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 8, 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 $G = (V, E, len)$. 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 $len : E \to \mathbb{R}_{\geq 0}$ 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 $len(p)$ is defined by the sum of the weights of the edges on the path $len(p) = \sum_{i=0}^{k-1} len((v_i, v_{i+1}))$.

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, $\overleftarrow{G} = (V, \overleftarrow{E}, \overleftarrow{len})$

with $\overleftarrow{E} := \{(v,u) \in V \times V : (u,v) \in E\}$ and $\overleftarrow{len}(u,v) = len(v,u)$. 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 allowed to drive without stopping for a break. This leads to mandatory breaks on longer journeys. The regulation affect 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 $G = (V, E, len)$ 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 R of driving time constraints r_i. Each driving time constraint is defined by a maximum permitted driving time $r_{i,d}$ and a break time $r_{i,b}$. Thereby, the driving time constraints define a relation $r_i \leq r_{i+1}$ with $r_i \leq r_j \implies r_{i,d} \leq r_{j,d} \wedge r_{i,b} \leq r_{j,b} \forall i, j$. 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 r_i with a longer driving time limit, but shorter break time than another constraint r_j. Before exceeding a driving time of $r_{i,d}$, the driver must stop and break for a time of at least $r_{i,b}$. Afterwards, the driver is allowed to drive for a maximum time of $r_{i,d}$ again without stopping. Breaks can only take place at nodes $v \in P$.

A journey j from s to t includes the path of visited nodes $p = \langle s = v_0, v_1, ..., t = v_k, \rangle$ and a break time $breakTime : p \to \{0, r_{i,d}\}$ at each node i. It is $breakTime(v_i) = 0$ $\forall v_i \notin P$. We also define the $breakTime$of an entire journey j on p as $breakTime(j) = \sum_{i=0}^{k-1} breakTime((v_i, v_{i+1}))$. A journey 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 R meaning that it has to comply with all $r_l \in R$.*

Let v_i be the starting node s or any node on the path with $breakTime(v_i) \geq r_{l,b}$ and let v_j be the target node t or any node on the path with $breakTime(v_j) \geq r_{l,b}$ and $i < j$. Then let q be the subpath of p from v_i to v_j. A path complies with driving time constraint r_l if $\mu_{dt}(q) < r_{d,l}$ for all possible subpaths q or there is a node v_m on q with $breakTime(v_m) \geq r_{l,b}$ and $i < m < j$.

We differentiate between driving time and travel time between of a journey.

Definition 3.2 (Driving Time of a Journey)**.** *The driving time $drivingTime(j)$ of a journey j is the length of the path $p = \langle s = v_0, v_1, ..., t = v_k, \rangle$ of the journey.*

Definition 3.3 (Travel Time of a Journey)**.** *The travel time $travelTime(j)$ of a journey is the sum of driving time $drivingTime(j)$ and break time $breakTime(p)$ on its path.*

Definition 3.4 (Shortest Journey)**.** *A journey between two nodes s and t is called a shortest journey if its path is valid and there exists no different journey 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 journey between them is denoted as $\mu_{tt}(s,t)$. Accordingly, $\mu_{dt}(s,t)$ denotes the driving time of the shortest journey 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 $G = (V, E, len)$, a set of parking nodes $P \subseteq V$, a set of driving time constraints R, and start and target nodes $s, t \in V$

Problem: Find the shortest journey from s to t in G. In other words, find the journey j with $\mu_{tt}(s,t) = travelTime(j)$.

In many practical applications, the number of different driving time constraints is limited to only one or two constraints, i.e., $|R| = 1$ or $|R| = 2$. 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. We then describe extensions of the base algorithm to achieve better runtimes on road networks in practice. At first, we will restrict the problem to one driving time constraint for simplicity and drop that constraint later.

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 r 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 journey, 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 journeys to v, i.e., journeys which are not a shortest journey between s and v. A label set never contains label which represent journeys with an invalid path according to r. 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 $travelTime(l') = travelTime(l)+len(e)$, $breakDist(l') = breakDist(l) + len(e)$, and $pred(l') = l$.

Label Pruning.

After propagating a label, we discard the label if it violates the driving time constraint r. That is, if $breakDist(l) > r_d$.

