Karlsruhe Institute of Technology

Efficient Long-Haul Truck Driver Routing

Master Thesis of

Max Oesterle

At the Department of Informatics
Institute of Theoretical Informatics

Reviewers:	Dr. rer. nat. Torsten Ueckerdt
	?
Advisors:	Tim Zeitz
	Alexander Kleff
	Frank Schulz

Time Period: 15th January 2022 – 15th July 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, July 4, 2022

Abstract

A short summary of what is going on here.

Deutsche Zusammenfassung

Kurze Inhaltsangabe auf deutsch.

Contents

1. Introduction

In this work, we present an algorithm to solve the long-haul truck driver routing problem, an example of a real-world problem which constitutes an extension to the simple shortest path problem. TODO

1. introduction routing, spp
2. practical improvements
3. extensions of the spp similar to this thesis and arising problems
4. outline of algorithm and work on which is based
5. outline of thesis

2. Preliminaries

In this chapter, we will introduce our basic notation and discuss important algorithmic concepts on which the work of this thesis is based.

We define a weighted, directed graph G as a tuple $G = (V, E, \text{len})$. V is the set of vertices and E the set of edges $(u, v) \subseteq V \times V$ between those vertices. The function len is the weight function $\text{len}\colon E \to \mathbb{R}_{\geq 0}$ which assigns each edge a non-negative weight which we often also call length of an edge. A path p in G is defined as a sequence of nodes $p = \langle v_0, v_1, ..., v_k \rangle$ with $(v_i, v_{i+1}) \in E$. For simplicity, we will reuse the same function len which we use to denote the length of an edge, to denote the length of a path p in G. The length of a path $\text{len}(p)$ is defined by the sum of the weights of the edges on the path $\text{len}(p) = \sum_{i=0}^{k-1} \text{len}((v_i, v_{i+1}))$. A path must not necessarily be simple, i.e. nodes can appear multiple times in the same path.

Given two nodes s and t in a graph, we denote the shortest distance between them as $\mu(s, t)$. The shortest distance between two nodes is the minimum length of a path between them. The problem of finding the shortest distance between two nodes in a graph is called the shortest path problem which we often abbreviate as SPP. The problem of finding a fast path through a road network can be formalized as solving the SPP on a weighted graph. Each edge of the graph represents a road and each node represents an intersection. Unless stated otherwise, the length len of an edge (u, v) will always correspond to the time it takes to travel from u to v on the road which the edge represents. A solution of the SPP then yields the shortest time between two intersections in the road network and a path between them.

Dijkstra's algorithm, published in 1959, solves the SPP [Dij59]. It operates on the graph G without any additionally information or precomputed data structures. It maintains a queue Q of nodes with ascending tentative distance from the starting node s and two arrays, a distance value $d[v]$ and a predecessor node $pred[v]$ for each node. At the beginning, Q only contains the start node s with the distance zero. The two arrays are initialized with $d[v] = \infty$ and $pred[v] = \perp$ except for $d[s] = 0$ and $pred[s] = s$. Iteratively, the node u with the minimum distance is removed from Q and each outgoing edge $(u, v) \in E$ of u is *relaxed*. We call this process *settling* a node u. Relaxing an edge (u, v) consists of three steps: First, the sum $d[u] + \text{len}((u, v))$ is calculated. Second, it is tested if the distance $d[v]$ can be improved by choosing u as a predecessor. Finally, if that is the case, the queue key of the node v is decreased. If v is not contained in Q yet, the node is inserted into Q. The search can be stopped if the target node t was removed from the queue [Dij59].

For many practical applications and for large graphs, Dijkstra's algorithm is too slow. A common extension is the A* algorithm [HNR68]. A* uses a *heuristic* which yields a lower bound for the distance from each node to the target node to direct the search towards the goal. With a tight heuristic, A* can significantly reduce the search space, i.e., the amount of nodes it touches during the search in comparison to Dijkstra's algorithm. In the route planning context, the heuristic often is called *potential*. We will denote the potential of a node v to a node t with $\pi_t(v)$.

To further reduce the search space, it is possible to run a *bidirectional* search. A bidirectional search to solve the SPP from s to t on a graph G consists of a forward search and a backward search. The forward search operates on G with start node s and target node t and maintains a forward queue $\overrightarrow{Q}$, a forward distance array $\overrightarrow{d}[v]$, and a forward predecessor array $\overrightarrow{pred}[v]$. The backward search operates on a backward graph $\overleftarrow{G}$ with start node t and target node s and maintains a backward queue $\overleftarrow{Q}$, a backward distance array $\overleftarrow{d}[v]$, and a backward predecessor array $\overleftarrow{pred}[v]$. The backward graph is defined as the graph G with inverted edges, i.e, $\overleftarrow{G} = (V, \overleftarrow{E}, \overleftarrow{\text{len}})$ with $\overleftarrow{E} = \{(v,u) \in V \times V \mid (u,v) \in E\}$ and $\overleftarrow{\text{len}}(u,v) = \text{len}(v,u)$. Additionally, an array $d[v]$ is maintained for combined tentative distances of forward and backward search and is initialized with ∞ for all nodes. A value $d[v]$ constitutes the shortest known distance for an s-t path using v.

Forward and backward search now alternately settle a node v. If v was settled by both searches, forward and backward search met at v. The value $d[v]$ is updated to the sum of $\overrightarrow{d}[v] + \overleftarrow{d}[v]$ if it is an improvement over the old value, i.e., it yields a shorter distance for an s-t path via v.

The bidirectional search only yields an advantage over a unidirectional search if the two searches are stopped earlier than in a unidirectional search. If not, the bidirectional search would simply execute the work of an s-t search twice. Therefore, stronger stopping criteria are introduced. When introducing a strong stopping criterion for a bidirectional A* search, the stopping criterion also depends on the potential function being used. The work of [GH05] introduces a potential and stopping criterion which leads to an improvement over a unidirectional A* search. If a forward potential $\overrightarrow{\pi}_t$ and a backward potential $\overleftarrow{\pi}_s$ is used for the forward and the backward search, the search can be stopped when the minimum key of the forward queue $\text{minKey}(\overrightarrow{Q})$ or the minimum key of the backward queue $\text{minKey}(\overrightarrow{Q})$ is greater than the currently known shortest distance between s and t. If $\overrightarrow{\pi}_t + \overleftarrow{\pi}_s \equiv const.$ holds for the potentials, then the search can be stopped when $\text{minKey}(\overrightarrow{Q}) + \text{minKey}(\overleftarrow{Q})$ exceeds the shortest know distance between s and t.

2.1. Contraction Hierarchies

Road networks have strong hierarchies since some roads are more important than others. For example, a highway allows high average speeds and therefore is a preferred connection between points in a road network in comparison to smaller roads. Contraction Hierarchies [GSSV12] are a speed-up technique which exploits these hierarchies.

Contraction Hierarchies (CH) use a two phase approach. In the preprocessing phase, the CH is constructed given a graph $G = (V, E, \text{len})$ and a node order which sorts the nodes by importance. For example, an important node of a road network might be a node in a highway interchange, an unimportant node might be the head of a dead end. We denote the CH as G^+. The node order can be computed by searching for unimportant nodes [GSSV12] or for important nodes first [ADGW12, DGPW14]. The nodes then are sorted into multiple levels of increasing importance. Two nodes of the same level must not have an edge between them. We obtain the CH G^+ by iteratively contracting the least important node according to the node order by adding shortcut edges between its neighbors.

An outgoing edge of a node to another node of a higher level is called an *upward* edge, the opposite is called a *downward* edge. A path which consists of only upward edges is called an *up-path*, a path which consists of only downward edges is called a *down-path*. Finally, a path which consists of an up-path, followed by a down-path, is called an *up-down-path*. The node on an up-down-path with the highest level, i.e. the node which separates up-path and down-path, is called the *middle node* m of the path.

For every shortest path between node $s, t \in V$, there exists an up-down s-m-t path in G^+ of the exact same length [GSSV12]. Therefore, we can restrict the search to finding this exact path. We run a bidirectional search from s and t. The forward search from s may only use upward edges and finds the subpath s-m, the backward search from t may only use the inverted downward edges and finds the subpath m-t. The search is stopped if the minimum queue key of both queues of forward and backward search is greater or equal to the tentative minimum distance $\mu(s, t)$ [GSSV12].

2.2. Core Contraction Hierarchies

The core contraction hierarchy presents a compromise between constructing a full CH G^+ and running a bidirectional search on G. The core CH G^* is obtained by stopping the iterative contraction of nodes of increasing importance during the construction of the CH early. This leads to a set $C \subseteq V$ of so-called core nodes which are uncontracted. The set of nodes V therefore is separated into a set of core nodes C and a set of contracted nodes $V^+ = V \setminus C$. The graph $G^+ = (V^+, E^+, \text{len})$ therefore is a valid contraction hierarchy. It contains only contracted nodes and (shortcut) edges between those nodes. The graph $G_C = (V \setminus V^+, E \setminus E^+, \text{len})$ is called the core graph.

The core CH query again is a bidirectional search from s and t. The query represents a normal CH query while settling nodes $v \in V^+$, i.e., it consists of a forward search from s using only upward edges and a backward search from t using only inverted downward edges. If a search reaches an uncontracted core node, it considers all outgoing edges. Thus, the core CH query can be characterized as a CH query in G^+ which transitions into a full bidirectional search if it reaches the core G_C. We can use the same stopping criterion as for the CH query since the stopping criterion for a CH is more conservative than the stopping criterion for a pure bidirectional search.

2.3. CH-Potentials

CH-Potentials [SZ21] are an extension of CH to efficiently calculate distances $\mu(v, t)$ for many nodes $v \in V$ to a fixed node t. CH-Potentials are based on PHAST [DGNW11] which itself is an extension of CH to efficiently run all-to-one queries.

PHAST runs in two steps. The first step is a backward one-to-all search from t using only inverted downward edges. This is the same as running the backward search of a CH query from t without any stopping criterion. We obtain an array B where $B[v]$ is the length of the shortest down path from v to t or $B[v] = \infty$ if there is no such path. We then iteratively calculate the distance $\mu(v, t)$ of nodes of the same level, starting at the highest level. This is possible since nodes of the same level must not have edges between them. To obtain the distances of nodes u of the next lower level, we find the minimum $\min(B[u], \min_v(\text{len}(u, v) + \mu(v, t)))$ for all upward edges (u, v) of the node v. The distances $\mu(v, t)$ were already computed in the previous iteration.

