Karlsruhe Institute of Technology

Beschleunigtes LKW-Routing mit zeitabhängigen Fahrverboten

Bachelorarbeit
von

Julius Häcker

An der Fakultät für Informatik
Institut für Theoretische Informatik

Erstgutachter:	Dr. Torsten Ueckerdt
Zweitgutachter:	Prof. Dr. Peter Sanders
Betreuende Mitarbeiter:	Dr. Alexander Kleff
	Dr. Frank Schulz
	Tim Zeitz, M. Sc.

Bearbeitungszeit: 16. Dezember 2020 – 16. April 2021

Eidesstattliche Erklärung

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, 16. April 2021

Zusammenfassung

Routinganwendungen sind für uns heutzutage sehr wichtig und werden häufig genutzt. Für LKWs brauchen wir anderes Routing als für PKWs, da es für LKWs temporäre Fahrverbote gibt. Das können Nacht- oder Wochentagsfahrverbote sein, die ein großräumiges Gebiet betreffen, oder Fahrverbote für bestimmte Straßenabschnitte. Dazu gibt es für LKWs im Gegensatz zu PKWs gesetzlich geregelte Lenk- und Ruhezeiten. Das bedeutet, dass der Fahrer nach einer bestimmten Fahrzeit eine Pause machen muss. Daher können Pausen und das Anfahren von Parkplätzen nötig werden. Für sinnvolle LKW-Routen ist es also wichtig, dass temporäre Fahrverbote und Lenk- und Ruhezeiten bereits in der Routenplanung berücksichtigt werden. Die Qualität einer Pause hängt von der Qualität des Parkplatzes ab, auf dem die Pause gemacht wird. Daher kann es sinnvoll sein, einen größeren Umweg für einen Parkplatz besserer Qualität zu fahren. Dadurch entsteht ein Trade-Off für die Berechnung der Routen. In [KSWZ20] wurde ein Algorithmus vorgestellt, der diesen Trade-Off beim Routing berücksichtigt. Dabei werden keine Lenk- und Ruhezeiten berücksichtigt. Wir erweitern diesen Algorithmus zu einer bidirektionalen Suche, um stärkeres Pruning zu ermöglichen und dadurch die Laufzeit zu verbessern. Die Laufzeit konnte dabei bei einigen langsamen (>10s) Queries halbiert werden. Ansonsten konnten keine langsamen Queries wesentlich beschleunigt werden. Außerdem modifizieren wir den Algorithmus, um Lenk- und Ruhezeiten in einer vereinfachten Form zu berücksichtigen. Dabei vereinfachen sich die Ergebnisse einiger Queries, sodass es dort weniger berechnete Routen gibt.

Abstract

Routing applications are nowadays very important and often used. For trucks we need a different routing than for cars because there are temporary road blocks for trucks. These might be night driving bans where a large area is blocked or driving bans where a specific road section is blocked. In addition there are government regulations for truck drivers that limit non-stop driving times. This means that the driver has to take a break after a certain accumulated driving time. Therefore it is important for truck driver routes to take temporary road blocks and limits on non-stop driving into account already at the route planning. The quality of a break depends on the quality of the parking lot where the break is taken. Therefore it might be good to take a larger detour for a parking lot of higher quality. This leads to a trade off for the calculation of truck driver routes. In [KSWZ20] an algorithm is introduced which takes this trade off into account for routing. Limits on non-stop driving are ignored in this algorithm. We extend this algorithm to a bidirectional search to enable stronger pruning and reduce the runtime. We were able to reduce the runtime of some slow queries (>10s) by half. Apart from these queries no slow query got much faster. In addition to that we will modify this algorithm to take limits on non-stop driving times in a simplified variant into account. Thereby the results of some queries get simplified so that the number of calculated routes gets smaller.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Literaturübersicht . . . 2
- 1.2 Übersicht . . . 3

2 Grundlagen **5**
- 2.1 Graph mit Straßensperrungen und Parkplätzen . . . 5
- 2.2 Routen . . . 6
- 2.3 Pareto-Optimalität . . . 6
- 2.4 Basisproblem . . . 7
- 2.5 Modifiziertes Basisproblem . . . 7

3 Basisalgorithmus **9**
- 3.1 Beschreibung . . . 9
- 3.2 Korrektheit und Laufzeit . . . 10
- 3.3 Beschleunigungstechniken . . . 10
  - 3.3.1 Pruning . . . 11
  - 3.3.2 A*-Suche . . . 11
  - 3.3.3 Distanz-Berechnung . . . 12

4 Beschleunigter Basisalgorithmus **13**
- 4.1 Rückwärtssuche . . . 13
- 4.2 Beschreibung . . . 16

5 Modifizierter Basisalgorithmus **19**
- 5.1 Beschreibung . . . 19
- 5.2 Korrektheit . . . 22
- 5.3 Laufzeit . . . 22
- 5.4 Problem-Komplexität . . . 24