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 journey from s to v than another label m but might have shorter remaining driving time budget $r_d - breakDist(l)$. 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 to 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 $m \in L(v)$ if $travelTime(m) \geq travelTime(l)$ and $breakDist(m) \geq breakDist(l)$.*

If a label $l \in L(v)$ is dominated by another label $m \in L(v)$, then m represents a journey from s to t with a shorter or equal total travel time and longer 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 r 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 $m \in L(v)$.*

We will only add a label l to a label set $L(v)$ if it is pareto-optimal. Labels $m \in L(v)$ are then removed from $L(v)$ if l dominates them. $L(v)$ therefore is the set of pareto-optimal solutions at v. In fig. 4.2 we define the procedure removeDominated(l) as a label set operation.

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 operation which is defined on a label set L, removes all labels from the set which are dominated by the label l.

```
1  Procedure RELAXEDGE'((u,v), l):
2      if breakDist(l) + len(u,v) < r_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 $v \in P$ then 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 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) < r_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) + r_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 $G = (V, E, len)$, set of parking nodes $P \subseteq V$, set of driving time constraints $R = \{r\}$, 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 journey 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
    settleNextNode()
    if minimum of Q is label at t 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 because edge lengths are non negative by definition. Therefore, relaxing any edge of those 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. A* with One Driving Time Constraint

In this section, we transform the base algorithm described in section 4.1 to a goal-directed A* search. We introduce a new potential pot_t which is based on the CH potential 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 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 key $travelTime(l) + pot_t(l, v)$. 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 $travelTime(l) + \pi_t(l, v)$ 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 $chPot_t(v)$ yields a perfect estimate for the distance $\mu(v, t)$ from v to t without regard for driving time constraints and breaks which also 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 from 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 $minBreaks$as a function which calculates the minimum amount of necessary breaks given a driving time d.

Algorithm 4.2: A*+1-DTC

Input: Graph $G = (V, E, len)$, set of parking nodes $P \subseteq V$, a set of driving time constraints $R = \{r\}$, start and target nodes $s, t \in V$, potential $pot_t()$
Data: Priority queue Q, per node set $\mathsf{L}(v)$ of labels for all $v \in V$
Output: Shortest journey 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) < r_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.

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

We now can calculate a lower bound for the necessary break time given a driving time d

$$minBreakTime(d) = minBreaks(d) \cdot r_p \tag{4.2}$$

and finally define our node potential as

$$pot'_t(v) = minBreakTime(d) + chPot_t(v) \tag{4.3}$$
$$= minBreakTime(d) + \mu(v, t) \tag{4.4}$$

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

$$len(u,v) - pot_t(u) + pot_t(v) \geq 0 \quad \forall (u,v) \in E \tag{4.5}$$

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 example 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 $pot_t(s) = 8$ since the potential includes the minimum required break time for a path from s to t. Consequently, with $pot'_t(v) = 5$ and $len(s,v) = 2$, $len(s,v) - pot'_t(s) + pot'_t(v) = -1$.

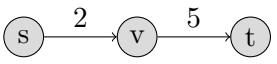

Figure 4.6.: A graph with the potential to break the potential.

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. 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.

$$\begin{aligned} pot_t(l,v) &= minBreakTime(breakDist(l) + chPot(v)) + chPot(v) \\ &= minBreakTime(breakDist(l) + \mu(v,t)) + \mu(v,t) \end{aligned} \tag{4.6}$$