We can use PHAST as to compute potentials if we run its two steps beforehand as a preprocessing step and then look up the respective distances. This preprocessing would be slow since it computes the distances to all nodes. CH-Potentials mitigate this problem

by computing the results of the second PHAST step lazily while the first step remains the same. We do not calculate the distances of all nodes beforehand. To compute the potential of a node u, we recursively compute the potential for all nodes v which are connected to u by upward edges $(u, v) \in E^+$. We then can compute the distance to t as in the PHAST algorithm by finding $\min(B[u], \min_v(\text{len}(u, v) + \mu(v, t)))$. We save all the calculated distances $\mu(v, t)$ in order to not compute any distance value twice.

CH-Potentials can be optimized further as [SZ21] describes in detail.

3. Related Work

The planning of breaks on a route in a routing or scheduling context has been studied beforehand. The introduction of regulation EC 561/2006 of the European Union [Par06] which regulates driving and rest times has led to the work of [Goe09] who introduce a model for the regulation in the context of the Truck Driver Scheduling Problem (TDSP) which they solve using a label propagation algorithm. In the TDSP, a schedule must be found to visit multiple customers with given distances between them. Each customer must be visited in a given time window. The authors revisit the TDSP in later publications, addressing regulations of different countries, such as the US [GK12]. The authors of [SSVB07] revisit the TDSP and name the respective variants EU-TDSP for the EU and US-TDSP for the US. All variants have in common that breaks must be taken on a route in regular intervals. In [Goe12], the authors introduce the Minimum Truck Driver Scheduling Problem (MD-TDSP) which aims to find a valid truck driver schedule with a minimal total duration while breaks can only be taken at customers and specified rest areas. The problem is solved using mixed integer linear programming. The Truck Driver Scheduling and Routing Problem (TDSRP) presents an extension of the TDSP and has been studied with slightly varying definitions. The work of [Sha08] solves a time-dependent version the TDSRP heuristically and allows breaks on every node, there are no dedicated parking nodes. The Bachelor's thesis of [Brä16] aims to solve the problem with an exact algorithm but fails to present an algorithm with practicable running time. The dissertation of [Kle19], which also addresses the TDSP, extends this work and presents an exact algorithm and a heuristic for the TDSRP. They allow parking only at dedicated nodes in the network for which they introduce the term *no-break-en-route policy*. The number of breaks between customers is limited to one break. The master's thesis of [Bom20] targets the TDSRP with no-break-en-route-policy in its non-time-dependent version. They restrict the number of types of driving time constraints to two (TDSRP-2B) and provide an exact solution using mixed integer linear programming and, additionally, a heuristic approach. They also briefly present an extension to solve a time-dependent variant of the problem (TD-TDSRP-2B). The TDSRP-2B in contrast to the TDSP is already related to the long-haul truck driver routing problem (LH-TDRP) of our work. The LH-TDRP closely resembles the problem of routing between customers in the TDSRP-2B with regard to parking areas (no-breaks-en-route-policy) and driving time constraints.

The LH-TDRP itself or variations of it were also addressed in previous work, although using different terminology. The work of [KBS+17] restricts the LH-TDRP by only allowing one break on a route, but includes traffic predictions using time-dependent road networks.

It uses a version of core contraction hierarchies where all the parking nodes are part of the core, which we will revisit later in this work. The authors of [vdTdWB18] also allow only one break on a route and combine their approach with temporary driving bans and road closures. First, they introduce a multi-label version of Dijkstra's algorithm. Second, they present a heuristic version using time-dependent CH with an improved runtime in comparison. In [KSWZ20], the long-haul truck driver routing problem is restricted to one type of driving time constraint but with an unlimited amount of breaks on a route. The work also addresses temporary driving bans, road closures, and a rating of parking areas. It therefore searches for pareto-optimal solutions with regard to travel time and ratings of the used parking areas. The approach is too slow to be used in practice, but it uses a combination of the A* algorithm and CH-Potentials which we also will revisit later.

Other publications address different complex real-world scenarios which present an extension to the SPP. As in this work, most of them strive to integrate the additional requirements into CH or other speed-up techniques which exploit the hierarchy of road networks [BDG+15]. Integrating traffic predictions, respectively time-dependent road networks with CH is addressed by [BDSV09, BGSV13]. Turn costs are addressed by [GV11]. Mitigating the complex task of integrating additional information into CH is addressed by [SZ21] by building a tight potential based on CH which then is used in combination with the A* algorithm. They use the same principle to address live traffic and extend the CH variant to a customizable contraction hierarchy [DSW15] to address temporary driving bans.

TODO: [MDGCNR20] - neccessary?

4. Problem and Definitions

Truck drivers have to follow regulations regarding the maximum time they are allowed to drive without taking a break. Therefore, on longer routes planning becomes necessary. We propose an extension of the shortest path problem which accounts for driving time limits and mandatory breaks, denoted as the long-haul truck driver routing problem. It can be formalized as follows:

Let $G = (V, E, \text{len})$ be a graph and s and t nodes with $s, t \in V$. We extend the graph with a set $P \subseteq V$ of parking nodes. Additionally, we introduce a set C of driving time constraints c_i. Each driving time constraint is defined by a maximum permitted driving time c_i^d and a break time c_i^b. We assume an order among the constraints and that both the driving time and the break time correspond to this order, i.e. $i < j \implies c_i^d < c_j^d \wedge c_i^b < c_j^b$. Before exceeding a driving time of c_i^d, the driver must stop and break for a time of at least c_i^b. Afterwards, the driver is allowed to drive for a maximum time of c_i^d again without stopping. Breaks can only take place at nodes $v \in P$.

A route r from s to t includes the path of visited nodes $p = \langle s = v_0, v_1, \ldots, t = v_k \rangle$ and a break time function breakTime: $p \to \{0, c_1^d, \ldots, c_{|C|}^d\}$ at each node i. For non-parking nodes $v_i \notin P$, the break time must be zero. We also define the breakTime of an entire route r on p as $\text{breakTime}(r) = \sum_{i=0}^{k-1} \text{breakTime}((v_i, v_{i+1}))$.

Definition 4.1 (Valid Route)**.** *A valid route with path* $p = \langle s = v_0, v_1, ..., t = v_k \rangle$ *must comply with all driving time constraints in* C*.A path complies with a specific driving time constraint* $c \in C$ *if there is no subpath* p' *between two nodes* $u, w \in P' = \{s, t\} \cup \{v_i \in p \mid \text{breakTime}(v_i) \geq c^b\}$ *on the path which exceeds the driving time limit* $\text{len}(p') > c^d$ *and has no third node* $v_i \in P'$ *in between* u *and* w*.*

We define the travel time of a route and the shortest route between two nodes as follows.

Definition 4.2 (Travel Time of a Route)**.** *The travel time* $\text{travelTime}(r)$ *of a route* r *is the sum of the length of its path* $\text{len}(p)$ *and the accumulated break time* $\text{breakTime}(p)$*.*

Definition 4.3 (Shortest Route)**.** *A route between two nodes* s *and* t *is called a shortest route if it is valid and there exists no different valid route between* s *and* t *with a smaller travel time.*

The shortest travel time between two nodes s and t, i.e., the travel time of the shortest route between them is denoted as $\mu_{tt}(s, t)$. The long-haul truck driver routing problem now can be defined as follows.

Long-Haul Truck Driver Routing

Input: A graph $G = (V, E, \text{len})$, a set of parking nodes $P \subseteq V$, a set of driving time constraints C, and start and target nodes $s, t \in V$

Problem: Find the shortest valid route r from s to t in G.

In many practical applications, the number of different driving time constraints is limited to only one or two constraints, i.e., $|C| = 1$ or $|C| = 2$. Therefore, we will often only consider two special cases.

5. Algorithm

In this chapter, we introduce a labeling algorithm which solves the long-haul truck driver routing problem. At first, we will restrict the problem to one driving time constraint for simplicity and drop that constraint later. We then describe extensions of the base algorithm to achieve better running times on realistic problem instances.

5.1. Dijkstra's Algorithm with One Driving Time Constraint

We will adapt Dijkstra's algorithm for solving the long-haul truck driver routing problem with one driving time constraint $C = \{c\}$ and abbreviate this restriction of the problem *1-DTC*. While Dijkstra's algorithm manages a queue of nodes and assigns each node one tentative distance, our algorithm manages a queue Q of labels and a set $L(v)$ of labels for each node $v \in V$.

Labels in a label set $L(v)$ represent a possible route, respectively a possible solution for a query from s to v. A label $l \in L(v)$ may represent suboptimal routes to v, i.e., routes which are not a shortest route between s and v. Nevertheless, we will ensure that a label set never contains labels which represent invalid routes according to c. A label l contains

- travelTime(l), the total travel time from the starting node s
- breakDist(l), the distance since the last break
- pred(l), its preceding label

As in Dijkstra's algorithm, the queue is a min-Queue with ascending keys. The key of a label l is its accumulated $\mu_{tt}(l)$.

5.1.1. Settling a Label

In contrast to Dijkstra's algorithm, the search *settles* a label $l \in L(u)$ in each iteration instead of a node u. When settling a label, the search first removes l from the queue. Similar to Dijkstra, it then relaxes all edges $(u, v) \in E$ with $l \in L(u)$ as shown in Figure 5.1.

Relaxing an edge consists of the three steps label *propagation*, *pruning* and *dominance* checks.

Label Propagation. Labels can be propagated along edges. Let $l \in L(u)$ be a label at u and $(u, v) = e \in E$, then l can be propagated to v resulting in a label l' with travelTime(l') = travelTime(l) + len(e), breakDist(l') = breakDist(l) + len(e), and pred(l') = l.

Algorithm 5.1: Settling a label $l \in L(u)$ removes the label from the queue and relaxes all the outgoing edges of u.

```
1 Procedure SettleNextLabel():
2     l ← Q.deleteMin()
3     forall (u, v) ∈ E do
4         RelaxEdge((u, v), l)
```

Label Pruning. After propagating a label, we discard the label if it violates the driving time constraint c, that is, if $\text{breakDist}(l) > c^d$.