6 Versuchsaufbau **27**
- 6.1 Hardware und Kompiler . . . 27
- 6.2 Straßennetzwerk . . . 27
- 6.3 Queries . . . 28

7 Evaluation des beschleunigten Basisalgorithmus **31**
- 7.1 Planungshorizont und Strategien . . . 31
- 7.2 Laufzeit Basisalgorithmus . . . 33
- 7.3 Vollständige Rückwärtssuche . . . 35
- 7.4 Target Pruning . . . 35
- 7.5 Laufzeiten der Strategien . . . 39
  - 7.5.1 Keine Limitierung . . . 40
  - 7.5.2 Limitierung . . . 41
  - 7.5.3 Verzögerter Start . . . 44

7.6 Parallelisierung 50
7.7 Beispiel-Query 53

8 Evaluation des modifizierten Basisalgorithmus 57
8.1 Planungshorizont, Testmengen und Parameter 57
8.2 Ergebnisveränderung 57
8.3 Laufzeit 62
8.4 Modell 63

9 Fazit 65

Literaturverzeichnis 67

1. Einleitung

In Europa gibt es für LKWs an vielen Orten temporäre Fahrverbote. Das können zum Beispiel Nachtfahrverbote sein, bei denen ein großes Gebiet oder ein ganzes Land für eine Zeit gesperrt ist. Es gibt auch Fahrverbote, bei denen aufgrund eines erhöhten Verkehrsaufkommens ein kleineres Gebiet oder auch nur ein einzelner Straßenabschnitt temporär gesperrt ist. Das ist zum Beispiel in Innenstädten oder an Grenzübergängen der Fall. Würde man für LKWs dasselbe Routing wie für PKWs verwenden, dann kann es vorkommen, dass das Fahren der berechneten Route nicht legal ist oder der Fahrer unterwegs eine außerplanmäßige Pause machen muss. Daher müssen temporäre Fahrverbote für LKWs bereits bei der Routenplanung berücksichtigt werden. Dann können diese Fahrverbote zum Beispiel umfahren werden. Wenn das gesperrte Gebiet wie bei Nachtfahrverboten aber sehr groß ist, ist es sinnvoller, das Fahrverbot abzuwarten und die Fahrt nach Ende des Fahrverbots fortzusetzen. Für das Warten muss dann ein Parkplatz angefahren werden. Deshalb müssen diese Fahrverbote bereits bei der Routenplanung berücksichtigt werden, um möglicherweise bessere Alternativrouten zum Umfahren oder Abwarten der Fahrverbote zu berechnen.
Durch das Berücksichtigen von temporären Fahrverboten beim Routing lässt sich Warten oder das Fahren von Umwegen nicht vermeiden. Das Warten kann dabei je nach Parkplatz eine unterschiedliche Qualität für den Fahrer haben: Auf einem Parkplatz, der z.B. nicht über eine öffentliche Toilette verfügt, ist das Warten weniger komfortabel als auf einem Parkplatz, der mit Toiletten, Restaurants etc. ausgestattet ist. Für das Anfahren eines besseren Parkplatzes muss dabei aber unter Umständen ein größerer Umweg gefahren werden, wodurch eine spätere Ankunft und höhere Fahrtkosten entstehen. Für das Routing müssen also mehrere Kriterien betrachtet werden: Ankunftszeit und Reisekosten (Kosten für Fahren und Warten). Eine Route mit späterer Ankunft kann dabei sinnvoll sein, falls sie geringere Reisekosten durch Warten auf einem Parkplatz besserer Qualität aufweist. Dadurch entsteht eine Menge von Routen, die bezüglich der Kriterien Ankunftszeit und Reisekosten optimal sind. Für Routen in dieser Menge gilt, dass bei späterer Ankunft die Reisekosten immer geringer werden.
In [KSWZ20] wurde ein Algorithmus vorgestellt, der eine solche Menge von optimalen Routen berechnet. Diesen Algorithmus bezeichnen wir in dieser Arbeit als *RouBIRT*-Algorithmus (*Routing with Ban Intervals Ratings and Traveltime costs*). Für Anfragen, bei denen Start und Ziel weit auseinanderliegen und die schnellste Route unter Berücksichtigung von temporären Fahrverboten nicht gültig ist, ist die Laufzeit dieses Algorithmus teilweise bei 30s oder mehr. Für diese Anfragen ist die Laufzeit des Algorithmus also nicht praktibel. Um diese langsamen Anfragen zu beschleunigen, werden wir als ersten Schwerpunkt in