Since the potential pot_t now uses label information, it no longer is a node potential and the feasibility definition as defined in inequality 4.5 can no longer be applied. We still want to use potential and label information to calculate lower bound estimates for the length of paths. We therefore have to show that queue keys of labels, which represent a lower bound estimate for the travel time of the entire journey, 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 $travelTime(l_{i-1}) + pot_t(l_{i-1}, v_{i-1}) \leq travelTime(l_i) + pot_t(l_i, v_i)$.*

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. $d > d' \implies minBreakTime(d) > minBreakTime(d')$
2. $minBreakTime(d + r_d) = r_b + minBreakTime(d)$
3. $r_d > breakDist(l') \geq breakDist(l) + len(u,v) \geq breakDist(l)$
   (line 2 in RELAXEDGE in fig. 4.5)
4. $len(u,v) - chPot_t(u) + chPot_t(v) \geq 0$ (feasibility of the CH potential)
5. $len(u,v) + chPot_t(v) \geq chPot_t(u)$

We show that $travelTime(l_i) + pot_t(l_i, v_i) - travelTime(l_{i-1}) - pot_t(l_{i-1}, v_{i-1}) \geq 0$.

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

$$
\begin{aligned}
&travelTime(l') - travelTime(l) - pot_t(l,u) + pot_t(l',v) \\
&= travelTime(l) + len(u,v) - travelTime(l) \\
&\quad - (minBreakTime(breakDist(l) + chPot(u)) + chPot(u)) \\
&\quad + minBreakTime(breakDist(l') + chPot(v)) + chPot(v) \\
&= len(u,v) + minBreakTime(breakDist(l') + chPot(v)) \\
&\quad - minBreakTime(breakDist(l) + chPot(u)) - chPot(u) + chPot(v) \\
&= len(u,v) + minBreakTime(breakDist(l) + len(u,v) + chPot(v)) \\
&\quad - minBreakTime(breakDist(l) + chPot(u)) - chPot(u) + chPot(v) \\
&\overset{\text{(1. and 5.)}}{\geq} len(u,v) + minBreakTime(breakDist(l) + chPot(u)) \\
&\quad - minBreakTime(breakDist(l) + chPot(u)) - chPot(u) + chPot(v) \\
&= len(u,v) - chPot(u) + chPot(v) \\
&\overset{(4.)}{\geq} 0
\end{aligned}
\tag{4.7}
$$

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

$$
\begin{aligned}
&travelTime(l') - travelTime(l) - pot_t(l,u) + pot_t(l',v) \\
&= travelTime(l) + len(u,v) + r_b - travelTime(l) \\
&\quad - (minBreakTime(breakDist(l) + chPot(u)) + chPot(u)) \\
&\quad + minBreakTime(chPot(v)) + chPot(v) \\
&= len(u,v) + r_b + minBreakTime(chPot(v)) \\
&\quad - minBreakTime(breakDist(l) + chPot(u)) - chPot(u) + chPot(v) \\
&\overset{(2.)}{=} len(u,v) + minBreakTime(r_d + chPot(v)) \\
&\quad - minBreakTime(breakDist(l) + chPot(u)) - chPot(u) + chPot(v) \\
&\overset{\text{(1. and 3.)}}{\geq} len(u,v) + minBreakTime(breakDist(l) + len(u,v) + chPot(v)) \\
&\quad - minBreakTime(breakDist(l) + chPot(u)) - chPot(u) + chPot(v) \\
&\overset{\text{(1. and 4.)}}{\geq} len(u,v) + minBreakTime(breakDist(l) + chPot(u)) \\
&\quad - minBreakTime(breakDist(l) + chPot(u)) - chPot(u) + chPot(v) \\
&= len(u,v) - chPot(u) + chPot(v) \\
&\overset{(4.)}{\geq} 0
\end{aligned}
\tag{4.8}
$$

□

TODO is this even necessary?

Lemma 4.4. *The potential $pot_t(l,v)$ of a label l at a node v is a lower bound for the distance including breaks from v to t.*

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 $travelTime(l_{i-1}) + pot_t(l_{i-1}, v_{i-1}) \geq travelTime(l_i) + pot_t(l_i, v_i)$ for all edges on p, the total length $len(p)$ of the path must follow $pot_t(l_i, v_i) \leq len(p) + pot_t(l_k, t) \Leftrightarrow l(p) \geq pot_t(l_i, v_i) - pot_t(l_k, t)$. Since $pot_t(l_k, t) = 0$, $l(p) \geq pot_t(l_i, v_i)$ holds. □

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

Proof. When a label l at t is removed from the queue during a s-t query, all remaining label m of a node v in the queue fulfill $travelTime(t) + pot_t(l,t) \leq travelTime(v) + pot_t(m,v)$. Assume that $travelTime(t)$ is not the shortest distance from s to t. Then, a shorter path $p = \langle s = v_0, v_1, \ldots, t = v_k \rangle$ exists which uses at least one unsettled label $m \in L(v_i)$. Since l was already removed from the queue, $travelTime(t) = travelTime(t) + pot_t(l,t) \leq travelTime(v) + pot_t(m,v) \leq l(p)$ which contradicts the assumption that p yields a shorter s-t distance than $travelTime(t)$. □

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 r_i. We abbreviate the generalized problem *n-DTC*. For a number of n driving time constraints, a label now contains the total travel time $travelTime$ and n driving time values $breakDist_1, \ldots, breakDist_n$. Each $breakDist_i$ represents the driving time since the last break at a node v with break time $breakTime(v) \geq r_{i,b}$. Pausing at a node occurs with one of the available break times $r_{i,b}$ of a driving time constraint $r_i \in R$. Pausing with an arbitrary break time is possible but yields longer travel times and no advantage. When a path breaks at v for a time $r_{i,b}$, the corresponding label $l \in L(v)$ has $breakDist(l) = 0$ for all $0 < j \leq i$ since the breaks with shorter break times are *included* in the longer break.