Label Dominance In general, it is no longer clear when a label presents a better solution than another label since it now contains two distance values. A label l at a node v might represent a shorter route from s to v than another label l' but might have shorter remaining driving time budget $c^d - \text{breakDist}(l)$. The label l yields a better solution for a query s-v, but this does not imply that it is part of a better solution for a query from s-t. It might not even yield a valid route to t at all while l' reaches the target due to the greater remaining driving time budget. In one case, we can prove that a label $l \in L(v)$ cannot yield a better solution than a label $l' \in L(v)$. We say l' *dominates* l.

Definition 5.1 (Label Dominance for 1-DTC)**.** *A label $l \in L(v)$ dominates another label $l' \in L(v)$ if* $\text{travelTime}(l') > \text{travelTime}(l)$ *and* $\text{breakDist}(l') \geq \text{breakDist}(l)$ *or* $\text{travelTime}(l') \geq \text{travelTime}(l)$ *and* $\text{breakDist}(l') > \text{breakDist}(l)$.

If a label $l \in L(v)$ is dominated by another label $l' \in L(v)$, then l' represents a route from s to t with a shorter or equal total travel time and longer or equal remaining driving time budget until the next break. Therefore, in each solution which uses the label l, l can trivially be replaced by the label l'. The solution will still comply with the driving time constraint c and yield a shorter or equal total travel time, so we are allowed to simply discard dominated labels in our search.

Definition 5.2 (Pareto-Optimal Label)**.** *A label $l \in L(v)$ is pareto-optimal if it is not dominated by any other label $l' \in L(v)$.*

A label l will only be inserted into a label set $L(v)$ if it is pareto-optimal. If a label l is inserted into $L(v)$, labels $l' \in L(v)$ are removed from $L(v)$ if l dominates them. $L(v)$ therefore is the set of known pareto-optimal solutions at v. In Figure 5.2 we define the procedure RemoveDominated(l) as an operation on a label set.

Algorithm 5.2: The procedure L.RemoveDominated(l) removes all labels from the label set L which are dominated by the label l.

```
1 Procedure RemoveDominated(l):
2     forall l' ∈ L do
3         if l dominates l' then
4             L.remove (l');
```

5.1.2. Parking at a Node

When propagating a label $l \in L(u)$ along an edge $(u, v) \in E$ and $v \in P$, we have to consider pausing at v. Since we do not know if pausing at v or continuing without a break is the better solution, we generate both labels and add them to the label set $L(v)$ and the queue Q. We now can define the procedure RelaxEdge as in Figure 5.3.

Algorithm 5.3: Relaxing an edge $(u, v) \in E$ when settling a label $l \in L(u)$ with regard to parking nodes.

```
1  Procedure RelaxEdge((u,v), l):
2      D ← {}
3      if breakDist(l) + len(u, v) ≤ c^d then
4          D.insert((travelTime(l) + len(u, v), breakDist(l) + len(u, v), l))
5          if v ∈ P then
6              D.insert((travelTime(l) + len(u, v) + c^b, 0, l))
7          forall l' ∈ D do
8              if l' is not dominated by any label in L(v) then
9                  L(v).RemoveDominated(l')
10                 L(v).insert(l')
11                 Q.queueInsert(travelTime(l'),l')
```

5.1.3. Initialization and Main Loop

We initialize the label set $L(s)$ of s and the queue Q with a label which only contains distances of zero and a dummy element as a predecessor. We stop the search when t was removed from Q. The definition of the final algorithm is given as Algorithm 5.4 Dijkstra+1-DTC.

Algorithm 5.4: Dijkstra+1-DTC

Input: Graph $G = (V, E, \text{len})$, set of parking nodes $P \subseteq V$, set of driving time constraints $C = \{r\}$, start and target nodes $s, t \in V$
Data: Priority queue Q, per node set L(v) of labels for all $v \in V$
Output: Shortest route with travelTime(j) $= \mu_{tt}(s, t)$

```
// Initialization
1  Q.queueInsert(0,(0, 0, ⊥))
2  L(s).insert((0, 0, ⊥))
// Main loop
3  while Q is not empty do
4      SettleNextLabel()
5      if label at t was settled then
6          return
```

5.1.4. Correctness

Given a start node s and a target node t, Dijkstra's algorithm returns the shortest distance between s and t. The algorithm can be stopped after t was removed from the queue since all the following nodes in the queue have larger distances and the edge lengths are non-negative by definition. Therefore, relaxing an outgoing edge of these nodes cannot lead to an improvement of the distance at t.

In our case, the algorithm shall return the shortest travel time $\mu_{tt}(s, t)$ between two nodes s and t. We have a queue of labels which is sorted in ascending order by their travel time. When removing a label l from the queue, all the other labels in the queue therefore have a larger travel time than travelTime(l). Additionally, all the edge lengths and the break times are non-negative. Relaxing an edge thus can only increase the travel time. The same

Algorithm 5.5: A*+1-DTC

Input: Graph $G = (V, E, \text{len})$, set of parking nodes $P \subseteq V$, a set of driving time constraints $C = \{r\}$, start and target nodes $s, t \in V$, potential $\text{pot}_t()$
Data: Priority queue Q, per node set $\mathsf{L}(v)$ of labels for all $v \in V$
Output: Shortest route with $\text{travelTime}(j) = \mu_{tt}(s, t)$

```
// Initialization
l_s ← (0, 0, ⊥)
Q.QUEUEINSERT(pot_t((l_s), s), l_s)
L(s).INSERT(l_s)
// Main loop
while Q is not empty do
    SETTLENEXTNODE()
    if minimum of Q is label at t then
        return
```

argument as for the correctness of Dijkstra's algorithm applies: Since relaxing an edge can only increase travel time, labels which are added later to the queue will also have a larger travel time than $\text{travelTime}(l)$. Therefore, when we remove the first label $l_t \in L(t)$ at t from the queue, we know that this time cannot be improved further and it is correct to stop the search.

When relaxing an edge (u, v) and propagating a label $l \in L(u)$, we check if the new label complies with the driving time constraints. We do not insert the new label into $L(v)$ and the queue if it violates a constraint. Therefore, label sets and the queue only contain labels which represent a valid route and l_t must also represent a valid route.

If we propagated a label to a parking node, we produce a second label since we have the two choices of parking and not parking at the node. We treat both labels equally and insert both labels into label set and queue if they are not dominated or represent an invalid route. The label with the smaller travel time will be removed first from the queue, and the second label will be handled at a later stage when its travel time becomes the smallest of all labels in the queue. It is not possible to miss possible routes between s and t because we produce all the labels and only discard dominated labels and labels representing invalid routes. There therefore cannot be a label with a better travel time from s to t than the label l_t which we removed as the first label at t from the queue. Therefore, its travel time $\text{travelTime}(l_t)$ is equal to the shortest travel time $\mu_{tt}(s, t)$ between s and t.

5.2. Goal-Directed Search with One Driving Time Constraint

In this section, we extend the base algorithm described in Section 5.1 to a goal-directed search with the A* algorithm. We introduce a new potential pot_t based on the CH-Potentials in Section 2.3. We then show that we still can stop the search when the first label at t is removed from the queue.

The difference between Dijkstra and A* is the order in which nodes are being removed from the queue. In our case, this corresponds to the order of labels being removed from the queue. Instead of using their travel time $\text{travelTime}(l)$ as a queue key, a label $l \in L(v)$ is added to the queue with the key $\text{travelTime}(l) + \text{pot}_t(l, v)$. Algorithm 5.5 shows the adaption of the coarse algorithm.

The only thing left is the adaption of RELAXEDGE in Figure 5.3 where we change the queue keys to use $\text{travelTime}(l) + \text{pot}(l, v)$ instead. The result is shown in Figure 5.6.

Algorithm 5.6: Relaxing an edge with regard to the potential.

```
1  Procedure RelaxEdge((u, v), l):
2      D ← {}
3      if breakDist(l) + len(u, v) ≤ c^d then
4          D.insert((travelTime(l) + len(u, v), breakDist(l) + len(u, v), l))
5          if v ∈ P then
6              D.insert((travelTime(l) + len(u, v) + c^b, 0, l))
7          forall l' ∈ D do
8              if l' is not dominated by any label in L(v) then
9                  L(v).RemoveDominated(l')
10                 L(v).insert(l')
11                 Q.queueInsert(travelTime(l') + pot_t(l', v), l')
```

5.2.1. Potential for One Driving Time Constraint

In general, every feasible potential can be used for the goal-directed algorithm. We use CH-Potentials as foundation to build a potential which accounts for necessary break times on the route.

Given a target node t, the CH-Potentials yield a perfect estimate for the distance $\mu(v, t)$ from v to t without regard for driving time constraints and breaks. This is a lower bound for the remaining travel time for any label at v. A better lower bound for the remaining travel time of a label at v to t, including breaks due to the driving time limit, can be calculated by taking the minimum necessary amount of breaks into account. We define minBreaks(d) as a function of time which calculates the minimum amount of necessary breaks for any arbitrary driving time d.

$$\text{minBreaks}(d) = \begin{cases} \left\lceil \frac{d}{c^d} \right\rceil - 1 & d > 0 \\ 0 & else \end{cases} \tag{5.1}$$

Simply using $\left\lfloor \frac{d}{c^d} \right\rfloor$ is not sufficient since we do not need to pause for a driving time of exactly c^d. We now can calculate a lower bound for the minimum necessary break time given an arbitrary driving time d

$$\text{minBreakTime}(d) = \text{minBreaks}(d) \cdot c^b \tag{5.2}$$

and finally define our node potential as

$$\text{pot}'_t(v) = \text{minBreakTime}(\text{chPot}_t(v)) + \text{chPot}_t(v) \tag{5.3}$$

A node potential is called *feasible* if it does not overestimate the distance of any edge in the graph, i.e.

$$len(u, v) - \text{pot}_t(u) + \text{pot}_t(v) \geq 0 \quad \forall (u, v) \in E \tag{5.4}$$

A feasible node potential allows us to stop the A* search when the node t, respectively the first label at t, was removed from the queue. Following counterexample of a query using the graph in Fig. 5.1 shows that pot'_t is not feasible. With a driving time limit of 6 and a break time of 1, the potential here will yield a value $\text{pot}_t(s) = 8$ since the potential includes the minimum required break time for a path from s to t. Consequently, with $\text{pot}'_t(v) = 5$ and $len(s, v) = 2$, $len(s, v) - \text{pot}'_t(s) + \text{pot}'_t(v) = -1$.