dieser Arbeit den Algorithmus zu einer bidirektionalen Suche erweitern. Dabei berechnen wir in der Rückwärtssuche lediglich minimale Reisezeiten zum Ziel. Mit diesen minimalen Reisezeiten können wir dann vorläufige Kostenwerte am Zielknoten berechnen und ein stärkeres Pruning ermöglichen, um den Suchraum besser einzuschränken.
In vielen Ländern gibt es für LKW-Fahrer Regelungen zu Lenk- und Ruhezeiten. Dabei muss ein Fahrer nach einer bestimmten Fahrzeit eine Pause machen. Das führt zur Erholung des Fahrers und zu einer gesteigerten Konzentration beim Fahren und somit auch zu mehr Sicherheit im Straßenverkehr. Dabei gibt es zwei Arten von Pausen: kurze Pausen und lange Pausen. Kurze Pausen müssen nach einer bestimmten Fahrzeit am Stück gemacht werden, lange Pausen nach einer bestimmten Fahrzeit seit der letzten langen Pause. In der EU sind diese nach Verordnung (EG) Nr.561/2006 geregelt. Lange Pausen sind dabei mindestens 11h lang und für eine Pause muss das Fahrzeug auf einem Parkplatz geparkt werden. Es kann zudem sinnvoll sein, bereits vor dem Erreichen der maximalen Fahrzeit eine Pause zu machen, um gleichzeitig ein Fahrverbot abzuwarten. Daher muss das Anfahren von Parkplätzen bereits während der Routenplanung berücksichtigt werden. Im Rahmen dieser Arbeit unterscheiden wir nicht zwischen zwei Arten von Pausen, sondern beschränken uns auf den Fall, dass der Fahrer nach einer bestimmten Fahrzeit am Stück eine Pause bzw. Fahrtunterbrechung einer bestimmten Länge machen muss. Als zweiten Schwerpunkt untersuchen wir in dieser Arbeit, wie wir diese Fahrtunterbrechungen beim Routing berücksichtigen können.

1.1 Literaturübersicht

Diese Arbeit basiert auf [KSWZ20]. Dort wird ein Algorithmus vorgestellt, der für zeitunabhängige Fahrzeiten und verschiedene Wartekosten eine Menge von optimalen Routen berechnet. Dabei werden die Kriterien Ankunftszeit und Reisekosten für die Optimalität in Betracht gezogen. Dabei wird Pruning mithilfe des minimalen Fahrzeit zum Ziel verwendet, wenn temporäre Fahrverbote ignoriert werden.
In [KBS+17] wird ein Algorithmus zur Berechnung von Routen mit zeitabhängigen Fahrtdauern vorgestellt. Dabei werden Fahrtunterbrechungen berücksichtigt. Die Problemstellung ist insofern anders, als dass hier nicht zwischen Parkplätzen verschiedener Qualität unterschieden wird und nur nach einer Route mit frühest möglicher Ankunft gesucht wird. Für die Berechnung dieser Route werden Reisezeitprofile berechnet, die dann sowohl vorwärts vom Start aus als auch Rückwärts vom Ziel aus durch den Graph propagiert werden. Zudem ist das Pausemachen nur auf Parkplätzen erlaubt. Für die Beschleunigung des Algorithmus wird hier zudem eine heuristischer Ansatz vorgestellt.
In [vdTdWB18] wird ein Algorithmus vorgestellt, um Routen für LKW-Fahrer unter der Berücksichtigung von Fahrunterbrechungen und temporären Fahrverboten zu berechnen. Die Fahrtdauern sind hier zeitunabhängig. Zudem ist lediglich eine Pause auf der Route möglich. Der dem Straßennetz zugrundeliegende Graph wird dabei in zwei Teilgraphen aufgeteilt. Der Übergang vom ersten in den zweiten Teilgraph bedeutet dabei das Planen einer Pause. Dort wird nicht berücksichtigt, wo die Pause gemacht wird. Dadurch ist die Pause auch auf Kanten und auf Nicht-Parkplätzen möglich.
In [Bom20] wird ein weiterer Algorithmus zur Berechnung von Routen unter der Berücksichtigung von Fahrtunterbrechungen vorgestellt. Dabei werden im Gegensatz zu den beiden vorherigen Veröffentlichungen zwei Arten von Pausen betrachtet: kurze und lange Pausen. Ein weiterer Unterschied ist die Betrachtung von mehreren Kunden. Der Fahrer muss dabei verschiedene Kunden in einer vorgegebenen Reihenfolge anfahren. Die Kunden haben dabei jeweils ein oder mehrere Zeitfenster, in dem der Fahrer dort sein kann, um den LKW zu entladen bzw. beladen. Es wird eine exakte Lösung und eine Lösung mittels einer Heuristik verwendet. Für die exakte Lösung wird dabei ein *Mixed Integer Linear Program* (*MILP*) formuliert.

1.2 Übersicht

In dieser Arbeit erklären wir zunächst einige Grundlagen zum verwendeten Modell und den Algorithmus aus [KSWZ20]. Anschließend stellen wir die bidirektionale Suche für die Beschleunigung dieses Algorithmus und die Modifikation für die Berücksichtigung von Lenk- und Ruhezeiten vor. Danach werden wir die beiden daraus entstehenden Algorithmen evaluieren.

2. Grundlagen

In diesem Kapitel führen wir grundlegende Begriffe und Notation für die präsentierten Algorithmen ein.