Label Propagation

Label propagation simply extends the component-wise addition of the edge weight to all elements of the distance vector. 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 $breakDist_i(l') = breakDist_i(l) + len(e)$ $\forall i \leq |R|$, and $pred(l') = l$.

Label Pruning

The pruning rule for driving time constraints is generalized in a similar way. A label is discarded if $breakDist_i(l) > r_{i,d}$ for any i with $0 < i \leq |R|$.

Label Dominance

At last, 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 $m \in L(v)$ if $breakDist_i(m) \geq breakDist_i(l)$ $\forall i \leq |R|$.*

4.4. Potential for Multiple Driving Time Constraints

TODO: Rewrite for only two dtc? In section 4.2.1 we defined the potential $pot_t(l,v)$ to extend Dijkstra to an A* search with one driving time constraint. We will now generalize pot_t for the use with an arbitrary number n of driving time constraints.

In eq. 4.1 we used the distance $\mu(v,t)$ without regard for pausing from v to t and the driving time $breakDist(l)$ since the last break on the journey 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 r_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 $r_{i,d}$. At the same time, a break of length $r_{i,b}$ will also include breaks of lengths $r_{j,b}$ with $j < i$. We start with simply calculating the amount of necessary breaks $minBreaks_i(d)$ for all restrictions r_i.

$$minBreaks_i(d) = \begin{cases} \left\lceil \frac{d}{r_{i,d}} \right\rceil - 1 & d > 0 \\ 0 & else \end{cases} \tag{4.9}$$

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 journey 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 restriction r_i, we therefore must subtract the amount of longer breaks.

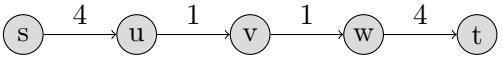

Figure 4.7.: An example graph where, depending on the driving time constraints, a long break at can render a short break obsolete.

This is an optimistic assumption since not in all cases, a longer break renders a short break obsolete. 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 $l \in L(v)$, we now can calculate a lower bound estimate for the amount of necessary breaks for each restriction r_i.

$$breakEstimate_i(l,v) = \begin{cases} minBreaks_i(breakDist_i(l) + chPot_t(v)) & 0 < i < n \\ \quad - \sum_{j=i+1}^{n} breakEstimate_j(l,v) & \\ minBreaks_n(breakDist_n(l) + chPot_t(v)) & i = n \end{cases} \tag{4.10}$$

Since we just subtract all break estimates for driving time constraints greater i to obtain the estimate for i, we can just use

$$breakEstimate_i(l,v) = \begin{cases} minBreaks_i(breakDist_i(l) + chPot_t(v)) & 0 < i < n \\ \quad - minBreaks_{i+1}(breakDist_{i+1}(l) + chPot_t(v)) & \\ minBreaks_n(breakDist_n(l) + chPot_t(v)) & i = n \end{cases} \tag{4.11}$$

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

$$\begin{aligned} pot_t(l,v) &= chPot(v,t) + \sum_{i=1}^{n} breakEstimate_i(l,v) \cdot r_{i,b} \\ &= \mu(v,t) + \sum_{i=1}^{n} breakEstimate_i(l,v) \cdot r_{i,b} \end{aligned} \tag{4.12}$$

We have to proof that queue keys still cannot decrease when propagating labels as as in lemma 4.3. If that is true, lemma 4.4 and theorem 4.5 follow as a consequence.

Lemma 4.7. *Lemma 4.3 still holds for two driving time constraints.*

Proof. We follow the same outline as in the proof of lemma 4.4 and therefore revisit the procedure RELAXEDGE at an edge $(u, v) \in E$ which now can produce three labels at v for each label at u: not parking at v, short break at v, and long break at v.

Following additional general observations can be made:

6. Propagating a label $l \in L(u)$ to obtain a label $l' \in L(v) \implies breakEstimate(l', v) \geq breakEstimate(l, u)$
   (because of non negative edge weights and there can only be more or longer breaks for greater distances)

Case 1: Not parking at v. In this case, $travelTime(l') = travelTime(l) + len(u, v)$ and $breakDist_i(l') = breakDist_i(l) + len(u, v)$.

$$
\begin{aligned}
&travelTime(l') - travelTime(l) - pot_t(l, u) + pot_t(l', v) \\
&= travelTime(l) + len(u, v) - travelTime(l) \\
