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, June 15, 2022

Abstract

A short summary of what is going on here.

Deutsche Zusammenfassung

Kurze Inhaltsangabe auf deutsch.

Contents

1. Introduction

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 and Related Work

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 to 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, $\overleftarrow{G} = (V, \overleftarrow{E}, \overleftarrow{\mathrm{len}})$ with $\overleftarrow{E} = \{(v,u) \in V \times V \mid (u,v) \in E\}$ and $\overleftarrow{\mathrm{len}}(u,v) = \mathrm{len}(v,u)$. 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 alternatively 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 only 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.

2.1. Contraction Hierarchies

2.2. CH Potential

2.3. Core Contraction Hierarchies

3. 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 $\text{breakTime}\colon p \to \{0, c_1^d, \ldots, c_{|C|}^d\}$ 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 3.1 (Valid Route)**.** *A valid route $p = \langle s = v_0, v_1, ..., t = v_k \rangle$ 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 differentiate between the driving time and the travel time of a route.

Definition 3.2 (Driving Time of a Route)**.** *The driving time* $\text{drivingTime}(r)$ *of a route r is the length* $\text{len}(p)$ *of the path $p = \langle s = v_0, v_1, ..., t = v_k \rangle$ of the route.*

Definition 3.3 (Travel Time of a Route)**.** *The travel time* $\text{travelTime}(r)$ *of a route r is the sum of driving time* $\text{drivingTime}(r)$ *and break time* $\text{breakTime}(p)$ *on its path.*

Definition 3.4 (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 $\mu_{tt}(s,t)$. Accordingly, $\mu_{dt}(s,t)$ denotes the driving time of the shortest route between s and t. In general, the driving time $\mu_{dt}(s,t)$ and the distance $\mu(s,t)$ between two nodes are not equal.

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.

4. 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.

4.1. 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 driving time since the last break
- pred(l), its preceding label

4.1.1. 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 4.1.

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.

```
1 Procedure SettleNextLabel():
2     $l \leftarrow$ Q.deleteMin()
3     forall $(u, v) \in E$ do
4         RelaxEdge($(u, v), l$)
```

Figure 4.1.: Settling a label $l \in L(u)$ removes the label from the queue and relaxes all the outgoing edges of u.

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 4.1 (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 4.2 (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 4.2 we define the procedure RemoveDominated(l) as an operation on a label set.

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

Figure 4.2.: The procedure L.RemoveDominated(l) removes all labels from the label set L which are dominated by the label l.

4.1.2. 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

Algorithm 4.1: Dijkstra+1-DTC

```
Input: Graph $G = (V, E, \mathrm{len})$, 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 $\mathsf{L}(v)$ of labels for all $v \in V$
Output: Shortest route with $\mathrm{travelTime}(j) = \mu_{tt}(s, t)$

// Initialization
1  Q.queueInsert($0, (0, 0, \perp)$)
2  $\mathsf{L}(s)$.insert($(0, 0, \perp)$)

// Main loop
3  while Q is not empty do
4      SettleNextLabel()
5      if label at $t$ was settled then
6          return
```

better solution, we generate both labels and add them to label set $L(v)$ and the queue Q. We now can define the procedure RelaxEdge as in Figure 4.3.

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

Figure 4.3.: Relaxing an edge $(u, v) \in E$ when settling a label $l \in L(u)$ with regard to parking nodes.

4.1.3. Initialization and Stopping Criterion

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 4.1 Dijkstra+1-DTC is now trivial.

4.1.4. Correctness

An s-t query with Dijkstra's algorithm can be stopped when t was removed from the queue since all of the following nodes in the queue have larger distances and 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 labels in the queue are ordered by their travel time. Relaxing an edge can only increase the travel time since both edge lengths and break times are non-negative. Therefore, the same argument as for Dijkstra's algorithm applies and the first label at t which was removed from the queue contains the shortest travel time $\mu_{tt}(s, t)$.

