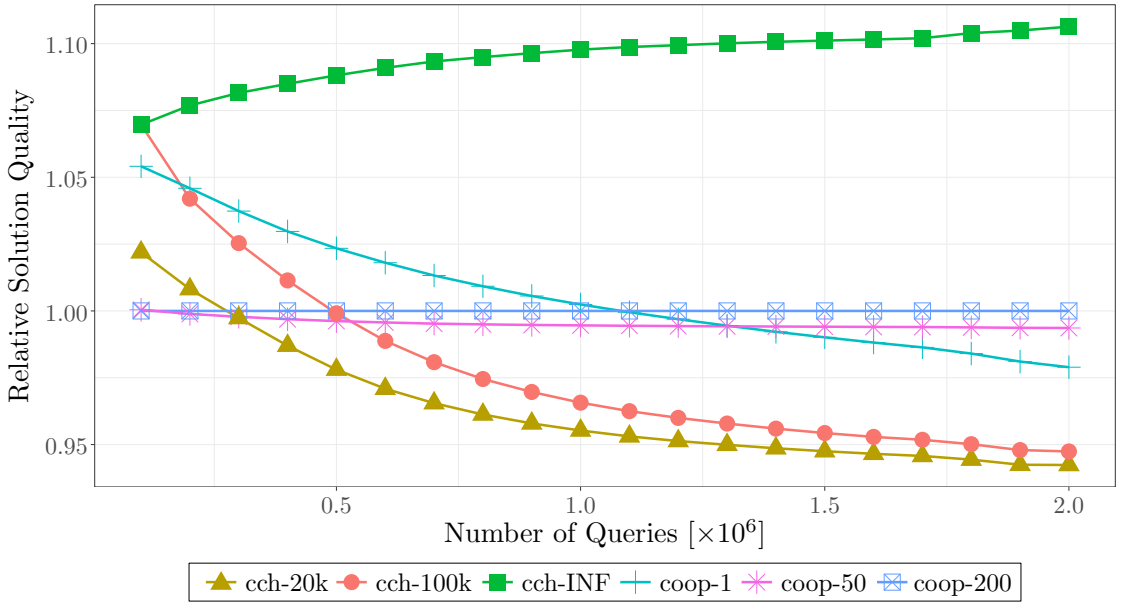


Figure 5.5: Evaluating the solution quality of different distribution strategies (*cooperative* vs. *selfish*). The resulting total travel time is compared with the benchmark instance *coop-200*.

Solution Quality. Finally, we evaluate the solution quality of our cooperative distribution algorithm. In particular, we investigate whether our applied strategy distributes traffic evenly and minimizes the overall travel time. We compare our cooperative approach with *selfish* methods that do not adjust the edge profiles in each step and aim only at optimizing the result for the current query. The experiment is conducted as follows. First, each approach determines the routes for the given requests. Then, the travel times of the assigned routes are re-evaluated on a benchmark instance, i. e. a Capacity-Graph with adjusted edge weights. We obtain the solution quality of each approach from the sum over the corresponding travel times of all proposed routes.

The evaluation is conducted on three Capacity-Graphs with different bucket counts (1, 50, and 200, respectively). This allows us to investigate the impact of time-dependence, too. We use the latter instance as a benchmark for the subsequent comparisons. The selfish routing instances apply a CCH to accelerate the computations. We regularly customize the CCH graphs with the current travel times obtained from the benchmark instance. For simplification, we refrain from instantiating a selfish approach with time-dependent edge weights. Instead, the travel time functions are evaluated at the departure timestamp of the current query.

Figure 5.5 shows the results of the comparison. We have run two million queries (ordered by departure time) on the graph of Baden-Württemberg and conducted a comparison after 100k steps each. The cooperative instances are labeled as *coop-x*, where x refers to the bucket count in the respective Capacity-Graph. Moreover, the selfish approaches (*cch-x*) are customized after every x requests. We also take into account an instance that is never synchronized (*cch-INF*). The y-Axis of the figure represents the *relative solution quality* compared to the benchmark instance (*coop-200*). This value is retrieved by dividing the total travel times of the respective approach by the benchmark result. Therefore, the solution quality of the benchmark instance is always equal to 1. As we consider a minimization problem, results below 1 indicate improvements over the benchmark instance. Conversely, a higher quotient implies a poorer solution quality.