2.1 Graph mit Straßensperrungen und Parkplätzen

Ein Graph mit Straßensperrungen und Parkplätzen wird durch die folgenden Elemente charakterisiert:

- Eine Knotenmenge V mit $|V| = n$ und eine Menge von gerichteten Kanten E mit $|E| = m$
- Eine Abbildung $\delta : E \to \mathbb{N}_0$. Dabei gibt $\delta(e)$ für eine Kante $e \in E$ die Zeit an, die benötigt wird, um vollständig über die Kante zu fahren. Dabei werden keine Sperrungen berücksichtigt, δ ist also zeitunabhängig.
- Eine Menge von Parkplätzen $P \subseteq V$. Die Menge der Knoten wird damit unterteilt in eine Menge von Parkplätzen und eine Menge von Nicht-Parkplätzen. Im Rahmen dieser Arbeit ist Warten sowohl auf Parkplätzen als auch auf Nicht-Parkplätzen möglich. Das Warten auf Nicht-Parkplätzen wir dabei stärker bestraft als das Warten auf Parkplätzen. Es ist auch möglich auf Kanten zu warten. Dabei unterscheiden wir nicht zwischen dem Warten auf Nicht-Parkplätzen und dem Warten auf Kanten. Es wird jeweils gleich bestraft.
- Eine Menge von Ratings $R = \{0, .., r\}$ und eine Abbildung $\rho : V \to R$, die jedem Knoten ein Rating zuweist. Dabei ist $\rho(v) = 0$ für alle Knoten $v \notin P$, die keine Parkplätze sind. Das Rating eines Parkplatzes $v \in P$ wird durch die Anzahl der dort verfügbaren Stellplätze bestimmt.
- Eine Menge von Wartekosten $W = \{w_0, ..., w_r\}$. Dabei kostet es pro Zeiteinheit $w_{\rho(v)}$, auf einem Knoten $v \in V$ zu warten. Das Warten auf Kanten kostet w_0. O.B.d.A gilt $w_0 > w_1 > ... > w_r$. Das Fahren entlang einer Kante kostet dabei pro Zeiteinheit $d := w_0$.
- Eine Abbildung Φ, die jeder Kante eine Menge von Sperrungen zuordnet. Dabei ist $\Phi(e) = \{[a_1, b_1), ..., [a_{\phi(e)}, b_{\phi(e)})\}$ und $\phi(e) = |\Phi(e)|$ für eine Kante $e \in E$. Es gilt $a_i < b_i$ für alle $i \in \{1, ..., \phi(e)\}$. Die Dauer einer Sperrung $[a, b)$ ist gegeben durch $b - a$.

Aus G können wir verschiedene Funktionen ableiten. Als erstes definieren wir eine Distanzfunktion $dist_v : V \to \mathbb{N}_0$. Dabei gibt $dist_v(u)$ die Dauer des schnellsten Weges von u nach v an, wenn es in G keine Sperrungen gibt.
Zusätzlich definieren wir für jede Kante $e = (u, v) \in E$ eine Hilfsfunktion $\tau_e^{\rightarrow} : \mathbb{N}_0 \to \mathbb{N}_0$. $\tau_e^{\rightarrow}(t)$ gibt dabei die kürzeste Reisedauer an, um die Kante e zu traversieren und zum Zeitpunkt t am Knoten v zu sein. Formal ist $\tau_e^{\rightarrow}(t)$ in Gleichung 2.1 definiert.

$$\tau_e^{\rightarrow}(t) = \min\{t_0 \in \mathbb{N}_0 \mid [t - t_0, t) - \sum_{x \in \Phi(e)} |[t - t_0, t) \cap x| \geq \delta(e)\} \tag{2.1}$$

Daraus ergibt sich, dass $t - \tau_e^{\rightarrow}(t)$ der späteste Abfahrtszeitpunkt am Knoten u ist, um zum Zeitpunkt t am Knoten v zu sein. Wir nennen diese Hilfsfunktion auch die Reisezeitfunktion der Kante e. Die Reisezeitfunktion ist eine stückweise lineare Funktion. Sie weist keine Sprünge nach oben auf, Sprünge nach unten sind jedoch möglich. Dies hat immer das Ende einer Sperrung zur Ursache.
Wir definieren für jede Kante $e \in E$ eine weitere Hilfsfunktion $\tau_e^{\leftarrow} : \mathbb{N}_0 \to \mathbb{N}_0$. Diese gibt für einen Zeitpunkt t an, wie lange es minimal dauert die Kante $e = (u, v)$ zu traversieren, wenn man zum Zeitpunkt t am Knoten u losfährt. Also ist $t + \tau_e^{\leftarrow}(t)$ der frühest mögliche Ankunftszeitpunkt am Knoten v. Diese Funktion ist stückweise linear und kann Sprünge nach oben aufweisen. Dies hat den Beginn einer Sperrung zur Ursache.