Algorithm 4.2: 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
1 $l_s \leftarrow (0, 0, \perp)$
2 Q.queueInsert($\text{pot}_t((l_s), s)$, $l_s$)
3 L($s$).insert($l_s$)

// Main loop
4 while Q is not empty do
5     settleNextNode()
6     if minimum of Q is label at $t$ then
7         return
```

4.2. Goal-Directed Search with One Driving Time Constraint

In this section, we transform the base algorithm described in section 4.1 to a goal-directed search with the A* algorithm. We introduce a new potential pot_t which is based on the CH potential chPot_t which is described in section 2.2. 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 $\text{travelTime}(l) + \text{pot}_t(l, v)$. As shown in algorithm 4.2, the adaption of the pseudocode of the coarse algorithm is trivial.

The only thing left is the adaption of RelaxEdge in Figure 4.3 where we change the queue keys to use $\text{travelTime}(l) + \text{travelTime}(l, v)$ instead. The result is shown in Figure 4.4.

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

Figure 4.4.: Relaxing an edge with regard to the potential.

4.2.1. Potential for One Driving Time Constraint

Given a target node t, the CH potential $\text{chPot}_t(v)$ yields a perfect estimate for the distance $\mu(v, t)$ from v to t without regard for driving time constraints and breaks. This trivially 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 which calculates the minimum amount of necessary breaks given a driving time d.

$$\text{minBreaks}(d) = \begin{cases} \left\lceil \frac{d}{c^d} \right\rceil - 1 & d > 0 \\ 0 & else \end{cases} \tag{4.1}$$

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 a driving time d

$$\text{minBreakTime}(d) = \text{minBreaks}(d) \cdot c^b \tag{4.2}$$

and finally define our node potential as

$$\begin{aligned} \text{pot}'_t(v) &= \text{minBreakTime}(d) + \text{chPot}_t(v) \\ &= \text{minBreakTime}(d) + \mu(v,t) \end{aligned} \tag{4.3}$$

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

$$len(u,v) - \text{pot}_t(u) + \text{pot}_t(v) \geq 0 \quad \forall (u,v) \in E \tag{4.4}$$

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. 4.5 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 $len(s,v) = 2$, $len(s,v) - \text{pot}'_t(s) + \text{pot}'_t(v) = -1$.

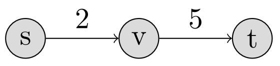

Figure 4.5.: A graph for which the feasibility condition of equation 4.4 does not always hold with the potential pot′.

A variant of the potential accounts for the driving time since the last break of a label breakDist(l) to calculate the minimum required break time on the v-t path.

$$\begin{aligned} \text{pot}_t(l,v) &= \text{minBreakTime}(\text{breakDist}(l) + \text{chPot}(v)) + \text{chPot}(v) \\ &= \text{minBreakTime}(\text{breakDist}(l) + \mu(v,t)) + \mu(v,t) \end{aligned} \tag{4.5}$$

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 inequality 4.4 can no longer be applied. We therefore have to show that queue keys of labels, which represent a lower bound estimate for the travel time of the entire route, can only increase over time.

Lemma 4.3. *Let $p = \langle s = v_0, v_1, \dots, t = v_k \rangle$ be a path with labels l_i at nodes v_i. Then* $\mathrm{travelTime}(l_{i-1}) + \mathrm{pot}_t(l_{i-1}, v_{i-1}) \leq \mathrm{travelTime}(l_i) + \mathrm{pot}_t(l_i, v_i)$.