The comparatively poor solution quality of the cooperative approaches contradicts our original expectations. After processing two million queries, we observe that the selfish routing approaches provide even better routes. Moreover, using Capacity-Graphs with a low degree of time-dependence seems beneficial in the considered cooperative model. As seen in the last experiment, our cooperative model has limited information about future traffic flows. This affects both the running time of the A* algorithm and the quality of the obtained routes. In addition, we have considered *uniform* queries that are distributed evenly over the entire day. Therefore, predictions based on the current traffic situation are suddenly more precise. This leads to the paradoxical case that the CCH-based selfish approaches provide better routes. Compared to their cooperative competitors, the total travel time is improved by around 5%. For the same reason, a higher degree of time-dependence is also disadvantageous in the given scenario. The fewer edge buckets the Capacity-Graph has, the more likely it is to incorporate past and current traffic data in the current query.

Nevertheless, these results do not imply that cooperative route planning is inferior to selfish routing. We also observe that the cooperative approaches perform far better than the selfish instance *cch-INF* which does not apply any traffic updates at all. Therefore, we emphasize that further adjustments to the studied model are necessary to exploit the benefits of cooperative routing to its full extent. In particular, more information about future traffic conditions must be available. This could e. g. be achieved by incorporating historic data. The stored traffic loads can then be used to detect unusual traffic peaks in advance. Another solution could be to allow subsequent adjustments of the routes. As we have already seen, time-dependent A* potentials perform comparatively well if future traffic flows are predictable. We assume that this also improves the quality of the suggested routes. To this end, we encourage revising the cooperative model and the provided on-line traffic distribution algorithm.

6. Conclusion

We have introduced and discussed efficient algorithms for cooperative route planning on time-dependent graphs. For this, we have studied a model in which the queries are processed on-line, ordered by ascending departure time. A simple strategy has been applied to distribute traffic evenly in the road network. Our main focus has been on designing time-dependent A* potentials to accelerate the search for shortest paths.

As a main result, our proposed A* potentials outperform the current state-of-the-art *CCH-Potential* [SZ21] by up to an order of magnitude in terms of query time. Both *Multi-Metric-Potential* and *Corridor-Lowerbound-Potential* incorporate time-dependence with manageable computation effort during the query phase. In addition, the two approaches also point out interesting algorithmic insights. As exploited in the *Multi-Metric-Potential*, we can obtain time-dependent A* heuristics by applying a time-independent speed-up technique. These techniques usually require shorter preprocessing times than their time-dependent counterparts. Hence, the potential is particularly suitable for applications where fast customization is necessary. Besides that, A* potentials which violate the *feasibility* property are not necessarily less efficient. Although the A* algorithm may visit some nodes several times, this allows us to apply further optimizations. For example, we approximate travel time functions in the *Corridor-Lowerbound-Potential* to achieve significant performance gains.

The proposed time-dependent potentials have different strengths and weaknesses. On the one hand, the *Corridor-Lowerbound-Potential* provides tight estimates and enables the fastest running times in the query phase. On the other hand, it uses a time-dependent CCH as the underlying speed-up technique. This leads to slow preprocessing times and considerable memory usage. In contrast to this, the *Multi-Metric-Potential* offers a trade-off between the flexibility of the *CCH-Potential* and the accuracy of the *Corridor-Lowerbound-Potential*. Although its estimates are not as precise as the *Corridor-Lowerbound-Potential*, the preprocessing step runs comparatively fast. Moreover, it also outperforms the *CCH-Potential* for most query types.

Our experiments also show that the considered model for cooperative route planning requires further investigations. In particular, the simple distribution algorithm is insufficient to provide good routes on congested road networks. For random queries, we have seen that processing the current traffic situation yields better results than using the prediction of future traffic conditions. This underlines the need to revise the model itself and incorporate more traffic information.