&\quad - (breakEstimate_1(l, u) \cdot r_{1,b} + breakEstimate_2(l, u) \cdot r_{2,b} + chPot(u)) \\
&\quad + (breakEstimate_1(l', v) \cdot r_{1,b} + breakEstimate_2(l', v) \cdot r_{2,b} + chPot(v)) \\
&= len(u, v) + breakEstimate_1(l', v) \cdot r_{1,b} + breakEstimate_2(l', v) \cdot r_{2,b} \\
&\quad - (breakEstimate_1(l, u) \cdot r_{1,b} + breakEstimate_2(l, u) \cdot r_{2,b}) - chPot(u) + chPot(v) \\
&= len(u, v) + minBreaks_2(breakDist_2(l') + chPot_t(v)) \cdot r_{2,b} \\
&\quad + (minBreaks_1(breakDist_1(l') + chPot_t(v)) \\
&\quad - minBreaks_2(breakDist_2(l') + chPot_t(v))) \cdot r_{1,b} \\
&\quad - (breakEstimate_1(l, u) \cdot r_{1,b} + breakEstimate_2(l, u) \cdot r_{2,b}) - chPot(u) + chPot(v) \\
&= len(u, v) + minBreaks_2(breakDist_2(l) + len(u, v) + chPot_t(v)) \cdot r_{2,b} \\
&\quad + (minBreaks_1(breakDist_1(l) + len(u, v) + chPot_t(v)) \\
&\quad - minBreaks_2(breakDist_2(l) + len(u, v) + chPot_t(v))) \cdot r_{1,b} \\
&\quad - (breakEstimate_1(l, u) \cdot r_{1,b} + breakEstimate_2(l, u) \cdot r_{2,b}) - chPot(u) + chPot(v) \\
&\overset{(5.\text{ and }6.)}{\geq} len(u, v) + minBreaks_2(breakDist_2(l) + chPot_t(u)) \cdot r_{2,b} \\
&\quad + (minBreaks_1(breakDist_1(l) + chPot_t(u)) \\
&\quad - minBreaks_2(breakDist_2(l) + chPot_t(u))) \cdot r_{1,b} \\
&\quad - (breakEstimate_1(l, u) \cdot r_{1,b} + breakEstimate_2(l, u) \cdot r_{2,b}) - chPot(u) + chPot(v) \\
&= len(u, v) + breakEstimate_2(l, u) \cdot r_{2,b} + breakEstimate_1(l, u) \cdot r_{1,b} \\
&\quad - (breakEstimate_1(l, u) \cdot r_{1,b} + breakEstimate_2(l, u) \cdot r_{2,b}) - chPot(u) + chPot(v) \\
&= len(u, v) - chPot(u) + chPot(v) \\
&\overset{(4.)}{\geq} 0
\end{aligned}
\tag{4.13}
$$

Case 2: Short break at v. In this case, $travelTime(l') = travelTime(l) + len(u, v) + r_b$ and $breakDist_1(l') = 0$ and $breakDist_2(l') = breakDist_2(l) + len(u, v)$.

$$
\begin{aligned}
&travelTime(l') - travelTime(l) - pot_t(l,u) + pot_t(l',v)\\
&= travelTime(l) + len(u,v) + r_{1,b} - travelTime(l)\\
&\quad - (breakEstimate_1(l,u)\cdot r_{1,b} + breakEstimate_2(l,u)\cdot r_{2,b} + chPot(u))\\
&\quad + (breakEstimate_1(l',v)\cdot r_{1,b} + breakEstimate_2(l',v)\cdot r_{2,b} + chPot(v))\\
&= len(u,v) + r_{1,b} + breakEstimate_1(l',v)\cdot r_{1,b} + breakEstimate_2(l',v)\cdot r_{2,b}\\
&\quad - (breakEstimate_1(l,u)\cdot r_{1,b} + breakEstimate_2(l,u)\cdot r_{2,b}) - chPot(u) + chPot(v)\\
&= len(u,v) + r_{1,b} + minBreaks_2(breakDist_2(l) + len(u,v) + chPot_t(v))\cdot r_{2,b}\\
&\quad + (minBreaks_1(0 + chPot_t(v))\\
&\quad - minBreaks_2(breakDist_2(l) + len(u,v) + chPot_t(v)))\cdot r_{1,b}\\
&\quad - (breakEstimate_1(l,u)\cdot r_{1,b} + breakEstimate_2(l,u)\cdot r_{2,b}) - chPot(u) + chPot(v)\\
&\overset{(2.)}{=} len(u,v) + minBreaks_2(breakDist_2(l) + len(u,v) + chPot_t(v))\cdot r_{2,b}\\
&\quad + (minBreaks_1(r_{1,d} + chPot_t(v))\\
&\quad - minBreaks_2(breakDist_2(l) + len(u,v) + chPot_t(v)))\cdot r_{1,b}\\
&\quad - (breakEstimate_1(l,u)\cdot r_{1,b} + breakEstimate_2(l,u)\cdot r_{2,b}) - chPot(u) + chPot(v)\\
&\overset{(1.\text{ and }3.)}{\geq} len(u,v) + minBreaks_2(breakDist_2(l) + len(u,v) + chPot_t(v))\cdot r_{2,b}\\
&\quad + (minBreaks_1(breakDist_1(l) + len(u,v) + chPot_t(v))\\
&\quad - minBreaks_2(breakDist_2(l) + len(u,v) + chPot_t(v)))\cdot r_{1,b}\\
&\quad - (breakEstimate_1(l,u)\cdot r_{1,b} + breakEstimate_2(l,u)\cdot r_{2,b}) - chPot(u) + chPot(v)\\
&\overset{(5.\text{ and }4.)}{\geq} len(u,v) + minBreaks_2(breakDist_2(l) + chPot_t(u))\cdot r_{2,b}\\
&\quad + (minBreaks_1(breakDist_1(l) + chPot_t(u))\\
&\quad - minBreaks_2(breakDist_2(l) + chPot_t(u)))\cdot r_{1,b}\\
&\quad - (breakEstimate_1(l,u)\cdot r_{1,b} + breakEstimate_2(l,u)\cdot r_{2,b}) - chPot(u) + chPot(v)\\
&= len(u,v) + breakEstimate_2(l,u)\cdot r_{2,b} + breakEstimate_1(l,u)\cdot r_{1,b}\\
&\quad - (breakEstimate_1(l,u)\cdot r_{1,b} + breakEstimate_2(l,u)\cdot r_{2,b}) - chPot(u) + chPot(v)\\
&= len(u,v) - chPot(u) + chPot(v)\\
&\overset{(4.)}{\geq} 0
\end{aligned}
\tag{4.14}
$$

Case 3: Long break at v. In this case, $travelTime(l') = travelTime(l) + len(u,v) + r_b$ and $breakDist_i(l') = 0$.

$$
\begin{aligned}
&travelTime(l') - travelTime(l) - pot_t(l,u) + pot_t(l',v) \\
&= travelTime(l) + len(u,v) + r_{2,b} - travelTime(l) \\
&\quad - (breakEstimate_1(l,u) \cdot r_{2,b} + breakEstimate_2(l,u) \cdot r_{2,b} + chPot(u)) \\
&\quad + (breakEstimate_1(l',v) \cdot r_{2,b} + breakEstimate_2(l',v) \cdot r_{2,b} + chPot(v)) \\