Proof. Let $(u, v) \in E$ be an edge. The procedure RelaxEdge in Figure 4.4 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. $d \geq d' \implies \mathrm{minBreakTime}(d) \geq \mathrm{minBreakTime}(d')$
2. $\mathrm{minBreakTime}(d + c^d) = c^b + \mathrm{minBreakTime}(d)$
3. $c^d \geq \mathrm{breakDist}(l') \geq \mathrm{breakDist}(l) + \mathrm{len}(u, v) \geq \mathrm{breakDist}(l)$
   (line 2 in RelaxEdge in Figure 4.4)
4. $\mathrm{len}(u, v) - \mathrm{chPot}_t(u) + \mathrm{chPot}_t(v) \geq 0$ (feasibility of the CH potential)
5. $\mathrm{len}(u, v) + \mathrm{chPot}_t(v) \geq \mathrm{chPot}_t(u)$

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

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

$$\begin{aligned}
&\mathrm{travelTime}(l') - \mathrm{travelTime}(l) - \mathrm{pot}_t(l, u) + \mathrm{pot}_t(l', v)\\
&= \mathrm{travelTime}(l) + \mathrm{len}(u, v) - \mathrm{travelTime}(l)\\
&\qquad - (\mathrm{minBreakTime}(\mathrm{breakDist}(l) + \mathrm{chPot}(u)) + \mathrm{chPot}(u))\\
&\qquad + \mathrm{minBreakTime}(\mathrm{breakDist}(l') + \mathrm{chPot}(v)) + \mathrm{chPot}(v)\\
&= \mathrm{len}(u, v) + \mathrm{minBreakTime}(\mathrm{breakDist}(l') + \mathrm{chPot}(v))\\
&\qquad - \mathrm{minBreakTime}(\mathrm{breakDist}(l) + \mathrm{chPot}(u)) - \mathrm{chPot}(u) + \mathrm{chPot}(v)\\
&= \mathrm{len}(u, v) + \mathrm{minBreakTime}(\mathrm{breakDist}(l) + \mathrm{len}(u, v) + \mathrm{chPot}(v))\\
&\qquad - \mathrm{minBreakTime}(\mathrm{breakDist}(l) + \mathrm{chPot}(u)) - \mathrm{chPot}(u) + \mathrm{chPot}(v)\\
&\overset{(1.\text{ and }5.)}{\geq} \mathrm{len}(u, v) + \mathrm{minBreakTime}(\mathrm{breakDist}(l) + \mathrm{chPot}(u))\\
&\qquad - \mathrm{minBreakTime}(\mathrm{breakDist}(l) + \mathrm{chPot}(u)) - \mathrm{chPot}(u) + \mathrm{chPot}(v)\\
&= \mathrm{len}(u, v) - \mathrm{chPot}(u) + \mathrm{chPot}(v)\\
&\overset{(4.)}{\geq} 0
\end{aligned} \tag{4.6}$$

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

$$\begin{aligned}
&\text{travelTime}(l') - \text{travelTime}(l) - \text{pot}_t(l,u) + \text{pot}_t(l',v) \\
&= \text{travelTime}(l) + \text{len}(u,v) + c^b - \text{travelTime}(l) \\
&\quad - (\text{minBreakTime}(\text{breakDist}(l) + \text{chPot}(u)) + \text{chPot}(u)) \\
&\quad + \text{minBreakTime}(\text{chPot}(v)) + \text{chPot}(v) \\
&= \text{len}(u,v) + c^b + \text{minBreakTime}(\text{chPot}(v)) \\
&\quad - \text{minBreakTime}(\text{breakDist}(l) + \text{chPot}(u)) - \text{chPot}(u) + \text{chPot}(v) \\
&\overset{(2.)}{=} \text{len}(u,v) + \text{minBreakTime}(c^d + \text{chPot}(v)) \\
&\quad - \text{minBreakTime}(\text{breakDist}(l) + \text{chPot}(u)) - \text{chPot}(u) + \text{chPot}(v) \qquad (4.7) \\
&\overset{(1.\text{ and }3.)}{\geq} \text{len}(u,v) + \text{minBreakTime}(\text{breakDist}(l) + \text{len}(u,v) + \text{chPot}(v)) \\
&\quad - \text{minBreakTime}(\text{breakDist}(l) + \text{chPot}(u)) - \text{chPot}(u) + \text{chPot}(v) \\
&\overset{(1.\text{ and }4.)}{\geq} \text{len}(u,v) + \text{minBreakTime}(\text{breakDist}(l) + \text{chPot}(u)) \\