2.2 Routen

Eine u-v-Route ist ein Tripel (R, A, D). Dabei gilt $l := |R| = |A| = |D|$.
Die Menge $R = \{v_1, ..., v_l\}$ enthält die auf der Route besuchten Knoten und beschreibt einen Pfad. Insbesondere ist $v_1 = u$ und $v_l = v$. Die Mengen $A = \{a_1, ..., a_l\}$ und $D = \{d_1, ..., d_l\}$ enthalten die zu den Knoten gehörigen Ankunfts- und Abfahrtsszeitpunkte.
Eine Route ist gültig, wenn die Bedingungen in Gleichung 2.2 erfüllt sind.

$$\begin{aligned}
&\forall i \in \mathbb{N}\ mit\ 1 \leq i < l: && (v_i, v_{i+1}) \in E \\
&\forall i \in \mathbb{N}\ mit\ 1 \leq i \leq l: && a_i \leq d_i \\
&\forall i \in \mathbb{N}\ mit\ 1 \leq i < l: && (a_{i+1} - d_i) - \textstyle\sum_{x \in \Phi(v_i, v_{i+1})} |[d_i, a_{i+1}) \cap x| \geq \delta(v_i, v_{i+1})
\end{aligned} \tag{2.2}$$

Wenn wir im Folgenden von einer Route sprechen, nehmen wir implizit an, dass diese Bedingungen erfüllt sind.
Die Kosten $cost(R, A, D)$ einer Route (R, A, D) sind in Gleichung 2.3 definiert.

$$cost(R, A, D) = \sum_{i=1}^{l-1} (a_{i+1} - d_i) \cdot d + \sum_{i=2}^{l-1} (d_i - a_i) \cdot w_{\rho(v_i)} \tag{2.3}$$

Wir berechnen im ersten Term die Kosten für die gefahrene Zeit und im zweiten Term die Kosten für das Warten an Knoten. Dabei ist Warten an Start- und Zielknoten kostenlos.
Eine u-v-Route (R, A, D) liegt in einem Horizont $\mathcal{H}$, falls $a_1, d_l \in \mathcal{H}$ gilt.

2.3 Pareto-Optimalität

Wir nennen eine u-v-Route (R, A, D) bezüglich eines Horizontes $\mathcal{H}$ pareto-optimal (oder nur optimal), falls für alle anderen u-v-Routen (R', A', D') in $\mathcal{H}$ nicht gilt, dass $cost(R', A', D') < cost(R, A, D)$ und $a_l > a_{l'}$. Es gibt also keine andere Route im Horizont mit geringeren Kosten, die früher am Zielknoten ankommt.

2.4 Basisproblem

Eine Anfrage für das Basisproblem besteht aus einem Startknoten $s \in V$, einem Zielknoten $z \in V$ und einem Planungshorizont $[t_{min}, t_{max}]$. Im Rahmen der später vorgestellten Algorithmen werden wir immer $\mathcal{H} := \{t \in \mathbb{N}_0 | t_{min} \leq t \leq t_{max}\}$ als Planungshorizont verwenden. Wir suchen eine maximale Menge optimaler s-z-Routen bezüglich des Planungshorizontes. Wir bezeichnen das Basisproblem als *RouBIRT*-Problem. Der Name bedeutet "Routing with Ban Intervals, Ratings and Travel time costs". Das *RouBIRT*-Problem können wir mit dem *RouBIRT*-Algorithmus lösen. Später werden wir diesen erweitern, um das *RouBIRT*-Problem effizienter lösen zu können.

2.5 Modifiziertes Basisproblem

Wir modifizieren das *RouBIRT*-Problem und erhalten dadurch das *RouBIRT-DB*-Problem, um auch Lenk- und Ruhezeiten zu berücksichtigen. Der Name steht dabei für "RouBIRT with Driving time Breaks". Eine Anfrage beinhaltet zusätzlich zwei Parameter *limit* und *break*. Der Fahrer darf höchstens *limit* am Stück fahren, bis er eine Pause der Länge *break* machen muss. Dabei zählt das Warten an Nicht-Parkplätzen und auf Kanten nicht als Pause und auch nicht als Fahrzeit. Für das Traversieren einer Kante $e \in E$ wird also nur $\delta(e)$ als Fahrzeit berechnet, auch wenn das Traversieren der Kante Warten auf dieser beinhaltet. Formal ist eine Route, die die Bedingungen in Gleichung 2.2 erfüllt, auch bezüglich der Fahrzeitlimitierung gültig, falls eine Menge von Indizes $B := \{i_1, ..., i_b\}$ $(i_1 < ... < i_b)$ existiert, die die Bedingungen in Gleichung 2.4 erfüllt.

$$\begin{aligned} & i_1 = 1 \wedge i_b = l \\ \forall k \in \{1, ..., b-1\}: \quad & \textstyle\sum_{j=i_k}^{i_{k+1}-1} \delta(v_j, v_{j+1}) \leq limit \\ \forall k \in \{2, ..., b-1\}: \quad & \rho(v_{i_k}) > 0 \wedge d_{i_k} - a_{i_k} \geq break \end{aligned} \tag{2.4}$$

Wir suchen eine maximale Menge optimaler s-z-Routen unter allen s-z-Routen, bei denen der Fahrer nie länger als *limit* am Stück fährt. Wir können das *RouBIRT-DB*-Problem mit dem *RouBIRT-DB*-Algorithmus lösen.