&= len(u,v) + r_{2,b} + breakEstimate_1(l',v) \cdot r_{1,b} + breakEstimate_2(l',v) \cdot r_{2,b} \\
&\quad - (breakEstimate_1(l,u) \cdot r_{1,b} + breakEstimate_2(l,u) \cdot r_{2,b}) - chPot(u) + chPot(v) \\
&\overset{\text{(2.) and TODO is label}}{=} len(u,v) + minBreaks_2(r_{2,d} + chPot_t(v)) \cdot r_{2,b} \\
&\quad + (minBreaks_1(r_{2,d} + chPot_t(v)) - minBreaks_2(r_{2,d} + chPot_t(v))) \cdot r_{1,b} \\
&\quad - (breakEstimate_1(l,u) \cdot r_{1,b} + breakEstimate_2(l,u) \cdot r_{2,b}) - chPot(u) + chPot(v) \\
&\overset{\text{(1. and 3. and } r_{2,d} >= r_{1,d})}{\geq} len(u,v) + minBreaks_2(breakDist_2(l) + len(u,v) + chPot_t(v)) \cdot r_{2,b} \\
&\quad + (minBreaks_1(breakDist_1(l) + len(u,v) + chPot_t(v)) \\
&\quad - minBreaks_2(breakDist_2(l) + len(u,v) + chPot_t(v))) \cdot r_{1,b} \\
&\quad - (breakEstimate_1(l,u) \cdot r_{1,b} + breakEstimate_2(l,u) \cdot r_{2,b}) - chPot(u) + chPot(v) \\
&\overset{\text{(5. and 4.)}}{\geq} len(u,v) + minBreaks_2(breakDist_2(l) + chPot_t(u)) \cdot r_{2,b} \\
&\quad + (minBreaks_1(breakDist_1(l) + chPot_t(u)) \\
&\quad - minBreaks_2(breakDist_2(l) + chPot_t(u))) \cdot r_{1,b} \\
&\quad - (breakEstimate_1(l,u) \cdot r_{1,b} + breakEstimate_2(l,u) \cdot r_{2,b}) - chPot(u) + chPot(v) \\
&= len(u,v) + breakEstimate_2(l,u) \cdot r_{2,b} + breakEstimate_1(l,u) \cdot r_{1,b} \\
&\quad - (breakEstimate_1(l,u) \cdot r_{1,b} + breakEstimate_2(l,u) \cdot r_{2,b}) - chPot(u) + chPot(v) \\
&= len(u,v) - chPot(u) + chPot(v) \\
&\overset{(4.)}{\geq} 0
\end{aligned}
\tag{4.15}
$$

□

4.5. Bidirectional Search with Multiple Driving Time Constraints

A common approach to improve constant factors of the runtime of shortest path queries is to start two searches which search for a path in forward direction from s and in backward direction from t. The searches ideally meet in the middle, thus halving the theoretical search space of a Dijkstra search since the search radius of each search now is only halve the distance to the target node. The distances of the forward and backward search are combined at nodes which were settled by both searches. Since we aim to stop the search as early as possible, we have to decide on a new 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 stopping criterion especially becomes interesting if we use the potential of section 4.4 for a bidirectional A* search.

4.5.1. Bidirectional Dijkstra with Multiple Driving Time Constraints

The input of the search remains a graph $G = (V, E, len)$, a set of parking nodes $P \subseteq V$, a set of driving time constraints R, and start and target nodes $s, t \in V$. We now define

the backwards graph $\overleftarrow{G}$ as the graph $(V, \overleftarrow{E}, \overleftarrow{len})$ with $\overleftarrow{E} := \{(v,u) \in V \times V : (u,v) \in E\}$ and $\overleftarrow{len}(u,v) = len(v,u)$. The forward search then is a normal A* search on 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 forward, respectively backward label sets $\overrightarrow{L}(v)$ and $\overleftarrow{L}(v)$ 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 distance for $\mu_{tt}(s,t)$ in a variable $dist(s,t)$ which we initialize with ∞ before settling the first node.

When a search settles a node v, it additionally checks 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 label combination of labels $l \in \overrightarrow{L}(v)$ and $m \in \overleftarrow{L}(v)$ that yields the shortest travel time $\mu_{tt}(s,t) = \mu_{tt}(s,v) + \mu_{tt}(v,t)$ without violating any constraint r_i. This is simply done by choosing the labels l and m with minimal $travelTime(l) + travelTime(m)$ while $breakDist_1(l) + breakDist_1(m) < r_{1,d}$ and $breakDist_1(l) + breakDist_1(m) < r_{2,d}$. If the resulting distance for an s-t path via v is smaller than the previously known minimum tentative distance $dist(s,t)$, we update $dist(s,t)$ accordingly. We stop the forward search if the minimum key of $\overrightarrow{Q}$ is greater than $dist(s,t)$ and stop the backward search when the minimum key of $\overleftarrow{Q}$ is greater than $dist(s,t)$.

Theorem 4.8. *At the point in time when forward and backward search have stopped, $dist(s,t) = \mu_{tt}(s,t)$. In other words, when the search stops then $dist(s,t)$ equals the minimum travel time from s to t which complies with the driving time constraints R.*

TODO SPLIT THIS UP INTO PARTS SINCE NO HUMAN CAN UNDERSTAND THIS BLOCK