6.1 Future Work

As our experiments have shown, the application scope of time-dependent A* potentials is not limited to cooperative route planning. In general, the potentials are suitable for applications in which the edge weights of the graph are subject to frequent changes. Moreover, real-world applications must consider further modelling aspects, such as *turn costs* or the distinction between *live* and *predicted* traffic data [SZ21]. Integrating these aspects into hierarchical speed-up techniques can be tedious. However, by decoupling the speed-up technique from the input graph, we can exploit A* potentials to achieve this with less effort.

We assume that the performance of both proposed time-dependent A* potentials can yet be improved. For completeness, a sensitivity analysis related to the number of metrics (*Multi-Metric-Potential*) and the number of approximated intervals (*Corridor-Lowerbound-Potential*) should be performed. Some other setup parameters should be reconsidered, too. For example, the preprocessing effort of our time-dependent heuristics is comparatively low for small graphs. Therefore, the underlying CCH can be synchronized more frequently with the current traffic situation. Moreover, we suggest further design improvements related to the *Multi-Metric-Potential*. In particular, the super-quadratic running time of the metric reduction step constitutes the main performance bottleneck during the preprocessing. The remaining parts take less than a minute, even on continental-sized graphs. Regarding the *Corridor-Lowerbound-Potential*, it is possible to reduce the preprocessing time by skipping some optional stages of the time-dependent customization at the expense of additional computations during the query phase. However, this measure will only marginally reduce the preprocessing time and therefore does not solve the main drawback of this approach.

Last but not least, we discuss the applicability of cooperative route planning. To integrate the studied model into real-world applications, further aspects must be addressed. We suggest considering historical traffic data when predicting future traffic flows. Then, we can adjust the expected travel times by route information obtained from other road users. This is particularly useful during unexpected situations such as road accidents or major events. Besides that, it is naive to assume that the suggested route remains optimal over the whole trip. Subsequent route adjustments offer the advantage of taking into account additional traffic information. Applied to our model, this implies that the traffic loads stored in the edge buckets can both increase and decrease. This poses further challenges for the design of valid potential functions. Lastly, it would also be interesting to compare the paths obtained by the cooperative model with the *social optimum* state. This would allow us to quantify the inevitable loss of solution quality due to the real-time route assignments to each request.

Overall, several issues have to be resolved before our cooperative route planning approach can be integrated into real-world applications. However, the use of a central authority that distributes traffic evenly in real-time could lead to a noticeable easing of the traffic situation in urban areas. Nonetheless, incentives must be provided to ensure that all drivers follow their assigned routes. The success of cooperative routing ultimately depends on the extent to which the road users are willing to cooperate rather than pursue their interests.

Bibliography

[ADGW11] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. A Hub-Based Labeling Algorithm for Shortest Paths in Road Networks. In *Experimental Algorithms - 10th International Symposium, SEA 2011, Kolimpari, Chania, Crete, Greece, May 5-7, 2011. Proceedings*, volume 6630 of *Lecture Notes in Computer Science*, pages 230–241. Springer, 2011. https://doi.org/10.1007/978-3-642-20662-7_20.

[ALS13] Julian Arz, Dennis Luxen, and Peter Sanders. Transit Node Routing Reconsidered. In *Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings*, volume 7933 of *Lecture Notes in Computer Science*, pages 55–66. Springer, 2013. https://doi.org/10.1007/978-3-642-38527-8_7.

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows - Theory, Algorithms and Applications*. Prentice Hall, 1993.

[BCRW16] Reinhard Bauer, Tobias Columbus, Ignaz Rutter, and Dorothea Wagner. Search-space size in contraction hierarchies. *Theoretical Computer Science*, 645:112–127, 2016. https://doi.org/10.1016/j.tcs.2016.07.003.

[BDG+16] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. *Route Planning in Transportation Networks*, volume 9220 of *Lecture Notes in Computer Science*, pages 19–80. Springer International Publishing, Cham, 2016. https://doi.org/10.1007/978-3-319-49487-6_2.

[Bel57] Richard E. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[BGSV13] Gernot V. Batz, Robert Geisberger, Peter Sanders, and Christian Vetter. Minimum Time-Dependent Travel Times with Contraction Hierarchies. *ACM Journal of Experimental Algorithmics*, 18, 2013. https://doi.org/10.1145/2444016.2444020.