3. Basisalgorithmus

In diesem Kapitel beschreiben wir den Basisalgorithmus, der das *RouBIRT*-Problem löst. Dieser wurde bereits in [KSWZ20] vorgestellt. Wir werden den Basisalgorithmus im Folgenden als *RouBIRT*-Algorithmus bezeichnen.

3.1 Beschreibung

Für den *RouBIRT*-Algorithmus definieren wir für jeden Knoten $v \in V$ eine Kostenfunktion $C_v : \mathcal{H} \to \mathbb{N}_0 \cup \{\infty\}$. Dabei gibt $C_v(t)$ die nötigen Kosten an, um zum Zeitpunkt t am Knoten v zu sein. Wir bezeichnen C_v auch als das Kostenprofil des Knoten v. Das Kostenprofil eines Knotens v ist eine stückweise lineare Funktion, da die Wartekosten auf Knoten und Kanten linear sind. Es weist keine Sprünge nach oben auf, Sprünge nach unten sind aber möglich. Ein Sprung nach unten bedeutet eine Änderung des Pfades zum Knoten v oder eine Änderung von Wartekosten. Im Laufe des Algorithmus werden die Kostenprofile der Knoten immer weiter verbessert.
Der *RouBIRT*-Algorithmus ist als Pseudocode in Algorithmus 3.1 dargestellt. Der Pseudocode ist aus [Wag19] entnommen. Die Eingabe besteht aus einem Graphen G, einem Startknoten $s \in V$, einem Zielknoten $z \in V$, einem Planungshorizont $\mathcal{H}$ und einer Menge von Wartekosten W.
In der Initialisierung des Algorithmus setzen wir für alle Knoten $v \in V$ das zugehörige Kostenprofil $C_v(t) = \infty$ $(t \in \mathcal{H})$. Für s setzen wir $C_s(t) = 0$ $(t \in \mathcal{H})$. Anschließend fügen wir (s, t_{min}) in die Priority-Queue (im Folgenden nur Queue) ein.
In der Haupt-Schleife nehmen wir das oberste Element (v, t_{relax}) aus der Queue und relaxieren alle Kanten $(u, v) \in E$. Dabei gehen wir für jede Kante $e = (u, v)$ wie folgt vor.

Linken

Im ersten Schritt berechnen wir gemäß Gleichung 3.1 ein Fahrt-Profil C'_v.

$$\forall t \in \mathcal{H}, t_{relax} \leq t - \tau_e^{\rightarrow}(t), t \leq t_{max} : C'_v(t) = C_u(t - \tau_e^{\rightarrow}(t)) + \tau_e^{\rightarrow}(t) \cdot d \tag{3.1}$$

Für alle anderen Ankunftszeiten t setzen wir $C'_v(t) = \infty$.
Um Warten am Knoten v zu berücksichtigen, berechnen wir im zweiten Schritt gemäß Gleichung 3.2 ein Warte-Profil C''_v.

$$\forall t \in \mathcal{H} : C''_v(t) = \min_{t' \in \mathcal{H}, t' \leq t} C'_v(t') + (t - t') \cdot w_{\rho(v)} \tag{3.2}$$

Algorithmus 3.1 : Berechnung der Pareto-Front

Input : Graph $G = (V, E, \delta, \rho, \Phi)$, source node s, target node t, planning horizon $\mathcal{H} = [t_{min}, t_{max}]$, waiting costs W
Data : Priority queue Q
Output : Pareto-Set of arrival time and cost at z

```
// Initialization
forall v ∈ V do
    forall t ∈ H do
        C_v(t) ← ∞
forall t ∈ H do
    C_s(t) ← 0
Q.INSERT(s, t_min)
// Main loop
while Q is not empty do
    (u, t_relax) ← Q.DELETEMIN()
    forall (u, v) ∈ E do
        C''_v ← LINK(u,v,t_relax)
        t_update ←MERGE(C_v, C''_v)
        if t_update ≠ ⊥ and t_update ≤ t_max then
            if Q.CONTAINS(v) then
                Q.DECREASEKEY(v, t_update)
            else
                Q.INSERT(v, t_update)
return {(t, C_z(t)) | t ∈ H, C_z(t) < ∞, ∄t' ∈ H : t' < t ∧ C_z(t') ≤ C_z(t)}
```

Dabei wird geprüft, ob es billiger ist, zu einem früheren Zeitpunkt an v anzukommen und dann an v zu warten.

