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, June 13, 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. 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 into the right direction. 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. 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. The backward search operates on a backward graph $\overleftarrow{G}$ with start node t and target node

s. The backward graph is defined as the graph G with inverted edges, i.e, $\overleftarrow{G} = (V, \overleftarrow{E}, \overleftarrow{\text{len}})$ with $\overleftarrow{E} = \{(v, u) \in V \times V : (u, v) \in E\}$ and $\overleftarrow{\text{len}}(u, v) = \text{len}(v, u)$. When the searches meet at a node v in the graph, the information of both searches is combined to yield an s-t path via v and hereby obtain a candidate for the shortest path between s and t. The bidirectional search does only yield an advantage over a unidirectional search if the two searches are stopped earlier than in a unidirectional search. In the latter case, the bidirectional case 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 used potential function. The work of [GH05] introduces a potential and stopping criterion which leads to an improvement over a unidirectional A* search.

TODO absatz konkret machen

2.1. Contraction Hierarchies

2.2. CH Potential

2.3. Core Contraction Hierarchies

3. Problem and Definitions

Truck drivers have to follow laws and regulations regarding the maximum time they are allowed to drive without stopping for a break. This leads to mandatory breaks on longer routes. The regulation also affects the duration of mandatory breaks. We name the extension of the shortest path problem which accounts for driving time limits and mandatory breaks 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 $c_{i,b}$. Thereby, the driving time constraints define a relation $c_i \leq c_{i+1}$ with $c_i \leq c_j \implies c_{i,d} \leq c_{j,d} \wedge c_{i,b} \leq c_{j,b} \forall i, j$. In other words, a greater or equal driving time constraint has a longer or equal driving and break time and there must be no constraint c_i with a longer driving time limit, but shorter break time than another constraint c_j. 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 $\text{breakTime}\colon p \to \{0, c_{1,d}, \ldots, c_{n,d}\}$ at each node i and $n = |C|$. It is $\text{breakTime}(v_i) = 0$ for all nodes $v_i \notin P$. We also define the breakTimeof an entire route r on p as $\text{breakTime}(r) = \sum_{i=0}^{k-1} \text{breakTime}((v_i, v_{i+1}))$. A route must always have a valid path.

Definition 3.1 (Valid Path)**.** *A valid path $p = \langle s = v_0, v_1, ..., t = v_k \rangle$ must comply with the driving time constraints C meaning that it has to comply with all $c \in C$. A path complies with a specific driving time constraint $c' \in C$ if there is no subpath p' between two nodes $u, w \in P' = \{s, t\} \cup \{v_i \in p\colon \text{breakTime}(v_i) \geq c'_b\}$ 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 driving time and travel time between 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 its path is valid and there exists no different route with a valid path between s and t with a smaller travel time.*

The shortest travel time between to 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, driving time $\mu_{dt}(s,t)$ and distance $\mu(s,t)$ as in the SPP 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 route from s to t in G. In other words, find the route r with $\mu_{tt}(s,t) = \text{travelTime}(r)$.

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 one of these 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 runtimes on road networks.

4.1. Dijkstra's Algorithm with One Driving Time Constraint

Dijkstra's algorithm solves the shortest path problem by maintaining a queue Q of nodes with ascending tentative distance from the starting node s, and iteratively settling the node with the smallest distance. When Dijkstra settles a node u, it tests if the distance to the neighbor nodes v with $(u, v) \in E$ can be improved by choosing the current node as a predecessor. We say it *relaxes* all edges $(u, v) \in E$. The search can be stopped if the target node t was removed from the queue [Dij59].

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 represent a possible route, respectively a possible solution for a query from s to the node they belong to. A label in a label set $L(v)$ may represent suboptimal routes to v, i.e., routes which are not a shortest route between s and v. A label set never contains labels which represent routes with an invalid path 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, 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 fig. 4.1.

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

```
1 Procedure settleNextLabel():
2     l ← Q.deleteMin()
3     forall (u, v) ∈ 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 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 $\text{travelTime}(l') = \text{travelTime}(l) + \text{len}(e)$, $\text{breakDist}(l') = \text{breakDist}(l) + \text{len}(e)$, and $\text{pred}(l') = l$.

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 m 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 path to t at all while m reaches the target due to the greater remaining driving time budget. In one case we can proof that a label $l \in L(v)$ cannot yield a better solution than a label $m \in L(v)$. We say m *dominates* l.

Definition 4.1 (Label Dominance for 1-DTC)**.** *A label $l \in L(v)$ dominates another label $l' \in L(v)$ if* $\text{travelTime}(l') \geq \text{travelTime}(l)$ *and* $\text{breakDist}(l') \geq \text{breakDist}(l)$.

If a label $l \in L(v)$ is dominated by another label $m \in L(v)$, then m 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 m. 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 $m \in L(v)$.*

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 $m \in L(v)$ are removed from $L(v)$ if l dominates them. $L(v)$ therefore is the set of known pareto-optimal solutions at v. In fig. 4.2 we define the procedure removeDominated(l) as an operation on a label set.

We now use removeDominated to define the procedure relaxEdge′ as described in fig. 4.3 which propagates a label along an edge and updates the neighbor node's label set if necessary.

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

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