&\quad - \text{minBreakTime}(\text{breakDist}(l) + \text{chPot}(u)) - \text{chPot}(u) + \text{chPot}(v) \\
&= \text{len}(u,v) - \text{chPot}(u) + \text{chPot}(v) \\
&\overset{(4.)}{\geq} 0
\end{aligned}$$

□

Lemma 4.4. *The sum* travelTime(l) + pot$_t$(l, v) *of a label l at a node v is a lower bound for the travel time from s to t using l.*

Proof. Let $p = \langle s = v_0, v_1, \ldots, t = v_k \rangle$ be a path with labels l_i at nodes v_i. With lemma 4.3 and $\text{pot}_t(l_k, t) = 0$ follows

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

□

Theorem 4.5. *The search can be stopped when the first label at t is removed from the queue.*

Proof. When a label l_t at t is removed from the queue during an s-t query, all remaining labels l at nodes v in the queue fulfill $\text{travelTime}(t) + \text{pot}_t(l_t, t) \leq \text{travelTime}(v) + \text{pot}_t(l, v)$. The same holds for all labels which will be inserted into the queue at a later point in time (lemma 4.3). Assume that travelTime(l_t) is not the shortest route from s to t with time $\mu_{tt}(s,t)$. Then, a shorter route exists which uses at least one unsettled label $l \in L(v)$ at a node v. With lemma 4.4 and $\text{pot}_t(l_t, t) = 0$ follows $\text{travelTime}(t) = \text{travelTime}(t) + \text{pot}_t(l_t, t) \leq \text{travelTime}(v) + \text{pot}_t(l, v) \leq \text{travelTime}(p)$ which contradicts the assumption that p yields a shorter s-t travel time. Therefore, it must be $\text{travelTime}(l_t) = \mu_{tt}(s,t)$ when l_t was removed from the queue and the search can be stopped. □

4.3. Multiple Driving Time Constraints

Dijkstra's algorithm with one driving time constraint (1-DTC) can easily be adapted to handle multiple driving time constraints c_i. With $n = |C|$ driving time constraints, a label l now contains the total travel time $\text{travelTime}(l)$ and n driving time values $\text{breakDist}_1(l), \ldots, \text{breakDist}_n(l)$. Each value $\text{breakDist}_i(l)$ represents the driving time 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 4.1 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 4.6 (Label Dominance). *A label $l \in L(v)$ dominates another label $l' \in L(v)$ if* $\text{travelTime}(l') \geq \text{travelTime}(l)$ *and* $\text{breakDist}_i(l') \geq \text{breakDist}_i(l) \ \forall \ 1 \leq i \leq |C|$.

4.3.1. Potential for Multiple Driving Time Constraints

TODO: Rewrite for only two dtc?

In section 4.2.1 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 4.1 we used the distance $\mu(v, t)$ without regard for pausing from v to t and the driving time $\text{breakDist}(l)$ 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.

$$\text{minBreaks}_i(d) = \begin{cases} \left\lceil \dfrac{d}{c_i^d} \right\rceil - 1 & d > 0 \\ 0 & else \end{cases} \tag{4.8}$$

Consider the example graph in Figure 4.6 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.

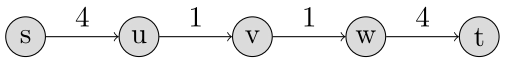

Figure 4.6.: An example graph where, depending on the driving time constraints, a long break at can render a short break obsolete.

This is an optimistic assumption since not in all cases a longer break spares a shorter break. Revisit the example graph of Figure 4.6 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