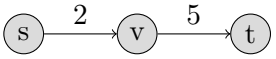

Figure 5.1.: A graph for which the feasibility condition of 5.4 does not always hold with the potential pot′.

A variant of the potential accounts for the distance since the last break of a label $\text{breakDist}(l)$ to calculate the minimum required break time on the v-t path.

$$\text{pot}_t(l, v) = \text{minBreakTime}(\text{breakDist}(l) + \text{chPot}_t(v)) + \text{chPot}_t(v) \tag{5.5}$$

Since the potential now uses information from a label l with $l \in L(v)$, it no longer is a node potential but also depends on the chosen label at v. The feasibility definition as defined in 5.4 can no longer be applied. We therefore have to show that queue keys of labels can only increase over time.

Lemma 5.3. *Let l be a label in the label set $L(u)$ which the algorithm propagates along the edge $(u, v) \in E$ to create a label $l' \in L(v)$. The sum of travel time and potential of a label can only increase, i.e,* $\text{travelTime}(l) + \text{pot}_t(l, u) \leq \text{travelTime}(l') + \text{pot}_t(l', v)$.

Proof. Let $(u, v) \in E$ be an edge. The procedure RelaxEdge in Figure 5.6 can produce two new labels at a node v for each label at u, depending on if v is a parking node. We differentiate the two cases not parking at v and parking at v. Let $l \in L(u)$ and $l' \in L(v)$.

Following general observations can be made:

1. $d \geq d' \implies \text{minBreakTime}(d) \geq \text{minBreakTime}(d')$
2. $c^b + \text{minBreakTime}(d) \geq \text{minBreakTime}(d + c^d)$
3. $c^d \geq \text{breakDist}(l') \geq \text{breakDist}(l) + \text{len}(u, v) \geq \text{breakDist}(l)$
   (Line 2 in RelaxEdge in Figure 5.6)
4. $\text{len}(u, v) - \text{chPot}_t(u) + \text{chPot}_t(v) \geq 0$ (feasibility of the CH-Potentials)
5. $\text{len}(u, v) + \text{chPot}_t(v) \geq \text{chPot}_t(u)$

We show that $\text{travelTime}(l') + \text{pot}_t(l', v) - \text{travelTime}(l) - \text{pot}_t(l, u) \geq 0$.

Case 1: Not parking at v. In this case, $\text{travelTime}(l') = \text{travelTime}(l) + \text{len}(u, v)$ and $\text{breakDist}(l') = \text{breakDist}(l) + \text{len}(u, v)$.

$$
\begin{aligned}
&\text{travelTime}(l') - \text{travelTime}(l) - \text{pot}_t(l,u) + \text{pot}_t(l',v) \\
&= \text{travelTime}(l) + \text{len}(u,v) - \text{travelTime}(l) \\
&\quad - (\text{minBreakTime}(\text{breakDist}(l) + \text{chPot}_t(u)) + \text{chPot}_t(u)) \\
&\quad + \text{minBreakTime}(\text{breakDist}(l') + \text{chPot}_t(v)) + \text{chPot}_t(v) \\
&= \text{len}(u,v) + \text{minBreakTime}(\text{breakDist}(l') + \text{chPot}_t(v)) \\
&\quad - \text{minBreakTime}(\text{breakDist}(l) + \text{chPot}_t(u)) - \text{chPot}_t(u) + \text{chPot}_t(v) \\
&= \text{len}(u,v) + \text{minBreakTime}(\text{breakDist}(l) + \text{len}(u,v) + \text{chPot}_t(v)) \qquad (5.6) \\
&\quad - \text{minBreakTime}(\text{breakDist}(l) + \text{chPot}_t(u)) - \text{chPot}_t(u) + \text{chPot}_t(v) \\
&\overset{\text{(1. and 5.)}}{\geq} \text{len}(u,v) + \text{minBreakTime}(\text{breakDist}(l) + \text{chPot}_t(u)) \\
&\quad - \text{minBreakTime}(\text{breakDist}(l) + \text{chPot}_t(u)) - \text{chPot}_t(u) + \text{chPot}_t(v) \\
&= \text{len}(u,v) - \text{chPot}_t(u) + \text{chPot}_t(v) \\
&\overset{\text{(4.)}}{\geq} 0
\end{aligned}
$$

Case 2: Parking at v. In this case, $\text{travelTime}(l') = \text{travelTime}(l) + \text{len}(u,v) + c^b$ and $\text{breakDist}(l') = 0$.

$$
\begin{aligned}
&\text{travelTime}(l') - \text{travelTime}(l) - \text{pot}_t(l,u) + \text{pot}_t(l',v) \\
&= \text{travelTime}(l) + \text{len}(u,v) + c^b - \text{travelTime}(l) \\
&\quad - (\text{minBreakTime}(\text{breakDist}(l) + \text{chPot}_t(u)) + \text{chPot}_t(u)) \\
&\quad + \text{minBreakTime}(\text{chPot}_t(v)) + \text{chPot}_t(v) \\
&= \text{len}(u,v) + c^b + \text{minBreakTime}(\text{chPot}_t(v)) \\
&\quad - \text{minBreakTime}(\text{breakDist}(l) + \text{chPot}_t(u)) - \text{chPot}_t(u) + \text{chPot}_t(v) \\
&\overset{\text{(2.)}}{\geq} \text{len}(u,v) + \text{minBreakTime}(c^d + \text{chPot}_t(v)) \\
&\quad - \text{minBreakTime}(\text{breakDist}(l) + \text{chPot}_t(u)) - \text{chPot}_t(u) + \text{chPot}_t(v) \qquad (5.7) \\
&\overset{\text{(1. and 3.)}}{\geq} \text{len}(u,v) + \text{minBreakTime}(\text{breakDist}(l) + \text{len}(u,v) + \text{chPot}_t(v)) \\
&\quad - \text{minBreakTime}(\text{breakDist}(l) + \text{chPot}_t(u)) - \text{chPot}_t(u) + \text{chPot}_t(v) \\
&\overset{\text{(1. and 4.)}}{\geq} \text{len}(u,v) + \text{minBreakTime}(\text{breakDist}(l) + \text{chPot}_t(u)) \\
&\quad - \text{minBreakTime}(\text{breakDist}(l) + \text{chPot}_t(u)) - \text{chPot}_t(u) + \text{chPot}_t(v) \\
&= \text{len}(u,v) - \text{chPot}_t(u) + \text{chPot}_t(v) \\
&\overset{\text{(4.)}}{\geq} 0
\end{aligned}
$$

□

Lemma 5.4. *The sum* $\text{travelTime}(l) + \text{pot}_t(l,v)$ *of a label* l *at a node* v *is a lower bound for the travel time from* s *to* t *of the route using* l*.*

Proof. Let r be a route between s and t with the path $p = \langle s = v_0, v_1, \ldots, t = v_k \rangle$. The route is represented by labels l_i at nodes v_i. With Lemma 5.3 and $\text{pot}_t(l_k, v_k) = 0$ follows

$$\begin{aligned}\text{travelTime}(l_i) + \text{pot}_t(l_i, v_i) &\leq \text{travelTime}(l_{i+1}) + \text{pot}_t(l_{i+1}, v_{i+1}) \\ &\leq \cdots \leq \text{travelTime}(l_k) + \text{pot}_t(l_k, v_k) \\ &= \text{travelTime}(l_k) = \text{travelTime}(r)\end{aligned}$$

□

Theorem 5.5. *The search can be stopped when the first label at t is removed from the queue.*

Proof. Let l_t be the first label at t which is removed from the queue during an s-t query. It represents a route r from s to t with a travel time of $\text{travelTime}(r) = \text{travelTime}(l_t)$. When the label l_t is removed from the queue, all remaining labels l at nodes v in the queue fulfill $\text{travelTime}(l) + \text{pot}_t(l, v) \geq \text{travelTime}(l_t) + \text{pot}_t(l_t, t)$. The same holds for all labels which will be inserted into the queue at a later point in time (Lemma 5.3). Assume for contradiction that r is not the shortest possible route from s to t. Then, a shorter route r' exists which uses at least one unsettled label $l \in L(v)$ at a node v. The label l is eventually propagated to t where a label $l'_t \in L(t)$ is created to represent the route r'. With Lemmas 5.3, 5.4, and $\text{pot}_t(l_t, t) = 0$ follows

$$\begin{aligned}\text{travelTime}(r) = \text{travelTime}(l_t) &= \text{travelTime}(l_t) + \text{pot}_t(l_t, t) \\ &\overset{(5.3)}{\leq} \text{travelTime}(l) + \text{pot}_t(l, v) \\ &\overset{(5.4)}{\leq} \text{travelTime}(l'_t) = \text{travelTime}(r')\end{aligned} \tag{5.8}$$

which contradicts the assumption that r' yields a shorter s-t route than r. Therefore, it must be $\text{travelTime}(r) = \mu_{tt}(s, t)$ when l_t was removed from the queue. The search can be stopped when the first label at t is removed from the queue. □

5.3. Multiple Driving Time Constraints

Dijkstra's algorithm with one driving time constraint (1-DTC) can be adapted to handle multiple driving time constraints c_i. With anumber of $|C|$ driving time constraints, a label l now contains the total travel time $\text{travelTime}(l)$ and $|C|$ distances $\text{breakDist}_1(l), \ldots, \text{breakDist}_{|C|}(l)$. Each value $\text{breakDist}_i(l)$ represents the distance since the last break at a node v with break time $\text{breakTime}(v) \geq c_i^b$. Pausing at a node occurs with one of the available break times c_i^b of a driving time constraint $c_i \in C$. Pausing with an arbitrary break time is permitted but yields longer travel times and no advantage and is therefore ignored. When a route breaks at v for a time c_i^b, the corresponding label $l \in L(v)$ has $\text{breakDist}(l) = 0$ for all $0 < j \leq i$ since the breaks with shorter break times are included in the longer break. In the following, we redefine the fundamental concepts of Section 5.1 for multiple driving time constraints.