[BMW56] Martin Beckmann, Charles B. McGuire, and Christopher B. Winsten. Studies in the Economics of Transportation. Technical report, Yale University Press, 1956.

[BNW05] Dietrich Braess, Anna Nagurney, and Tina Wakolbinger. On a Paradox of Traffic Planning. *Transportation Science*, 39(4):446–450, 2005. https://doi.org/10.1287/trsc.1050.0127.

[BoPR64] Bureau of Public Roads. Traffic Assignment Manual. *US Department of Commerce*, 1964.

[BSW19a] Valentin Buchhold, Peter Sanders, and Dorothea Wagner. Efficient Calculation of Microscopic Travel Demand Data with Low Calibration Effort. In *Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, SIGSPATIAL 2019, Chicago, IL, USA, November 5-8, 2019*, pages 379–388. ACM, 2019. https://doi.org/10.1145/3347146.3359361.

[BSW19b] Valentin Buchhold, Peter Sanders, and Dorothea Wagner. Real-time Traffic Assignment Using Engineered Customizable Contraction Hierarchies. *ACM Journal of Experimental Algorithmics*, 24(1):2.4:1–2.4:28, 2019. https://doi.org/10.1145/3362693.

[CH66] Kenneth L. Cooke and Eric Halsey. The Shortest Route Through a Network with Time-Dependent Internodal Transit Times. *Journal of Mathematical Analysis and Applications*, 14(3):493–498, 1966. https://doi.org/10.1016/0022-247X(66)90009-6.

[Dea04] Brian C. Dean. Shortest Paths in FIFO Time-Dependent Networks: Theory and Algorithms. Technical report, Massachusetts Institute of Technology, 2004.

[Del09] Daniel Delling. *Engineering and Augmenting Route Planning Algorithms*. PhD thesis, Karlsruhe Institute of Technology, 2009. http://digbib.ubka.uni-karlsruhe.de/volltexte/1000011046.

[DGNW13] Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. PHAST: Hardware-accelerated shortest path trees. *Journal of Parallel and Distributed Computing*, 73(7):940–952, 2013. https://doi.org/10.1016/j.jpdc.2012.02.007.

[DGPW17] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable Route Planning in Road Networks. *Transportation Science*, 51(2):566–591, 2017. https://doi.org/10.1287/trsc.2014.0579.

[DGW11] Daniel Delling, Andrew V. Goldberg, and Renato Fonseca F. Werneck. Faster Batched Shortest Paths in Road Networks. In *ATMOS 2011 - 11th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems, Saarbrücken, Germany, September 8, 2011*, volume 20 of *OASIcs*, pages 52–63. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2011. https://doi.org/10.4230/OASIcs.ATMOS.2011.52.

[Dij59] Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959. https://doi.org/10.1007/BF01386390.

[DP73] David H. Douglas and Thomas K. Peucker. Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line or its Caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 10(2):112–122, 1973. https://doi.org/10.3138/FM57-6770-U75U-7727.

[DSW14] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable Contraction Hierarchies. In *Experimental Algorithms - 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29 - July 1, 2014. Proceedings*, volume 8504 of *Lecture Notes in Computer Science*, pages 271–282. Springer, 2014. https://doi.org/10.1007/978-3-319-07959-2_23.

[DW09] Daniel Delling and Dorothea Wagner. Time-Dependent Route Planning. In *Robust and Online Large-Scale Optimization: Models and Techniques for Transportation Systems*, volume 5868 of *Lecture Notes in Computer Science*, pages 207–230. Springer, 2009. https://doi.org/10.1007/978-3-642-05465-5_8.

[FG19] Robert Follmer and Dana Gruschwitz. Mobility in Germany - short report, 2019. Edition 4.0 of the study by infas, DLR, IVT and infas 360 on behalf of the Federal Ministry of Transport and Digital Infrastructure (BMVI) (FE no. 70.904/15). Bonn, Berlin. www.mobilitaet-in-deutschland.de.