$$\text{breakEstimate}_i(l, v) = \begin{cases} \text{minBreaks}_i(\text{breakDist}_i(l) + \text{chPot}_t(v)) & 0 < i < n \\ \quad - \sum_{j=i+1}^{n} \text{breakEstimate}_j(l, v) & \\ \text{minBreaks}_n(\text{breakDist}_n(l) + \text{chPot}_t(v)) & i = n \end{cases} \tag{4.9}$$

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

$$\text{breakEstimate}_i(l, v) = \begin{cases} \text{minBreaks}_i(\text{breakDist}_i(l) + \text{chPot}_t(v)) & 0 < i < n \\ \quad - \text{minBreaks}_{i+1}(\text{breakDist}_{i+1}(l) + \text{chPot}_t(v)) & \\ \text{minBreaks}_n(\text{breakDist}_n(l) + \text{chPot}_t(v)) & i = n \end{cases} \tag{4.10}$$

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

$$\text{remBreakTime}(l, v) = \sum_{i=1}^{n} \text{breakEstimate}_i(l, v) \cdot c_i^b \tag{4.11}$$

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

$$\begin{aligned} \text{pot}_t(l, v) &= \text{remBreakTime}(l, v) + \text{chPot}(v, t) \\ &= \text{remBreakTime}(l, v) + \mu(v, t) \end{aligned} \tag{4.12}$$

If queue keys still cannot decrease when propagating labels, lemma 4.4 and theorem 4.5 follow as a consequence. We follow the outline of the proof of lemma 4.3 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 4.7. *Lemma 4.3 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 4.3:

6. $d \geq d' \implies \mathrm{minBreaks}_i(d) \geq \mathrm{minBreaks}_i(d')$ (adaption of 1.)

7. $\mathrm{minBreaks}_i(d + c_i^d) = 1 + \mathrm{minBreaks}_i(d)$ (adaption of 2.)

8. $c_i^d \geq \mathrm{breakDist}_i(l') \geq \mathrm{breakDist}_i(l) + \mathrm{len}(u, v) \geq \mathrm{breakDist}_i(l)$ (adaption of 3.)

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

$$
\begin{aligned}
&\mathrm{travelTime}(l') - \mathrm{travelTime}(l) - \mathrm{pot}_t(l, u) + \mathrm{pot}_t(l', v) \\
&= \mathrm{travelTime}(l) + \mathrm{len}(u, v) - \mathrm{travelTime}(l) \\
&\qquad - (\mathrm{remBreakTime}(l, u) + \mathrm{chPot}(u)) + (\mathrm{remBreakTime}(l', v) + \mathrm{chPot}(v)) \\
&= \mathrm{len}(u, v) - \mathrm{remBreakTime}(l, u) + \mathrm{remBreakTime}(l', v) - \mathrm{chPot}(u) + \mathrm{chPot}(v) \\
&= \mathrm{len}(u, v) + \mathrm{breakEstimate}_2(l', v) \cdot c_2^b + \mathrm{breakEstimate}_1(l', v) \cdot c_1^b \\
&\qquad - \mathrm{remBreakTime}(l, u) - \mathrm{chPot}(u) + \mathrm{chPot}(v) \\
&= \mathrm{len}(u, v) + \mathrm{minBreaks}_2(\mathrm{breakDist}_2(l) + \mathrm{len}(u, v) + \mathrm{chPot}_t(v)) \cdot c_2^b \\
&\qquad + (\mathrm{minBreaks}_1(\mathrm{breakDist}_1(l) + \mathrm{len}(u, v) + \mathrm{chPot}_t(v)) \\
&\qquad - \mathrm{minBreaks}_2(\mathrm{breakDist}_2(l) + \mathrm{len}(u, v) + \mathrm{chPot}_t(v))) \cdot c_1^b \\
&\qquad - \mathrm{remBreakTime}(l, u) - \mathrm{chPot}(u) + \mathrm{chPot}(v) \\
&\overset{(5.\text{ and }6.)}{\geq} \mathrm{len}(u, v) + \mathrm{minBreaks}_2(\mathrm{breakDist}_2(l) + \mathrm{chPot}_t(u)) \cdot c_2^b \\
&\qquad + (\mathrm{minBreaks}_1(\mathrm{breakDist}_1(l) + \mathrm{chPot}_t(u)) \\
&\qquad - \mathrm{minBreaks}_2(\mathrm{breakDist}_2(l) + \mathrm{chPot}_t(u))) \cdot c_1^b \\
&\qquad - \mathrm{remBreakTime}(l, u) - \mathrm{chPot}(u) + \mathrm{chPot}(v) \\