Mergen

Wir mergen nun das neu berechnete Profil C''_v mit dem bereits vorhandenen Profil C_v.

$$C_v(t) \leftarrow \min\{C_v(t), C''_v(t)\}$$

Dabei merken wir uns den ersten Zeitpunkt t_{update}, zu dem C_v verbessert werden konnte. Falls C_v nicht verbessert werden konnte, setzen wir $t_{update} = \perp$.
Falls C_v verbessert werden konnte, aktualisieren wir den Queue-Eintrag vom Knoten v oder fügen ihn in die Queue ein, falls er noch nicht in der Queue enthalten ist.
Der Algorithmus terminiert, wenn die Queue leer ist. Dann sind alle Kostenprofile final und wir geben die Menge der optimalen Pfade in Form ihres jeweiligen Ankunftszeitpunkts und der Kosten des Pfades aus.

3.2 Korrektheit und Laufzeit

Die Korrektheit von *RouBIRT* wurde bereits in [KSWZ20] begründet. Dort wurde zudem bewiesen, dass die Laufzeit polynomiell in $|V|$, $|E|$, $|W|$ und $\sum_{e \in E} |\Phi(e)|$ ist.

3.3 Beschleunigungstechniken

Um den *RouBIRT*-Algorithmus zu beschleunigen, wurden bei der Implementierung mehrere Beschleunigungstechniken verwendet.

3.3.1 Pruning

Target Pruning

Mithilfe der Distanzfunktion $dist_z$ des Ziel-Knotens z, deren Berechnung in Abschnitt 3.3.3 beschrieben ist, können wir Pfade, die nicht zu einem optimalen Pfad erweitert werden können, abschneiden. Dazu berechnen wir vor dem Explorieren eines Knotens v zum Zeitpunkt t die minimalen Kosten des aktuellen Pfades am Ziel-Knoten sowie den frühesten Ankunftszeitpunkt am Ziel-Knoten. Der aktuelle Pfad ist dabei der (nicht notwendigerweise eindeutige) Pfad von s nach v, der zum Zeitpunkt t mit Kosten $C_v(t)$ am Knoten v ist. Die notwendige Bedingung für das Pruning ist in Gleichung 3.3 dargestellt.

$$C_v(t) + dist_z(v) \cdot d > C_z(t + dist_z(v)) \tag{3.3}$$

Wenn die minimalen Kosten echt größer sind als die Kosten im Zielprofil zum frühest möglichen Ankunftszeitpunkt am Ziel, dann kann der aktuelle Pfad nicht zu einem optimalen Pfad führen.

Pruning mit Bounds

Wir speichern für jeden Knoten v eine obere Schranke $upper(v)$, sodass $C_v(t) \leq upper(v)$ für alle $t \in \mathcal{H}$ gilt. Dabei ist $upper(v)$ kleinstmöglich unter all diesen Schranken. Analog dazu speichern wir eine untere Schranke $lower(v)$. Beim Relaxieren einer Kante $e = (u, v)$ können wir dann prüfen, ob $lower(u) + \delta(e) \cdot d > upper(v)$ gilt. In diesem Fall können wir das Kostenprofil C_v nicht verbessern und wir können diese Kante überspringen.

Hopping-Reduction

Das Kostenprofil C_v eines Knotens v ist eine stückweise lineare Funktion. Wir können C_v also in eine Menge von linearen Segmenten unterteilen. Jedes dieser Segmente repräsentiert einen Pfad vom Startknoten s zum Knoten v. Wir speichern zu jedem dieser Segmente den vorherigen Knoten auf dem jeweiligen Pfad. Stimmt dieser für alle Segmente überein, dann nennen wir diesen Knoten $parent(v)$. Wenn v kein Parkplatz ist, dann können wir (falls $(v, parent(v)) \in E$ ist) vom Knoten v aus das Kostenprofil am Knoten $parent(v)$ nicht mehr verbessern. Wir können also die Kante $(v, parent(v))$ überspringen.

3.3.2 A*-Suche

Das Konzept der A*-Suche wurde 1968 in [HNR68] vorgestellt und ist eine Erweiterung von Dijkstra's Algorithums zum Berechnen kürzester Pfade in Graphen. Durch eine Anpassung der Reihenfolge, in der die Knoten exploriert werden, wird dabei der Suchraum eingeschränkt.
Bei der A*-Suche wird eine Potentialfunktion π_z verwendet, die die Distanz von einem Knoten v zum Ziel-Knoten z abschätzt. Die Potentialfunktion muss dabei unter anderem die Bedingung in Gleichung 3.4 erfüllen.

$$\forall e = (u, v) \in E : \delta(e) - \pi_z(u) + \pi_z(v) \geq 0 \tag{3.4}$$

Zudem darf die Potentialfunktion die tatsächliche Distanz zum Ziel nicht überschätzen. In Dijkstra's Algorithmus wird dann für einen Knoten v $td(v) + \pi_z(v)$ gespeichert (statt $td(v)$). Dabei ist $td(v)$ die tentative Distanz des Knotens v. Dijkstra's Algorithmus wird dadurch zu einer zielgerichteten Suche. Je nach Güte der Potentialfunktion wird die Suche entsprechend beschleunigt.
Im *RouBIRT*-Algorithmus verwenden wir als Potentialfunktion $dist_z$. Diese Potentialfunktion wird auch in [SZ21] vorgestellt. Wir fügen statt dem frühesten Zeitpunkt t_{update}, zu dem das Kostenprofil eines Knoten v verbessert werden konnte, $t_{update} + dist_z(v)$ in die Queue ein. Dadurch wird der *RouBIRT*-Algorithmus zielgerichtet.