Label Propagation Label propagation simply extends the component-wise addition of the edge weight. Let $l \in L(u)$ be a label at u and $(u, v) = e \in E$, then l can be propagated to v resulting in a label l' with $\text{travelTime}(l') = \text{travelTime}(l) + \text{len}(e)$, $\text{breakDist}_i(l') = \text{breakDist}_i(l) + \text{len}(e)\ \forall\ 1 \leq i \leq |C|$, and $\text{pred}(l') = l$.

Label Pruning The pruning rule for driving time constraints is generalized in a similar way. A label is discarded if $\text{breakDist}_i(l) > c_i^d$ for any i with $0 < i \leq |C|$.

Label Dominance Label dominance can be generalized to multiple driving time constraints as follows.

Definition 5.6 (Label Dominance)**.** *A label $l \in L(v)$ dominates another label $l' \in L(v)$ if* $\text{travelTime}(l') \geq \text{travelTime}(l)$ *and* $\text{breakDist}_i(l') \geq \text{breakDist}_i(l)\ \forall\ 1 \leq i \leq |C|$.

5.3.1. Potential for Multiple Driving Time Constraints

TODO: Rewrite for only two dtc?

In Section 5.2.1 we defined the potential $\text{pot}_t(l, v)$ to extend Dijkstra's algorithm with one driving time constraint to a goal-directed search using the A* algorithm. We will now generalize pot_t for the use with an arbitrary number of driving time constraints.

In Equation 5.1 we used the distance $\mu(v, t)$ without regard for pausing from v to t and the distance $\text{breakDist}(l)$ since the last break on the route to calculate a lower bound for the amount of necessary breaks until we reach the target node. We now have to calculate the lower bound with respect to all driving time constraints. How many breaks of which duration do we need at least to comply with all driving time constraints c_i? For longer driving time constraints, we will always need a greater or equal amount of breaks than for shorter driving time constraints since they have a longer maximum allowed driving time c_i^d. At the same time, a break of length c_i^b will also include breaks of lengths c_j^b with $j < i$. We start with calculating the amount of necessary breaks $\text{minBreaks}_i(d)$, given a driving time d, for all constraints c_i independently.

$$\text{minBreaks}_i(d) = \begin{cases} \left\lceil \frac{d}{c_i^d} \right\rceil - 1 & d > 0 \\ 0 & else \end{cases} \tag{5.9}$$

Consider the example graph in Figure 5.2 with two driving time constraints with permitted driving times of 4 and 9. Since the distance $\mu(s, t)$ is 10, a route must have at least one long and two short breaks. If the long break is made at u, only one additional short break must be made at w. The long break made one shorter break obsolete. To obtain a lower bound for the amount of breaks for a constraint c_i, we therefore must subtract the minimum amount of longer breaks being made on the route.

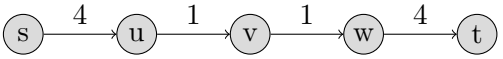

Figure 5.2.: An example graph where, depending on the driving time constraints, a long break at can render a short break obsolete.

This is an optimistic assumption since not in all cases a longer break spares a shorter break. Revisit the example graph of Figure 5.2 with permitted driving times of 4 and 5. We still need one long and two short breaks, but the long break now must take place at v while the short breaks must take place at u and w. The long break did not spare a short break. Since we are searching for a lower bound for the amount of breaks, optimistic assumptions are necessary. Given a label $l \in L(v)$, the lower bound estimate for the remaining number of breaks of length c_i^b on the route to t then becomes

$$\text{breakEstimate}_i(l, v) = \begin{cases} \text{minBreaks}_i(\text{breakDist}_i(l) + \text{chPot}_t(v)) & 0 < i < |C| \\ \quad - \sum_{j=i+1}^{|C|} \text{breakEstimate}_j(l, v) & \\ \text{minBreaks}_{|C|}(\text{breakDist}_{|C|}(l) + \text{chPot}_t(v)) & i = |C| \end{cases} \tag{5.10}$$

Since we subtract all break estimates for break times greater than c_i^b to obtain the estimate for c_i^b, we can just use

$$\text{breakEstimate}_i(l,v) = \begin{cases} \text{minBreaks}_i(\text{breakDist}_i(l) + \text{chPot}_t(v)) & 0 < i < |C| \\ \quad - \text{minBreaks}_{i+1}(\text{breakDist}_{i+1}(l) + \text{chPot}_t(v)) & \\ \text{minBreaks}_{|C|}(\text{breakDist}_{|C|}(l) + \text{chPot}_t(v)) & i = |C| \end{cases} \tag{5.11}$$

We now can calculate a lower bound estimate for the remaining necessary break time on the route to t for two driving time constraints.

$$\text{remBreakTime}(l,v) = \sum_{i=1}^{|C|} \text{breakEstimate}_i(l,v) \cdot c_i^b \tag{5.12}$$

Finally, the lower bound potential for a label $l \in L(v)$ and a target node t becomes

$$\text{pot}_t(l,v) = \text{remBreakTime}(l,v) + \text{chPot}_t(v,t) \tag{5.13}$$

If queue keys still cannot decrease when propagating labels, Lemma 5.4 and theorem 5.5 follow as a consequence. We follow the outLine of the proof of Lemma 5.3 and therefore revisit the procedure RelaxEdge at an edge $(u,v) \in E$ with a label $l \in L(u)$ and a new label $l' \in L(v)$.

Lemma 5.7. *Lemma 5.3 still holds for two driving time constraints.*

Proof. There now are three cases to differentiate: not parking at v, short break at v, and long break at v.

Following general observations can be made in an addition to the proof of Lemma 5.3:

6. $d \geq d' \implies \text{minBreaks}_i(d) \geq \text{minBreaks}_i(d')$ (adaption of 1.)
7. $1 + \text{minBreaks}_i(d) \geq \text{minBreaks}_i(d + c_i^d)$ (adaption of 2.)
8. $c_i^d \geq \text{breakDist}_i(l') \geq \text{breakDist}_i(l) + \text{len}(u,v) \geq \text{breakDist}_i(l)$ (adaption of 3.)

Case 1: Not parking at v. In this case, $\text{travelTime}(l') = \text{travelTime}(l) + \text{len}(u,v)$ and $\text{breakDist}_i(l') = \text{breakDist}_i(l) + \text{len}(u,v)$.

$$
\begin{aligned}
&\text{travelTime}(l') - \text{travelTime}(l) - \text{pot}_t(l,u) + \text{pot}_t(l',v) \\
&= \text{travelTime}(l) + \text{len}(u,v) - \text{travelTime}(l) \\
&\qquad - (\text{remBreakTime}(l,u) + \text{chPot}_t(u)) + (\text{remBreakTime}(l',v) + \text{chPot}_t(v)) \\
&= \text{len}(u,v) - \text{remBreakTime}(l,u) + \text{remBreakTime}(l',v) - \text{chPot}_t(u) + \text{chPot}_t(v) \\
&= \text{len}(u,v) + \text{breakEstimate}_2(l',v) \cdot c_2^b + \text{breakEstimate}_1(l',v) \cdot c_1^b \\
&\qquad - \text{remBreakTime}(l,u) - \text{chPot}_t(u) + \text{chPot}_t(v) \\
&= \text{len}(u,v) + \text{minBreaks}_2(\text{breakDist}_2(l) + \text{len}(u,v) + \text{chPot}_t(v)) \cdot c_2^b \\
&\qquad + (\text{minBreaks}_1(\text{breakDist}_1(l) + \text{len}(u,v) + \text{chPot}_t(v)) \\
&\qquad - \text{minBreaks}_2(\text{breakDist}_2(l) + \text{len}(u,v) + \text{chPot}_t(v))) \cdot c_1^b \\
&\qquad - \text{remBreakTime}(l,u) - \text{chPot}_t(u) + \text{chPot}_t(v) \\
&\overset{\text{(5. and 6.)}}{\geq} \text{len}(u,v) + \text{minBreaks}_2(\text{breakDist}_2(l) + \text{chPot}_t(u)) \cdot c_2^b \\
&\qquad + (\text{minBreaks}_1(\text{breakDist}_1(l) + \text{chPot}_t(u)) \\
&\qquad - \text{minBreaks}_2(\text{breakDist}_2(l) + \text{chPot}_t(u))) \cdot c_1^b \\
&\qquad - \text{remBreakTime}(l,u) - \text{chPot}_t(u) + \text{chPot}_t(v) \\
&= \text{len}(u,v) + \text{remBreakTime}(l,u) - \text{remBreakTime}(l,u) - \text{chPot}_t(u) + \text{chPot}_t(v) \\
&= \text{len}(u,v) - \text{chPot}_t(u) + \text{chPot}_t(v) \\
&\overset{\text{(4.)}}{\geq} 0
\end{aligned}
\tag{5.14}
$$

Case 2: Short break at v. In this case, $\text{travelTime}(l') = \text{travelTime}(l) + \text{len}(u,v) + c^b$ and $\text{breakDist}_1(l') = 0$ and $\text{breakDist}_2(l') = \text{breakDist}_2(l) + \text{len}(u,v)$.