&= \mathrm{len}(u, v) + \mathrm{remBreakTime}(l, u) - \mathrm{remBreakTime}(l, u) - \mathrm{chPot}(u) + \mathrm{chPot}(v) \\
&= \mathrm{len}(u, v) - \mathrm{chPot}(u) + \mathrm{chPot}(v) \\
&\overset{(4.)}{\geq} 0
\end{aligned}
\tag{4.13}
$$

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

$$\begin{aligned}
&\mathrm{travelTime}(l') - \mathrm{travelTime}(l) - \mathrm{pot}_t(l,u) + \mathrm{pot}_t(l',v)\\
&= \mathrm{travelTime}(l) + \mathrm{len}(u,v) + c_1^b - \mathrm{travelTime}(l)\\
&\quad - (\mathrm{remBreakTime}(l,u) + \mathrm{chPot}(u)) + (\mathrm{remBreakTime}(l',v) + \mathrm{chPot}(v))\\
&= \mathrm{len}(u,v) + c_1^b + \mathrm{remBreakTime}(l',v)\\
&\quad - \mathrm{remBreakTime}(l,u) - \mathrm{chPot}(u) + \mathrm{chPot}(v)\\
&= \mathrm{len}(u,v) + c_1^b + \mathrm{breakEstimate}_2(l',v)\cdot c_2^b + \mathrm{breakEstimate}_1(l',v)\cdot c_1^b\\
&\quad - \mathrm{remBreakTime}(l,u) - \mathrm{chPot}(u) + \mathrm{chPot}(v)\\
&= \mathrm{len}(u,v) + c_1^b + \mathrm{minBreaks}_2(\mathrm{breakDist}_2(l) + \mathrm{len}(u,v) + \mathrm{chPot}_t(v))\cdot c_2^b\\
&\quad + (\mathrm{minBreaks}_1(0 + \mathrm{chPot}_t(v))\\
&\quad - \mathrm{minBreaks}_2(\mathrm{breakDist}_2(l) + \mathrm{len}(u,v) + \mathrm{chPot}_t(v)))\cdot c_1^b\\
&\quad - \mathrm{remBreakTime}(l,u) - \mathrm{chPot}(u) + \mathrm{chPot}(v)\\
&\overset{(7.)}{=} \mathrm{len}(u,v) + \mathrm{minBreaks}_2(\mathrm{breakDist}_2(l) + \mathrm{len}(u,v) + \mathrm{chPot}_t(v))\cdot c_2^b\\
&\quad + (\mathrm{minBreaks}_1(r_1^d + \mathrm{chPot}_t(v))\\
&\quad - \mathrm{minBreaks}_2(\mathrm{breakDist}_2(l) + \mathrm{len}(u,v) + \mathrm{chPot}_t(v)))\cdot c_1^b\\
&\quad - \mathrm{remBreakTime}(l,u) - \mathrm{chPot}(u) + \mathrm{chPot}(v)\\
&\overset{(6.\text{ and }8.)}{\geq} \mathrm{len}(u,v) + \mathrm{minBreaks}_2(\mathrm{breakDist}_2(l) + \mathrm{len}(u,v) + \mathrm{chPot}_t(v))\cdot c_2^b\\
&\quad + (\mathrm{minBreaks}_1(\mathrm{breakDist}_1(l) + \mathrm{len}(u,v) + \mathrm{chPot}_t(v))\\
&\quad - \mathrm{minBreaks}_2(\mathrm{breakDist}_2(l) + \mathrm{len}(u,v) + \mathrm{chPot}_t(v)))\cdot c_1^b\\
&\quad - \mathrm{remBreakTime}(l,u) - \mathrm{chPot}(u) + \mathrm{chPot}(v)\\
&\overset{(5.\text{ and }6.)}{\geq} \mathrm{len}(u,v) + \mathrm{minBreaks}_2(\mathrm{breakDist}_2(l) + \mathrm{chPot}_t(u))\cdot c_2^b\\
&\quad + (\mathrm{minBreaks}_1(\mathrm{breakDist}_1(l) + \mathrm{chPot}_t(u))\\
&\quad - \mathrm{minBreaks}_2(\mathrm{breakDist}_2(l) + \mathrm{chPot}_t(u)))\cdot c_1^b\\
&\quad - \mathrm{remBreakTime}(l,u) - \mathrm{chPot}(u) + \mathrm{chPot}(v)\\
&= \mathrm{len}(u,v) + \mathrm{remBreakTime}(l,u) - \mathrm{remBreakTime}(l,u) - \mathrm{chPot}(u) + \mathrm{chPot}(v)\\
&= \mathrm{len}(u,v) - \mathrm{chPot}(u) + \mathrm{chPot}(v)\\
&\overset{(4.)}{\geq} 0
\end{aligned} \tag{4.14}$$

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

$$\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}(u)) + (\text{remBreakTime}(l',v) + \text{chPot}(v))\\
&= \text{len}(u,v) + c_2^b + \text{remBreakTime}(l',v)\\
&\quad - \text{remBreakTime}(l,u) - \text{chPot}(u) + \text{chPot}(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}(u) + \text{chPot}(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}(u) + \text{chPot}(v)\\