[FH95] Michael Florian and Donald Hearn. Network Equilibrium Models and Algorithms. In *Network Routing*, volume 8 of *Handbooks in Operations Research and Management Science*, pages 485–550. Elsevier, 1995. https://doi.org/10.1016/S0927-0507(05)80110-0.

[FHS11] Luca Foschini, John Hershberger, and Subhash Suri. On the Complexity of Time-Dependent Shortest Paths. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2011, San Francisco, California, USA, January 23-25, 2011*, pages 327–341. SIAM, 2011. https://doi.org/10.1137/1.9781611973082.27.

[Flo62] Robert W. Floyd. Algorithm 97: Shortest Path. *Communications of the ACM*, 5(6):345, 1962. https://doi.org/10.1145/367766.368168.

[FW56] Marguerite Frank and Philip Wolfe. An algorithm for quadratic programming. *Naval Research Logistics Quarterly*, 3(1-2):95–110, 1956. https://doi.org/10.1002/nav.3800030109.

[Geo73] Alan George. Nested Dissection of a Regular Finite Element Mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973. https://doi.org/10.1137/0710032.

[GH05] Andrew V. Goldberg and Chris Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, Vancouver, British Columbia, Canada, January 23-25, 2005*, pages 156–165. SIAM, 2005.

[GHUW19] Lars Gottesbüren, Michael Hamann, Tim N. Uhl, and Dorothea Wagner. Faster and Better Nested Dissection Orders for Customizable Contraction Hierarchies. *Algorithms*, 12(9):196, 2019. https://doi.org/10.3390/a12090196.

[GSSD08] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *Experimental Algorithms, 7th International Workshop, WEA 2008, Provincetown, MA, USA, May 30-June 1, 2008, Proceedings*, volume 5038 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2008. https://doi.org/10.1007/978-3-540-68552-4_24.

[HMT17] Márton T. Horváth, Tamás Mátrai, and János Tóth. Route Planning Methodology with Four-step Model and Dynamic Assignments. *Transportation Research Procedia*, 27:1017–1025, 2017. https://doi.org/10.1016/j.trpro.2017.12.127.

[HNR68] 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. https://doi.org/10.1109/TSSC.1968.300136.

[II88] Hiroshi Imai and Masao Iri. Polygonal Approximations of a Curve - Formulations and Algorithms. In *Computational Morphology*, volume 6 of *Machine Intelligence and Pattern Recognition*, pages 71–86. Elsevier, 1988. https://doi.org/10.1016/B978-0-444-70467-2.50011-4.

[KZT20] Alexander Krylatov, Victor Zakharov, and Tero Tuovinen. Principles of Wardrop for Traffic Assignment in a Road Network. In *Optimization Models and Methods for Equilibrium Traffic Assignment*, pages 17–43. Springer International Publishing, Cham, 2020. https://doi.org/10.1007/978-3-030-34102-2_2.

[LGT03] Maxim Likhachev, Geoffrey J. Gordon, and Sebastian Thrun. ARA*: Anytime A* with Provable Bounds on Sub-Optimality. In *Advances in Neural Information Processing Systems 16 [Neural Information Processing Systems, NIPS 2003, December 8-13, 2003, Vancouver and Whistler, British Columbia, Canada]*, pages 767–774. MIT Press, 2003.

[LPBM17] Thomas Liebig, Nico Piatkowski, Christian Bockermann, and Katharina Morik. Dynamic Route Planning with Real-Time Traffic Predictions. *Information Systems*, 64:258–265, 2017. https://doi.org/10.1016/j.is.2016.01.007.

[LS11] Dennis Luxen and Peter Sanders. Hierarchy Decomposition for Faster User Equilibria on Road Networks. In *Experimental Algorithms - 10th International Symposium, SEA 2011, Kolimpari, Chania, Crete, Greece, May 5-7, 2011. Proceedings*, volume 6630 of *Lecture Notes in Computer Science*, pages 242–253. Springer, 2011. https://doi.org/10.1007/978-3-642-20662-7_21.

[MM14] Enock T. Mtoi and Ren Moses. Calibration and Evaluation of Link Congestion Functions: Applying Intrinsic Sensitivity of Link Speed as a Practical Consideration to Heterogeneous Facility Types Within Urban Network. *Journal of Transportation Technologies*, 4:141–149, 2014. https://doi.org/10.4236/jtts.2014.42014.