```
1 Procedure relaxEdge'((u,v), l):
2     if breakDist(l) + len(u, v) ≤ c_d then
3         l' ← {(travelTime(l) + len(u, v), breakDist(l) + len(u, v)), l}
4         if l' is not dominated by any label in L(v) then
5             L(v).removeDominated(l')
6             L(v).insert(l')
7             Q.queueInsert(travelTime(l'), l')
```

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

4.1.2. Parking at a Node

The procedure relaxEdge$'$ does not account for parking nodes. 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 label set $L(v)$ and the queue Q as defined in fig. 4.4.

```
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')
```

Figure 4.4.: 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.

Algorithm 4.1: 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 $\mathsf{L}(v)$ of labels for all $v \in V$
Output: Shortest route with travelTime$(j) = \mu_{tt}(s, t)$

```
// Initialization
Q.queueInsert(0,(0, 0, ⊥))
L(s).insert((0, 0, ⊥))
// Main loop
while Q is not empty do
    settleNextLabel()
    if label at t was settled then
        return
```

4.1.4. Correctness

An s-t query with the original 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 the original Dijkstra's algorithm applies and the first label at t which was removed from the queue contains the shortest travel time $\mu_{tt}(s, t)$.

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 travelTime(l) as a queue key, a label $l \in L(v)$ is added to the queue with the key 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 fig. 4.4 where we change the queue keys to use travelTime(l) + travelTime(l, v) instead. The result is shown in fig. 4.5.

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.

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 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
```

```
Procedure RELAXEDGE((u,v), l):
    if breakDist(l) + len(u, v) < c_d then
        D.INSERT((travelTime(l) + len(u, v), breakDist(l) + len(u, v), l))
        if v ∈ P then
            D.INSERT((travelTime(l) + len(u, v) + breakDist_p, 0, l))
        forall l' ∈ D do
            if l' is not dominated by any label in L(v) then
                L(v).REMOVEDOMINATED(l')
                L(v).INSERT(l')
                Q.QUEUEINSERT(travelTime(l') + pot_t(l'), l')
```

Figure 4.5.: Relaxing an edge with regard to the potential.

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

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

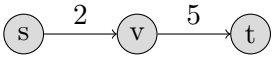

Figure 4.6.: A graph for which the feasibility condition of equation 4.4 does not always hold with the potential pot′.

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

Proof. Let $(u, v) \in E$ be an edge. The procedure RELAXEDGE in fig. 4.5 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. $\text{minBreakTime}(d + c_d) = c_b + \text{minBreakTime}(d)$
3. $c_d \geq \text{breakDist}(l') \geq \text{breakDist}(l) + \text{len}(u, v) \geq \text{breakDist}(l)$
   (line 2 in RELAXEDGE in fig. 4.5)
4. $\text{len}(u, v) - \text{chPot}_t(u) + \text{chPot}_t(v) \geq 0$ (feasibility of the CH potential)
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 $\text{breakDist}(l') = \text{breakDist}(l) + \text{len}(u, v)$.

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

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

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

□

Lemma 4.4. *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 *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 $\text{travelTime}(l_t)$ 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.

$$\text{minBreaks}_i(d) = \begin{cases} \left\lceil \frac{d}{c_{i,d}} \right\rceil - 1 & d > 0 \\ 0 & else \end{cases} \tag{4.8}$$

Consider the example graph in fig. 4.7 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.

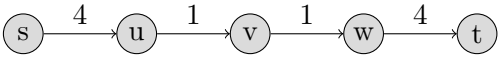

Figure 4.7.: An example graph where, depending on the driving time constraints, a long break at can render a short break obsolete.