Proof. We first show that when the search stops, all s-t paths which comply with the driving time constraints R and which were not found yield a larger travel time $\mu_{tt}(s,t)$ than the current $dist(s,t)$. Then, we show that we don't miss any paths and all paths which comply with the driving time constraints are constructed. That also means that if $dist(s,t) = \infty$ when the search stops, there exist no paths from s to t.

Since the search stops when the minimum key of $\overrightarrow{Q}$ and $\overleftarrow{Q}$ are both greater than $dist(s,t)$, all labels l which will be settled by continuing the search have a greater distance $travelTime(l)$. 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 $dist(s,t)$.

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 $travelTime(l) + travelTime(m) = \mu_{tt}(s,t)$. Each label is the result of a unidirectional search from s, respectively from t. In section 4.1.4 we proofed that a unidirectional search can be stopped when the first label at its target node was removed from the queue. Since l and m both are smaller or equal to $dist(s,t)$ and both queue keys of forward and backward queue are greater, both l and m where removed from the respective queue. Therefore, we know that $\overrightarrow{L}$ and $\overleftarrow{L}$ contain the labels with the shortest distance from s to v, respectively from t to v. Connecting the labels which correspond to the shortest subpaths s-v and t-v does not necessarily comply with the driving time constraints R. We therefore have to proof that in the case in which the shortest s-t path is not the combination of the two shortest subpaths, the correct solution is found nevertheless before the search stops. TODO □

4.5.2. Bidirectional A*

We now extend the bidirectional search to a bidirectional A* search. Thus, we use a forward potential $\overrightarrow{pot}_t(l, v)$ and a backwards potential $\overleftarrow{pot}_t(l, v)$. Connecting two subpaths of forward and backward search to an s-t path remains unchanged since we only use the potentials for the queue keys of $\overrightarrow{Q}$ and $\overleftarrow{Q}$. We use components $breakDist_1$ and $breakDist_1$ of the two labels to test if the connected path complies with the driving time constraints R and sum the both $travelTime$ components of the two labels to obtain the travel time $\mu_{tt}(s, t)$ of the full path.

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 $dist(s, t)$. 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 $travelTime(l) + pot_s(l, v)$ 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

4.6. Core Contraction Hierarchy with Two Driving Time Constraints

Given a graph $G = (V, E, len)$, 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 $G_{CH} = (V_{CH}, E_{CH}, len)$ with nodes V_{CH} and edges $E_{CH} = \{(u, v) \in E : u, v \in V_{CH}\}$ 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 $G_C = (V \setminus V_{CH}, E \setminus E_{CH}, len)$ is called the core graph.

The core CH query essentially is a bidirectional Dijkstra with a modified stopping criterion. It operates on a modified forward graph $\overrightarrow{G}^*$ and a modified backward graph $\overleftarrow{G}^*$. The forward graph $\overrightarrow{G}^* = (V, \overrightarrow{E}^*, len)$ 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 $\overleftarrow{G}^* = (V, \overleftarrow{E}^*, \overleftarrow{len})$ with $\overleftarrow{E}^* = \overleftarrow{E}_{CH} \cup E_C$ and $\overleftarrow{len}(u, v) = len(v, u)$.

The bidirectional query in G^* 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 Dijkstra search in G_C. A search may begin at a core node and only perform the Dijkstra search. 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 more conservative than the stopping criterion for a bidirectional Dijkstra search, we can use it for our combination of CH and bidirectional Dijkstra.

To proof the correctness of the search, we will differentiate between the cases where neither of the two searches reaches the core and where at least one of the searches does reach the core. TODO

Algorithm 4.3: CORE-CH WITH DRIVING TIME CONSTRAINTS

Input: Graph $G = (V, E, len)$, parking nodes $P \subseteq V$, driving time constraint r, potential $\mathsf{pot}()$, source node $s \in V$
Data: Priority queue Q, per node priority queue $\mathsf{L}(v)$ of labels for all $v \in V$
Output: Distances for all $v \in V$, tree of allowed shortest paths according to the restriction r from s, given by l_{pred}