[Ord89] Ariel Orda. Traveling without Waiting in Time-Dependent Networks is NP-hard. *Manuscript*, 1989.

[PERW15] Olga Perederieieva, Matthias Ehrgott, Andrea Raith, and Judith Y. T. Wang. A Framework for and Empirical Study of Algorithms for Traffic Assignment. *Computers & Operations Research*, 54:90–107, 2015. https://doi.org/10.1016/j.cor.2014.08.024.

[Poh71] Ira Pohl. Bi-directional Search. In *Proceedings of the Sixth Annual Machine Intelligence Workshop*, volume 6, pages 124–140. Edinburgh University Press, 1971.

[Pot88] Alex Pothen. The Complexity of Optimal Elimination Trees. Technical report, Pennsylvania State University, Department of Computer Science, 1988.

[PZ01] Srinivas Peeta and Athanasios K. Ziliaskopoulos. Foundations of Dynamic Traffic Assignment: The Past, the Present and the Future. *Networks and Spatial Economics*, 1(3):233–265, 2001. https://doi.org/10.1023/A:1012827724856.

[RT02] Tim Roughgarden and Éva Tardos. How Bad Is Selfish Routing? *Journal of the ACM*, 49(2):236–259, 2002. https://doi.org/10.1145/506147.506153.

[SN20] Arne Schneck and Klaus Nökel. Accelerating Traffic Assignment with Customizable Contraction Hierarchies. *Transportation Research Record*, 2674(1):188–196, 2020. https://doi.org/10.1177/0361198119898455.

[SS05] Peter Sanders and Dominik Schultes. Highway Hierarchies Hasten Exact Shortest Path Queries. In *Algorithms - ESA 2005, 13th Annual European Symposium, Palma de Mallorca, Spain, October 3-6, 2005, Proceedings*, volume 3669 of *Lecture Notes in Computer Science*, pages 568–579. Springer, 2005. https://doi.org/10.1007/11561071_51.

[SWZ21] Ben Strasser, Dorothea Wagner, and Tim Zeitz. Space-Efficient, Fast and Exact Routing in Time-dependent Road Networks. *Algorithms*, 14(3):90, 2021. https://doi.org/10.3390/a14030090.

[SZ21] Ben Strasser and Tim Zeitz. A Fast and Tight Heuristic for A* in Road Networks. In *19th International Symposium on Experimental Algorithms, SEA 2021, June 7-9, 2021, Nice, France*, volume 190 of *LIPIcs*, pages 6:1–6:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. https://doi.org/10.4230/LIPIcs.SEA.2021.6.

[TB10] Ozan K. Tonguz and Mate Boban. Multiplayer games over Vehicular Ad Hoc Networks: A new application. *Ad Hoc Networks*, 8(5):531–543, 2010. https://doi.org/10.1016/j.adhoc.2009.12.009.

[War52] John G. Wardrop. Road Paper. some Theoretical Aspects of Road Traffic Research. *Proceedings of the Institution of Civil Engineers*, 1(3):325–362, 1952. https://doi.org/10.1680/ipeds.1952.11259.

[WVLM11] David Wilkie, Jur P. Van den Berg, Ming C. Lin, and Dinesh Manocha. Self-Aware Traffic Route Planning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 25(1):1521–1527, 2011.

[ZMCS+19] Jorge L. Zambrano-Martinez, Carlos T. Calafate, David Soler, Lenin-Guillermo Lemus-Zúñiga, Juan-Carlos Cano, Pietro Manzoni, and Thierry Gayraud. A Centralized Route-Management Solution for Autonomous Vehicles in Urban Areas. *Electronics*, 8(7), 2019. https://doi.org/10.3390/electronics8070722.

[ZNN11] Michael Zilske, Andreas Neumann, and Kai Nagel. OpenStreetMap for Traffic Simulation. In *Proceedings of the 1st European state of the map: OpenStreetMap conference*, pages 126–134. OpenStreetMap Austria u.a., 2011. http://dx.doi.org/10.14279/depositonce-4679.