&\overset{(7.)}{=} \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}(u) + \text{chPot}(v) \qquad (4.15)\\
&\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}(u) + \text{chPot}(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}(u) + \text{chPot}(v)\\
&= \text{len}(u,v) + \text{remBreakTime}(l,u) - \text{remBreakTime}(l,u) - \text{chPot}(u) + \text{chPot}(v)\\
&= \text{len}(u,v) - \text{chPot}(u) + \text{chPot}(v)\\
&\overset{(4.)}{\geq} 0
\end{aligned}$$

□

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

We now extend the goal-directed approach of section 4.3.1 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 4.3.1.

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 4.8. *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 4.1.4 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 $\text{travelTime}(l) + \text{travelTime}(m)$. □

4.5. 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 $P \subseteq C$. The set of nodes V therefore is split into a set of core nodes C and a set of contracted nodes $V_{CH} = V \setminus C$. The graph $G_{CH} = (V_{CH}, E_{CH}, \text{len})$ with nodes V_{CH} and edges $E_{CH} = \{(u, v) \in E \mid u, v \in V_{CH}\}$ therefore is a valid contraction hierarchy. It contains only contracted nodes and edges between those nodes. Thus, E_{CH} contains only upward edges. The graph $G_C = (V \setminus V_{CH}, E \setminus E_{CH}, \text{len})$ is called the core graph.

The goal-directed core CH query reuses the bidirectional, goal-directed approach of section 4.4 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_{CH} 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_{CH}. 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 4.3.

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 4.3. Since the chosen stopping criterion is valid for this case, the result will be correct. □

4.5.1. Building the Core Contraction Hierarchy

5. Improvements and Implementation

In this chapter, we describe detailed modifications to the algorithm described in section 4.5 which enable further improvements of running time in specific cases and in practice.

Label Queue In section 4.1, 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. TODO WHAT IF PROPAGATING A LABEL IS NEW BEST?

Bidirectional Backward Pruning With a bidirectional A* search, we can use the progress and the potential of the backward search to prune the forward search and vice versa. [TODO cite] A label l at a node v which was propagated along an edge (u, v) can be discarded if we can proof that all paths using the label are longer or equal to the tentative travel time $tent(s, t)$. We know the travel time $\text{travelTime}(l)$ of the label l at v and need 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 with a key of $\text{travelTime}(l) + \text{pot}_s(l, v)$ for label $l \in \overleftarrow{L}(v)$. Labels are removed from the queue with an increasing key. Therefore, we know that if a label TODO continue

Core Contraction Hierarchy Stopping Criteria TODO when one search has no non core nodes left and the other search didn't reach core

Constructing the Core Contraction Hierarchy

6. Evaluation

1. theoretical complexity?