$$
\begin{aligned}
&\text{travelTime}(l') - \text{travelTime}(l) - \text{pot}_t(l,u) + \text{pot}_t(l',v) \\
&= \text{travelTime}(l) + \text{len}(u,v) + c_1^b - \text{travelTime}(l) \\
&\quad - (\text{remBreakTime}(l,u) + \text{chPot}_t(u)) + (\text{remBreakTime}(l',v) + \text{chPot}_t(v)) \\
&= \text{len}(u,v) + c_1^b + \text{remBreakTime}(l',v) \\
&\quad - \text{remBreakTime}(l,u) - \text{chPot}_t(u) + \text{chPot}_t(v) \\
&= \text{len}(u,v) + c_1^b + \text{breakEstimate}_2(l',v) \cdot c_2^b + \text{breakEstimate}_1(l',v) \cdot c_1^b \\
&\quad - \text{remBreakTime}(l,u) - \text{chPot}_t(u) + \text{chPot}_t(v) \\
&= \text{len}(u,v) + c_1^b + \text{minBreaks}_2(\text{breakDist}_2(l) + \text{len}(u,v) + \text{chPot}_t(v)) \cdot c_2^b \\
&\quad + (\text{minBreaks}_1(0 + \text{chPot}_t(v)) \\
&\quad - \text{minBreaks}_2(\text{breakDist}_2(l) + \text{len}(u,v) + \text{chPot}_t(v))) \cdot c_1^b \\
&\quad - \text{remBreakTime}(l,u) - \text{chPot}_t(u) + \text{chPot}_t(v) \\
&\overset{(7.)}{\geq} \text{len}(u,v) + \text{minBreaks}_2(\text{breakDist}_2(l) + \text{len}(u,v) + \text{chPot}_t(v)) \cdot c_2^b \\
&\quad + (\text{minBreaks}_1(r_1^d + \text{chPot}_t(v)) \\
&\quad - \text{minBreaks}_2(\text{breakDist}_2(l) + \text{len}(u,v) + \text{chPot}_t(v))) \cdot c_1^b \\
&\quad - \text{remBreakTime}(l,u) - \text{chPot}_t(u) + \text{chPot}_t(v) \\
&\overset{(6.\text{ and }8.)}{\geq} \text{len}(u,v) + \text{minBreaks}_2(\text{breakDist}_2(l) + \text{len}(u,v) + \text{chPot}_t(v)) \cdot c_2^b \\
&\quad + (\text{minBreaks}_1(\text{breakDist}_1(l) + \text{len}(u,v) + \text{chPot}_t(v)) \\
&\quad - \text{minBreaks}_2(\text{breakDist}_2(l) + \text{len}(u,v) + \text{chPot}_t(v))) \cdot c_1^b \\
&\quad - \text{remBreakTime}(l,u) - \text{chPot}_t(u) + \text{chPot}_t(v) \\
&\overset{(5.\text{ and }6.)}{\geq} \text{len}(u,v) + \text{minBreaks}_2(\text{breakDist}_2(l) + \text{chPot}_t(u)) \cdot c_2^b \\
&\quad + (\text{minBreaks}_1(\text{breakDist}_1(l) + \text{chPot}_t(u)) \\
&\quad - \text{minBreaks}_2(\text{breakDist}_2(l) + \text{chPot}_t(u))) \cdot c_1^b \\
&\quad - \text{remBreakTime}(l,u) - \text{chPot}_t(u) + \text{chPot}_t(v) \\
&= \text{len}(u,v) + \text{remBreakTime}(l,u) - \text{remBreakTime}(l,u) - \text{chPot}_t(u) + \text{chPot}_t(v) \\
&= \text{len}(u,v) - \text{chPot}_t(u) + \text{chPot}_t(v) \\
&\overset{(4.)}{\geq} 0
\end{aligned}
\tag{5.15}
$$

Case 3: Long break at v. In this case, $\text{travelTime}(l') = \text{travelTime}(l) + \text{len}(u,v) + c^b$ and $\text{breakDist}_i(l') = 0$.

$$
\begin{aligned}
&\text{travelTime}(l') - \text{travelTime}(l) - \text{pot}_t(l,u) + \text{pot}_t(l',v) \\
&= \text{travelTime}(l) + \text{len}(u,v) + c_2^b - \text{travelTime}(l) \\
&\quad - (\text{remBreakTime}(l,u) + \text{chPot}_t(u)) + (\text{remBreakTime}(l',v) + \text{chPot}_t(v)) \\
&= \text{len}(u,v) + c_2^b + \text{remBreakTime}(l',v) \\
&\quad - \text{remBreakTime}(l,u) - \text{chPot}_t(u) + \text{chPot}_t(v) \\
&= \text{len}(u,v) + c_2^b + \text{breakEstimate}_2(l',v) \cdot c_2^b + \text{breakEstimate}_1(l',v) \cdot c_1^b \\
&\quad - \text{remBreakTime}(l,u) - \text{chPot}_t(u) + \text{chPot}_t(v) \\
&= \text{len}(u,v) + c_2^b + \text{minBreaks}_2(0 + \text{chPot}_t(v)) \cdot c_2^b \\
&\quad + (\text{minBreaks}_1(0 + \text{chPot}_t(v)) - \text{minBreaks}_2(0 + \text{chPot}_t(v))) \cdot c_1^b \\
&\quad - \text{remBreakTime}(l,u) - \text{chPot}_t(u) + \text{chPot}_t(v) \\
&\overset{(7.)}{\geq} \text{len}(u,v) + \text{minBreaks}_2(c_2^d + \text{chPot}_t(v)) \cdot c_2^b \\
&\quad + (\text{minBreaks}_1(c_1^d + \text{chPot}_t(v)) - \text{minBreaks}_2(c_2^d + \text{chPot}_t(v))) \cdot c_1^b \\
&\quad - \text{remBreakTime}(l,u) - \text{chPot}_t(u) + \text{chPot}_t(v) \\
&\overset{(6.\text{ and }8.)}{\geq} \text{len}(u,v) + \text{minBreaks}_2(\text{breakDist}_2(l) + \text{len}(u,v) + \text{chPot}_t(v)) \cdot c_2^b \\
&\quad + (\text{minBreaks}_1(\text{breakDist}_1(l) + \text{len}(u,v) + \text{chPot}_t(v)) \\
&\quad - \text{minBreaks}_2(\text{breakDist}_2(l) + \text{len}(u,v) + \text{chPot}_t(v))) \cdot c_1^b \\
&\quad - \text{remBreakTime}(l,u) - \text{chPot}_t(u) + \text{chPot}_t(v) \\
&\overset{(5.\text{ and }6.)}{\geq} \text{len}(u,v) + \text{minBreaks}_2(\text{breakDist}_2(l) + \text{chPot}_t(u)) \cdot c_2^b \\
&\quad + (\text{minBreaks}_1(\text{breakDist}_1(l) + \text{chPot}_t(u)) \\
&\quad - \text{minBreaks}_2(\text{breakDist}_2(l) + \text{chPot}_t(u))) \cdot c_1^b \\
&\quad - \text{remBreakTime}(l,u) - \text{chPot}_t(u) + \text{chPot}_t(v) \\
&= \text{len}(u,v) + \text{remBreakTime}(l,u) - \text{remBreakTime}(l,u) - \text{chPot}_t(u) + \text{chPot}_t(v) \\
&= \text{len}(u,v) - \text{chPot}_t(u) + \text{chPot}_t(v) \\
&\overset{(4.)}{\geq} 0
\end{aligned}
\tag{5.16}
$$

□

5.4. Bidirectional Goal-Directed Search with Multiple Driving Time Constraints

We now extend the goal-directed approach of Section 5.3.1 to a bidirectional approach. Our algorithm will consist of a forward search from s in $\overrightarrow{G}$ and a backward search from t in $\overleftarrow{G}$. The distances of the forward and backward search are combined at nodes were which were settled by both searches. We therefore introduce a concept to merge the label sets of backward and forward search to find the best currently known valid route using information of both searches. Since we aim to stop the search as early as possible, we have to decide on a stopping criterion which allows the search to stop way before the forward search settles the target node t or the backward search settles s. The correctness of the stopping criterion is closely tied to the potential of Section 5.3.1.

The input of the search remains a graph $G = (V, E, \text{len})$, a set of parking nodes $P \subseteq V$, a set of driving time constraints C, and start and target nodes $s, t \in V$. There are two

potentials $\overrightarrow{\text{pot}}_t$ and $\overleftarrow{\text{pot}}_s$ which we call the forward and the backward potential. The forward potential yields lower bounds for the remaining travel time of a label to t in $\overrightarrow{G} = G$. The backward potential yields lower bounds for the remaining travel time of a label to s in $\overleftarrow{G}$. The forward search then is a normal A* search on $\overrightarrow{G}$ with start node s and target node t and the backward search is a normal A* search on $\overleftarrow{G}$ with start node t and target node s. Each search owns a queue of labels $\overrightarrow{Q}$ and $\overleftarrow{Q}$ and a label set $\overrightarrow{L}(v)$, respectively $\overleftarrow{L}(v)$ for each $v \in V$.

During the search, forward and backward search alternately settle nodes until the stopping criterion is met, one search completed the search by itself, or the queues ran empty. We hold the tentative value for $\mu_{tt}(s,t)$ in a variable $tent(s,t)$ which we initialize with ∞ before settling the first node. When forward or backward search settles a node v, they additionally check if the label set of the other search at v contains any settled labels. If this is the case, forward and backward search met at this node. We then search for the combination of labels which yields the shortest valid route between s and t via v. In other words, we want to find the labels $l \in \overrightarrow{L}(v)$ and $m \in \overleftarrow{L}(v)$ which minimize $\text{travelTime}(l) + \text{travelTime}(m)$ and for which $\text{breakDist}_1(l) + \text{breakDist}_1(m) < c_1^d$ and $\text{breakDist}_1(l) + \text{breakDist}_1(m) < c_2^d$. If the resulting distance for an s-t path via v is smaller than the previously known minimum tentative distance $tent(s,t)$, we update $tent(s,t)$ accordingly. We stop the forward search if the minimum key of $\overrightarrow{Q}$ is greater than $tent(s,t)$ and stop the backward search when the minimum key of $\overleftarrow{Q}$ is greater than $tent(s,t)$.

Theorem 5.8. *At the point in time when forward and backward search have stopped, $tent(s,t) = \mu_{tt}(s,t)$. In other words, when the search stops, $tent(s,t)$ equals the minimum travel time from s to t which complies with the driving time constraints C.*

Proof. We show that when the search stops, all valid s-t routes q which comply with the driving time constraints C and which were not found yet yield a larger travel time $\text{travelTime}(q)$ than the current $tent(s,t)$. This also implies that if $tent(s,t) = \infty$ at the point in time when the search stops, there exist no paths from s to t.

Since the search stops when the minimum keys of $\overrightarrow{Q}$ and $\overleftarrow{Q}$ are both greater than $tent(s,t)$, all labels l which will be settled by continuing the search have a greater distance $\text{travelTime}(l)$. Therefore, if any new connection between forward and backward search which complies with the driving time constraints will be found, its distance will be greater than $tent(s,t)$.