This is an optimistic assumption since not in all cases a longer break spares a shorter break. Revisit the example graph of fig. 4.7 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. $d \geq d' \implies \text{minBreaks}_i(d) \geq \text{minBreaks}_i(d')$ (adaption of 1.)
7. $\text{minBreaks}_i(d + c_{i,d}) = 1 + \text{minBreaks}_i(d)$ (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 $\text{breakDist}_i(l') = \text{breakDist}_i(l) + \text{len}(u,v)$.

$$
\begin{aligned}
&\text{travelTime}(l') - \text{travelTime}(l) - \text{pot}_t(l,u) + \text{pot}_t(l',v) \\
&= \text{travelTime}(l) + \text{len}(u,v) - \text{travelTime}(l) \\
&\quad - (\text{remBreakTime}(l,u) + \text{chPot}(u)) + (\text{remBreakTime}(l',v) + \text{chPot}(v)) \\
&= \text{len}(u,v) - \text{remBreakTime}(l,u) + \text{remBreakTime}(l',v) - \text{chPot}(u) + \text{chPot}(v) \\
&= \text{len}(u,v) + \text{breakEstimate}_2(l',v) \cdot c_{2,b} + \text{breakEstimate}_1(l',v) \cdot c_{1,b} \\
&\quad - \text{remBreakTime}(l,u) - \text{chPot}(u) + \text{chPot}(v) \\
&= \text{len}(u,v) + \text{minBreaks}_2(\text{breakDist}_2(l) + \text{len}(u,v) + \text{chPot}_t(v)) \cdot c_{2,b} \\
&\quad + (\text{minBreaks}_1(\text{breakDist}_1(l) + \text{len}(u,v) + \text{chPot}_t(v)) \\
&\quad - \text{minBreaks}_2(\text{breakDist}_2(l) + \text{len}(u,v) + \text{chPot}_t(v))) \cdot c_{1,b} \\
&\quad - \text{remBreakTime}(l,u) - \text{chPot}(u) + \text{chPot}(v) \\
&\overset{\text{(5. and 6.)}}{\geq} \text{len}(u,v) + \text{minBreaks}_2(\text{breakDist}_2(l) + \text{chPot}_t(u)) \cdot c_{2,b} \\
&\quad + (\text{minBreaks}_1(\text{breakDist}_1(l) + \text{chPot}_t(u)) \\
&\quad - \text{minBreaks}_2(\text{breakDist}_2(l) + \text{chPot}_t(u))) \cdot c_{1,b} \\
&\quad - \text{remBreakTime}(l,u) - \text{chPot}(u) + \text{chPot}(v) \\
&= \text{len}(u,v) + \text{remBreakTime}(l,u) - \text{remBreakTime}(l,u) - \text{chPot}(u) + \text{chPot}(v) \\
&= \text{len}(u,v) - \text{chPot}(u) + \text{chPot}(v) \\
&\overset{\text{(4.)}}{\geq} 0
\end{aligned}
\tag{4.13}
$$

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}(u)) + (\text{remBreakTime}(l',v) + \text{chPot}(v)) \\
&= \text{len}(u,v) + c_{1,b} + \text{remBreakTime}(l',v) \\
&\quad - \text{remBreakTime}(l,u) - \text{chPot}(u) + \text{chPot}(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}(u) + \text{chPot}(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}(u) + \text{chPot}(v) \\
&\overset{(7.)}{=} \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}(u) + \text{chPot}(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}(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}
\tag{4.14}
$$

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}(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 and valid path 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 $E_{CH} = \{(u, v) \in E \colon u, v \in V_{CH}\}$ 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. $C' = C \cup \{v \colon (u, v) \in E \wedge u \in C\}$.

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 search which leads to a correct result since the stopping criterion is valid. □

4.6. Goal-Directed Core Contraction Hierarchy Search

5. Improvements and Implementation

In this chapter, we describe detailed modifications to the algorithm described in section 4.6 which enable further improvements of running time in specific cases and in practice.

5.1. Label Queue

5.2. 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 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 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

5.3. Core Contraction Hierarchy Stopping Criteria

5.4. Constructing the Core Contraction Hierarchy

6. Evaluation

1. theoretical complexity?

6.1. Experiments

In this section, we present

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

B. List of Symbols

We use Greek symbols for theoretical and abstract concepts such as the shortest distance between two nodes or an arbitrary 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, otherwise concrete functions use italic bold.

B.1. Theoretical Concepts

π_t	A node potential to t
μ	Shortest distance $\mu(s,t)$ according to the shortest path problem between nodes s and t
μ_{tt}	Travel time $\mu_{tt}(s,t)$ according to the long-haul truck driver routing problem between nodes s and t (including pause time)
$tent$	Tentative travel time $tent(s,t)$ between nodes s and t
μ_{dt}	Driving time $\mu_{dt}(s,t)$ according to the long-haul truck driver routing problem between nodes s and t (excludes pause time)
c	Driving time constraint according to the long-haul truck driver routing problem
pred	Predecessor of a label