6.1. Experiments

In this section, we evaluate the running time and behavior of our algorithms of chapter 4 in various experiments and describe the underlying data. 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 1.59.0-nightly using the release profile with lto = true and codegen-units = 1.

Data. Our data is a road network of Europe from Open Street Map[1] (OSM). We extract the routing graph from the OSM data using RoutingKit[2], respectively a custom extension[3] of RoutingKit which is capable of extracting parking nodes accordingly and building the core CH as described in section 4.5.1. The obtained routing graph has 81.5 million nodes and 190 million edges. If not stated otherwise, our set P of parking nodes consists of 6800 nodes which were selected due to their OSM attributes. We will later conduct experiments with different choices for P.

Methodology. All experiments are run sequentially. We evaluate the different algorithms of chapter 4 and always include the optimizations of chapter 5. We conduct experiments regarding the preprocessing time of the OSM data 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.

6.1.1. Running Time of Queries

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

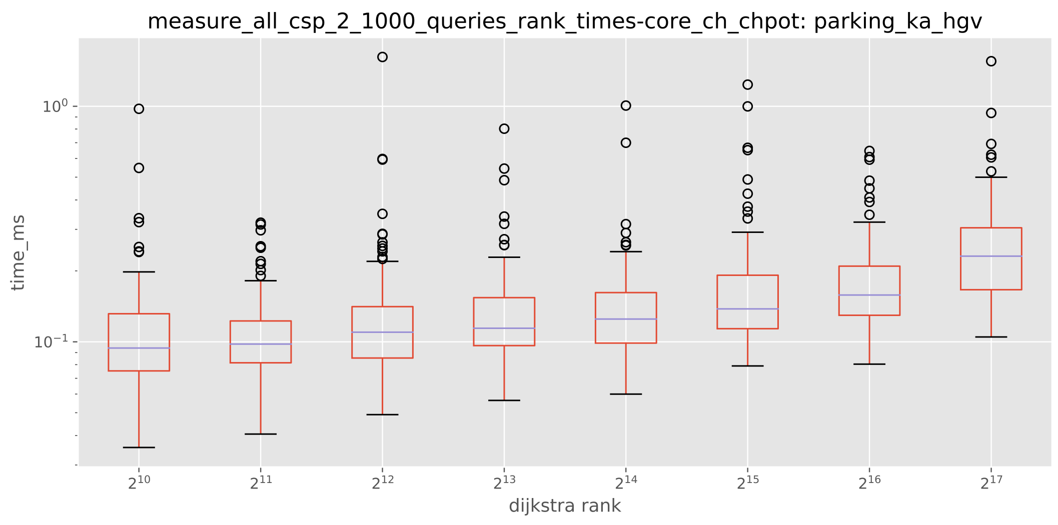

Figure 6.1.: Running times of the goal-directed core CH algorithm for queries to nodes of increasing Dijkstra rank, logarithmic scales.

			Running Time [ms]		Search Space [nodes]	
Goal-Directed	Bidirectional	Core CH	1-DTC	2-DTC	1-DTC	2-DTC
✗	✗	✗				
✓	✗	✗				
✗	✓	✗				
✓	✓	✗				
✗	✓	✓				
✓	✓	✓				

Table 6.1.: Comparison of running times of the base algorithm and its different extensions from section 4.

Backward Pruning	Additional Core CH Stopping Criteria	Running Time [ms]	Search Space [nodes]
✗	✗		
✓	✗		
✗	✓		
✓	✓		

Table 6.2.: Comparison of running times of the goal-directed core CH algorithm with different optimizations from section5.

7. Conclusion

Summary and outlook.

Bibliography

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

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

[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.

[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.

Appendix

A. List of Abbreviations

SPP	Shortest Path Problem
OSM	Open Street Map
1-DTC	Restriction of the long-haul truck driver routing problem to only one driving time constraint
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.
$\mu_{dt}(s,t)$	Driving time of the shortest route between s and t
r	A route consisting of a path and a function breakTime(v) function.
c	A driving time constraint consisting of maximum allowed driving time c^d and break time c^b
C	A set of one or more driving time constraints
$L(v)$	The label set of a node v
$\text{breakDist}_i(l)$	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
$\text{pred}(l)$	Predecessor label of a label l.
$\text{drivingTime}(l)$	Driving time of a label l
$\text{travelTime}(l)$	Travel time of a label l
$\text{pot}(l,v)$	The potential of a label $l \in L(v)$ as defined in section 4.3.1
$\text{len}((u,v))$	The weight, respectively length of an edge (u,v), alternatively used for the length $\text{len}(p)$ of a path p