The shortest path with travel time $\mu_{tt}(s,t)$ consists of two subpaths of forward and backward search which were connected at a node v. There exist label $l \in \overrightarrow{L}$ and $m \in \overleftarrow{L}$ with $\text{travelTime}(l) + \text{travelTime}(m) = \mu_{tt}(s,t)$. Each label is the result of a unidirectional search from s, respectively from t. In Section 5.1.4 we proved that a unidirectional search can be stopped when the first label at its target node was removed from the queue. Since l and m are both smaller or equal to $tent(s,t)$ and both queue keys of forward and backward queue are greater, l and m where already removed from the respective queue. Therefore, we know that $\overrightarrow{L}(v)$ and $\overleftarrow{L}(v)$ contain the labels with the shortest distance from s to v, respectively from t to v. Consequently, when the second of both labels was settled at v, $\mu_{tt}(s,t)$ was updated with the value $\text{travelTime}(l) + \text{travelTime}(m)$. □

5.5. Goal-Directed Core Contraction Hierarchy Search

Given a graph $G = (V, E, \text{len})$, we construct a core contraction hierarchy in which the core contains all the parking nodes $P \subseteq V$. We denote the set of uncontracted core nodes as $C \subseteq V$. It is $P = C$. The set of nodes V therefore is split into a set of core nodes C and a

set of contracted nodes $V^+ = V \setminus C$. The graph $G^+ = (V^+, E^+, \text{len})$ with nodes V^+ and edges $E^+ = \{(u, v) \in E \mid u, v \in V^+\}$ therefore is a valid contraction hierarchy. It contains only contracted nodes and edges between those nodes. Thus, E^+ contains only upward edges. The graph $G_C = (V \setminus V^+, E \setminus E^+, \text{len})$ is called the core graph.

The goal-directed core CH query reuses the bidirectional, goal-directed approach of Section 5.4 on a modified forward graph $\overrightarrow{G^*}$ and a modified backward graph $\overleftarrow{G^*}$. The forward graph $\overrightarrow{G^*} = (V, \overrightarrow{E}^*, \text{len})$ consists of the forward graph of the contraction hierarchy $\overrightarrow{G}_{CH}$, extended by the core graph G_C. It is $\overrightarrow{E}^* = \overrightarrow{E}_{CH} \cup E_C$. Equivalently, the backward graph is defined as $\overleftarrow{G^*} = (V, \overleftarrow{E}^*, \overleftarrow{\text{len}})$ with $\overleftarrow{E}^* = \overleftarrow{E}_{CH} \cup E_C$ and $\overleftarrow{\text{len}}(u, v) = \text{len}(v, u)$.

The bidirectional query in G^* consequently consists of a forward search from s in $\overrightarrow{G^*}$ and backward search from t in $\overleftarrow{G^*}$. Each search consists of two parts, an upward search in G^+ and a search in G_C. A search may begin at a core node skip the upward part, it also may not reach the core graph at all and only perform the CH upward search.

The stopping criterion of the bidirectional search must take into account that the graph G^* contains the contraction hierarchy G^+. The common stopping criterion for *s-t* queries in a CH is to stop the search if the minimum queue key of both queues $\overrightarrow{Q}$ and $\overleftarrow{Q}$ is greater or equal to the tentative minimum distance $\mu(s, t)$ [GSSV12]. Since this criterion is a valid stopping criterion for the bidirectional A* algorithm, we can use it for our combination of CH and bidirectional A*.

Correctness

Proof. We simply derive the correctness of the core contraction hierarchy search from the correctness of a CH search and the correctness of the bidirectional A*. The label set of a node $v \in C_{CH}$ can never contain more than one label for the forward search and one label for the backward search. Let C' be the set of nodes which are reachable from another core node, i.e. $C' = C \cup \{v \mid (u, v) \in E \wedge u \in C\}$.

Case 1: Neither forward nor backward search settle a node in C'. No parking is involved since $P \subseteq C$. The query constitutes a simple CH query without labels, extended by pruning with the driving time constraints and goal-direction. The correctness directly follows from the correctness of the bidirectional A* algorithm with driving time constraints as shown in Section 5.3.

Case 2: Forward or backward search settle a node in C', but not both. No valid *s-t* route exists per definition of C'. The correctness of the query is trivial since forward and backward search cannot settle a common node.

Case 3: Both forward and backward search settle a node in C'. The query continues in the core as a bidirectional goal-directed search as described in Section 5.3. Since the chosen stopping criterion is valid for this case, the result will be correct. □

5.6. Improvements and Implementation

In this section, we describe modifications to the algorithm described in Section 5.5 which enable further improvements of running time.

Label Queue In Section 5.1, we introduced our basic approach as an algorithm which maintains one queue Q of labels in contrast to Dijkstra's algorithm, which maintains one queue of nodes. In practice, we revert this change and Q becomes a queue of nodes again. The key of a node v in Q is the best known travel time of a label in $L(v)$. The label set $L(v)$ now becomes a priority queue of labels itself. When settling a label, we remove a

node v from Q and the best label l at v from $L(v)$. If $L(v)$ is not empty now, we insert the node v into Q again with the travel time of the new best label $l' \in L(v)$ as the key.

Bidirectional Backward Pruning When running the bidirectional A* algorithm in combination with CH-Potentials, the progress and the potential of the backward search can be used to prune the forward search and vice versa. This was introduced by [SZ18] for node potentials. We will adapt the concept for the use with labels and the potential as defined in Section 5.3.1.

A label l at a node v which was propagated along an edge (u, v) can be discarded if we can proof that all routes using the label are longer or equal to the tentative travel time $tent(s, t)$, i.e., the shortest currently known travel time from s to t. The travel time travelTime(l) of the label l at v is already known and it remains to find a lower bound for the remaining distance to t. We will describe the pruning of the forward search, the backward search can be pruned accordingly.

The backwards queue $\overleftarrow{Q}$ contains labels $l' \in \overleftarrow{L}(v)$ with travelTime(l') + $\text{pot}_s(l', v)$ as the queue key. Labels are removed from the queue with an increasing key. If a label l' was not yet removed from the backwards queue, we know that travelTime(l') + $\text{pot}_s(l', v) \geq \text{minKey}(\overleftarrow{Q})$ holds which gives us a lower bound for the travel time of the label travelTime(l') $\geq \text{minKey}(\overleftarrow{Q}) - \text{pot}_s(l', v)$. Unfortunately, we can only compute $\text{pot}_s(l', v)$ if the label l' was already inserted into the queue. We need to resort to a pure node potential which is not a function of labels for the case when l' was not yet inserted into the queue. Such a node potential is the potential $\text{pot}'_t(v)$ of Section 5.2.1 where we assessed it to be infeasible for the use in the A* algorithm, but $\text{pot}'_t(v)$ is a lower bound for the potential $\text{pot}_t(l, v)$ with $l \in L(v)$. We do not use it in queue keys but only as a lower bound of travelTime(l').

Now consider a forward label $l \in \overrightarrow{L}(v)$ at the node v. The label l has a distance since the last break which is zero or greater while l' started at t with a distance of zero since the last break. Therefore, the remaining travel time of l from v to t must be greater or equal to travelTime(l') which gives us a lower bound for the remaining travel time of the label l to t of travelTime(l) $\geq$ travelTime(l') $\geq \text{minKey}(\overleftarrow{Q}) - \text{pot}_s(l', v)$. The lower bound for an s-t path using l becomes travelTime(l) + $\text{minKey}(\overleftarrow{Q}) - \text{pot}_s(l', v)$. We do not insert the label l into the label set of v if $tent(s, t) \leq \text{travelTime}(l) + \text{minKey}(\overleftarrow{Q}) - \text{pot}_s(l', v)$ because the label l cannot lead to an improvement of the shortest known s-t travel time.

Core Contraction Hierarchy Stopping Criteria Only a subset of nodes of a core CH can be reachable from a core node. This includes all the core nodes and the last layer of contracted nodes of the CH, i.e. all nodes which are contracted themselves but are reachable through an outgoing edge of a core node. If the forward search stops without touching any of the nodes which are reachable from the core, the forward search can only connect with the backwards search within the fully constructed CH part of the core CH. If the backwards search has no contracted nodes left in its queue we can be sure, that forward and backward search will not connect anymore and that no path will be found. Therefore, we can terminate the search. Equivalently, we can terminate the search if the backward search terminates early without touching any nodes which are reachable from the core and the forward search has no uncontracted nodes left in its queue.

Constructing the Core Contraction Hierarchy In chapter 5.5 we introduced a core contraction hierarchy algorithm with a core of uncontracted nodes C which contains all the parking nodes P. It remains to determine if $C = P$ is the optimal choice for C or if it is beneficial to include more nodes in the core. In general, the parking nodes are not the most important nodes in the graph according to the node order of the CH. Because

we declare them as core nodes and contract everything but those nodes, we act as if they were the set of most important nodes which breaks the node order and leads to a CH of lower quality. The idea is to include the most important nodes according to the node order in the core to obtain a CH of higher quality, i.e. with lower node degrees. An additional advantage are shorter build times for the core CH. TODO finish

6. Evaluation

In this section, we evaluate the running time and behavior of our algorithms of Chapter 5. Our machine runs openSUSE Leap 15.3, has 128 GB (8x16 GB) of 2133 MHz DDR4 RAM, and a 4-core Intel Xeon E5-1630v3 CPU which runs at 3.7 GHz. The code is written in Rust and compiled with cargo 1.64.0-nightly using the release profile with `lto = true` and `codegen-units = 1`.

Data. Our data is a road network of Europe[1] and of Germany[2] from Open Street Map (OSM). We extract the routing graph and parking nodes from the OSM data using a custom extension[3] of RoutingKit[4]. The obtained routing graph of Europe has 81.5 million nodes and 190 million edges. If not stated otherwise, our set P of parking nodes in the European routing graph consists of 6796 nodes which were selected due to their OSM attributes. Equivalently, the routing graph of Germany has 12.5 million nodes, 29.5 million edges, and we selected 3222 nodes as parking nodes.

Methodology. All experiments are run sequentially. We conduct experiments regarding the preprocessing time of the core CH and the running time of queries on the extracted routing graph. We average preprocessing running times over 10 runs and running times of s-t queries over 10 000 queries with s and t independently chosen uniformly at random for each query. If not stated otherwise, we use c_1 with $c_1^d = 4.5\,\text{h}$ and $c_1^b = 0.45\,\text{min}$ and c_2 with $c_2^d = 9\,\text{h}$ and $c_2^b = 9\,\text{h}$ to approximate the regulations of the EU.

6.1. Algorithms and Optimizations

We evaluate the different algorithms of Chapter 5 and the optimizations of Chapter 5.6.

First, we compare the running times of queries of the algorithms of Chapter 5 on a German and European road network. We scale the experiment down from the European road network because some variants of the algorithms in this experiment cannot keep up with the performance of the goal-directed core CH algorithm and would render the experiment slow and impracticable. It also allows observing how well the algorithms scale on larger road networks, i.e., on longer routes.