```
// Initialization
1  Q.INSERT(s, (0,0))
2  L(s).INSERT((⊥,⊥), pot(0,0))
3  fwNext ← true
// Main loop
4  while stopping criterion is not met do
5      if fwNext then
6      u ← Q.DELETEMIN()
7      (travelTime, breakDist_1) ← L(u).MINKEY()
8      l ← L(u).DELETEMIN()
9      if L(u) is not empty then
10         k_dist ← L(u).MINKEY()
11         Q.INSERT(u, k_dist)
12     forall (u,v) ∈ E do
13         if travelTime + len(u,v) < r_d then
14             D ← {(travelTime + len(u,v), breakDist_1 + len(u,v))}
15             if v ∈ P then
16                 D.INSERT((travelTime + len(u,v) + r_p, 0))
17             forall x ∈ D do
18                 if x is not dominated by any label in L(v) then
19                     L(v).REMOVEDOMINATED(x)
20                     L(v).INSERT((l, (u,v)), x)
21                     if Q.CONTAINS(v) then
22                         Q.DECREASEKEY(v, x)
23                     else
24                         Q.INSERT(v, x)
```

4.6.1. Building the Contraction Hierarchy

4.7. Combining A* and Core Contraction Hierarchy

5. Evaluation

1. theoretical complexity?
2. experiments, setup, methodology, results

6. 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)
$dist$	Tentative travel time $dist(s,t)$ 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)
r	Driving time constraint according to the long-haul truck driver routing problem
$pred$	Predecessor of a label