[1] `https://download.geofabrik.de/europe-latest.osm.pbf` of March 22, 2022
[2] `https://download.geofabrik.de/europe/germany-latest.osm.pbf` of March 22, 2022
[3] `https://github.com/maxoe/RoutingKit`
[4] `https://github.com/RoutingKit/RoutingKit`

As Table 6.1 shows, on the German road network, the goal-directed search performs an order of magnitude better than the baseline Dijkstra's algorithm with our amendments for driving time constraints. The bidirectional search without goal-direction performs worse than the baseline. The best result shows the goal-directed bidirectional algorithm and the two core CH variants with the not goal-directed core CH algorithm falling back significantly. The best result of the goal-directed core CH improves the baseline by a factor of 10 000.

On the European road network, we omitted the baseline since it is too slow. Also, the TODO

			Germany [ms]		Europe [ms]	
Goal-Directed	Bidirectional	Core CH	1-DTC	2-DTC	1-DTC	2-DTC
✗	✗	✗	35110.14	44083.92	-	-
✓	✗	✗	1393.72	7.87	15055.56	-
✗	✓	✗	50463.27	-	-	-
✓	✓	✗	4.38	14.56	30372.80	-
✗	✓	✓	121.29	182.92	547.64	720.72
✓	✓	✓	3.47	4.15	126.06	339.97

Table 6.1.: Average running times of random queries on a German and European road network with one or two driving time constraints.

The variants of the algorithm without the core CH suffer from outliers with a very long running time.

Most of the performance gain of the goal-directed search originates from the very tight lower-bound given by the CH potentials. If the shortest travel time between two nodes s and t is way larger than $\mu(s,t)$ due to necessary breaks and even detours to parking nodes on the route, then the performance of the goal-directed search degrades. The bidirectional variant can mitigate this disadvantage since it connects two routes which each for itself have fewer breaks on the route. A disadvantage from the goal-directed search which cannot be mitigated by the bidirectional variant are its outliers. In cases where the algorithm is not able to find a route or the route needs a lot of breaks, the running time increases significantly. TODO

TODO outlier problem with graphics leas to non ch excluded even if fast

Second, we evaluate the different optimizations as described in chapter 5.6 in use with the goal-directed core CH algorithm on the European road network.

		Running Time [ms]	
Backward Pruning	Additional Core CH Stopping Criteria	1-DTC	2-DTC
✗	✗	152.64	295.18
✓	✗	137.15	273.07
✗	✓	151.87	301.29
✓	✓	151.62	303.12

Table 6.2.: Comparison of running times of the goal-directed core CH algorithm with different optimizations from Section5.6.

6.2. Goal-Directed Core CH Queries

Finally, we investigate queries of the goal-directed core CH algorithm with full optimizations more closely. What drives the running time of the algorithm? TODO

- different len
- variable break time, driving time limit
- parking node set

7. Conclusion

Summary and outlook.

Bibliography

[ADGW12] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Hierarchical Hub Labelings for Shortest Paths. In *Proceedings of the 20th Annual European Symposium on Algorithms (Esa'12), Volume 7501 of Lecture Notes in Computer Science*, pages 24–35. Springer, 2012.

[BDG+15] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route Planning in Transportation Networks, April 2015.

[BDSV09] G. Veit Batz, Daniel Delling, Peter Sanders, and Christian Vetter. Time-Dependent Contraction Hierarchies. In Irene Finocchi and John Hershberger, editors, *2009 Proceedings of the Eleventh Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 97–105. Society for Industrial and Applied Mathematics, Philadelphia, PA, January 2009.

[BGSV13] G. Veit Batz, Robert Geisberger, Peter Sanders, and Christian Vetter. Minimum time-dependent travel times with contraction hierarchies. *Journal of Experimental Algorithmics (JEA)*, April 2013.

[Bom20] Stefan Bomsdorf. Exact and Heuristic Solution of the Time-Dependent Truck Driver Scheduling and Routing Problem with Two Types of Breaks. Master's thesis, RWTH Aachen, Aachen, 2020.

[Brä16] Christian Bräuer. *Optimale zeitabhängige Pausenplanung für LKW-Fahrer mit integrierter Parkplatzwahl.* Bachelor's thesis, Karlsruhe Institute of Technology, 2016.

[DGNW11] Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. PHAST: Hardware-Accelerated Shortest Path Trees. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 921–931, May 2011.

[DGPW14] Daniel Delling, Andrew V Goldberg, Thomas Pajor, and Renato F. Werneck. Robust Exact Distance Queries on Massive Networks. In *Proceedings of the 22nd Annual European Symposium on Algorithms (ESA'14), Volume 8737 of Lecture Notes in Computer Science*, pages 321–333. Springer, 2014.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, December 1959.

[DSW15] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable Contraction Hierarchies, August 2015.

[GH05] Andrew V. Goldberg and Chris Harrelson. Computing the shortest path: A* search meets graph theory. pages 156–165, 2005.

[GK12] Asvin Goel and Leendert Kok. Truck Driver Scheduling in the United States. *Transportation Science*, 46(3):317–326, 2012.

[Goe09] Asvin Goel. Vehicle Scheduling and Routing with Drivers' Working Hours. *Transportation Science*, 43(1):17–26, 2009.

[Goe12] Asvin Goel. The Minimum Duration Truck Driver Scheduling Problem, May 2012.

[GSSV12] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact Routing in Large Road Networks Using Contraction Hierarchies. *Transportation science*, 46(3):388, 2012.

[GV11] Robert Geisberger and Christian Vetter. Efficient Routing in Road Networks with Turn Costs. In Panos M. Pardalos and Steffen Rebennack, editors, *Experimental Algorithms*, Lecture Notes in Computer Science, pages 100–111, Berlin, Heidelberg, 2011. Springer.

[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, July 1968.

[KBS+17] Alexander Kleff, Christian Bräuer, Frank Schulz, Valentin Buchhold, Moritz Baum, and Dorothea Wagner. Time-Dependent Route Planning for Truck Drivers. In Tolga Bektaş, Stefano Coniglio, Antonio Martinez-Sykora, and Stefan Voß, editors, *Computational Logistics*, Lecture Notes in Computer Science, pages 110–126, Cham, 2017. Springer International Publishing.

[Kle19] Alexander Kleff. *Scheduling and Routing of Truck Drivers Considering Regulations on Drivers' Working Hours*. PhD thesis, Karlsruhe Institute of Technology, 2019.

[KSWZ20] Alexander Kleff, Frank Schulz, Jakob Wagenblatt, and Tim Zeitz. Efficient Route Planning with Temporary Driving Bans, Road Closures, and Rated Parking Areas. page 13, 2020.

[MDGCNR20] Sérgio Fernando Mayerle, Daiane Maria De Genaro Chiroli, João Neiva de Figueiredo, and Hidelbrando Ferreira Rodrigues. The long-haul full-load vehicle routing and truck driver scheduling problem with intermediate stops: An economic impact evaluation of Brazilian policy. *Transportation Research Part A: Policy and Practice*, 140:36–51, October 2020.

[Par06] European Parliament. Regulation (ec) no 561/2006 of the european parliament and of the council of 15 march 2006 on the harmonisation of certain social legislation relating to road transport and amending council regulations (eec) no 3821/85 and (ec) no 2135/98 and repealing council regulation (eec) no 3820/85. *Official Journal of the European Union*, 2006.

[Sha08] Vidit Divyang Shah. Time dependent truck routing and driver scheduling problem with hours of service regulations. 2008.

[SSVB07] Carlo Sartori, Pieter Smet, and Greet Vanden Berghe. Scheduling hours of service for truck drivers with interdependent routes, 20210107.

[SZ18] Ben Strasser and Tim Zeitz. Using Incremental Many-to-One Queries to Build a Fast and Tight Heuristic for A* in Road Networks. 2018.

[SZ21] Ben Strasser and Tim Zeitz. A Fast and Tight Heuristic for A* in Road Networks. page 16, 2021.

[vdTdWB18] Marieke van der Tuin, Mathijs de Weerdt, and G. Batz. Route Planning with Breaks and Truck Driving Bans Using Time-Dependent Contraction Hierarchies. *Proceedings of the International Conference on Automated Planning and Scheduling*, 28:356–364, June 2018.

Appendix

A. List of Abbreviations

SPP	Shortest Path Problem
TDSP	Truck Driver Scheduling Problem
MD-TDSP	Minimum Duration Truck Driver Scheduling Problem
TDSRP	Truck Driver Scheduling and Routing Problem
TD-TDSRP	Time-Dependent Truck Driver Scheduling and Routing Problem
TD-TDSRP-2B	Time-Dependent Truck Driver Scheduling and Routing Problem with two types of breaks (two types of driving time restrictions)
LH-TDRP	Long-Haul Truck Driver Routing Problem
OSM	Open Street Map
1-DTC (2-DTC)	Restriction of the long-haul truck driver routing problem to only one (two) driving time constraint(s)
CH	Contraction Hierarchy

B. List of Symbols and Designations

We use Greek symbols for theoretical and abstract concepts such as the shortest distance between two nodes or an arbitrary valid node potential. Concrete functions and procedures, i.e., those with definitions, have verbose names. Algorithmic functions and procedures for which pseudocode is provided use SmallCaps.

$\pi_t(v)$	A node potential to t
$\mu(s,t)$	Shortest distance between s and t
$\mu_{tt}(s,t)$	Shortest travel time between s and t
r	A route consisting of a path and a function breakTime(v)
c	A driving time constraint
c^d	Maximum allowed driving time of a driving time constraint c
c^b	Minimum pause time of a driving time constraint c
C	A set of one or more driving time constraints
$L(v)$	The label set of a node v
breakDist$_i(l)$	Distance since the last break of a label with a duration of at least c_i^b or distance since the last break if the index i is omitted
pred(l)	Predecessor label of a label l
travelTime(l)	Travel time of a label l
pot(l,v)	The potential of a label $l \in L(v)$ as defined in Section 5.3.1
len((u,v))	The weight, respectively length of an edge (u,v), alternatively used for the length len(p) of a path p
G^+	A contraction hierarchy with vertices V^+ and edges E^+
G^*	A core contraction hierarchy with vertices V^* and edges E^*