3.3.3 Distanz-Berechnung

Die Distanzfunktion kann mithilfe von vorberechneten *Contraction Hierarchies* berechnet werden. Dabei wird jedem Knoten eine Priorität zugewiesen und die Knoten werden nach aufsteigender Priorität kontrahiert. Dabei wird der jeweilige Knoten aus dem Graphen entfernt und gegebenfalls Abkürzungen zwischen seinen Nachbarknoten eingefügt. Die genaue Berechnung der *Contraction Hierarchies* und das Berechnen der kürzesten Distanzen daraus wird in [GSSV12] beschrieben.
Im *RouBIRT*-Algorithmus berechnen wir in der Initialisierung die Distanz des Startknotens $dist_z(s)$ zum Zielknoten. Wenn wir einen Knoten zum ersten Mal über eine Kante erreichen, dann berechnen wir das Potential dieses Knotens und relaxieren anschließend die Kante. Wir verwenden dabei die Implementierung der *Contraction Hierarchies* von [Str19].

4. Beschleunigter Basisalgorithmus

In diesem Kapitel erweitern wir den *RouBIRT*-Algorithmus um eine Rückwärtssuche und zusätzliches Pruning, um ihn zu beschleunigen. Wir nennen den daraus entstehenden Algorithmus *BD-RouBIRT*-Algorithmus. Der Name steht dabei für "BiDirectional RouBIRT".

4.1 Rückwärtssuche

Die Rückwärtssuche berechnet für jeden Knoten ein Rückwärtsreisezeitprofil Θ_v (im Folgenden nur Rückwärtsprofil). Dieses gibt für jeden Zeitpunkt die kürzeste Reisedauer an, um von v aus den Zielknoten z zu erreichen. Falls z nicht vor Ende des Planungshorizontes erreicht werden kann, dann ist ist das Rückwärtsprofil an dieser Stelle ∞. Es gilt also, dass für einen Abfahrtszeitpunkt $t \in \mathcal{H}$ am Knoten v der frühest mögliche Ankunftszeitpunkt am Ziel $t + \Theta_v(t)$ ist (falls $\Theta_v(t) < \infty$). Diese Funktion ist eine stückweise lineare Funktion und kann Sprünge nach oben aufweisen. Dies hat immer den Beginn einer Sperrung zur Ursache. Das Ziel der Rückwärtssuche ist es, die Rückwärtsprofile der Knoten zu berechnen.

Die Rückwärtssuche arbeitet dabei analog zu *RouBIRT*. Der Pseudocode ist in Algorithmus 4.1 als Teil von *BD-RouBIRT* dargestellt. Das Explorieren eines Knotens ist in den Zeilen 20-28 beschrieben. In einer *priority queue* Q_{bw} werden Knoten zusammen mit dem größten Zeitpunkt, zu dem das Rückwärtsprofil des jeweiligen Knoten zuletzt aktualisiert wurde, gespeichert. Dabei ist zu beachten, dass die Zeitpunkte, die aus Q_{bw} entnommen werden, monoton fallend sind. Q_{bw} ist also eine *max-priority queue*.
In der Initialisierung der Rückwärtssuche wird für alle $t \in \mathcal{H}$ $\Theta_z(t) = 0$ und für alle anderen Knoten v $\Theta_v(t) = \infty$ gesetzt. Zudem wird (z, t_{max}) in Q_{bw} eingefügt.
In der Hauptschleife wird ein Knoten v zusammen mit dem Zeitpunkt t_{relax} aus der Queue entnommen. Für jede Kante $e = (u, v)$ gehen wir in zwei Schritten vor, um die Kante zu relaxieren.

Linken

Im ersten Schritt berechnen wir für den Knoten u ein neues Rückwärtsprofil Θ'_u gemäß der Vorschrift in Gleichung 4.1.

$$\forall t \in \mathcal{H}, t + \tau_e^{\leftarrow}(t) \leq t_{relax} : \Theta'_u(t) = \tau_e^{\leftarrow}(t) + \Theta_v(t + \tau_e^{\leftarrow}(t) \tag{4.1}$$

Da wir in der Rückwärtssuche nur minimale Reisezeiten berechnen und keine Kosten für Fahren und Warten berücksichtigen, müssen wir kein Warte-Profil berechnen.

Mergen

Falls für u bisher noch kein Rückwärtsprofil berechnet wurde, wird $\Theta_u = \Theta'_u$ gesetzt. Ansonsten wird $(\forall t \in \mathcal{H})\ \Theta_u(t) = \min\{\Theta_u(t), \Theta'_u(t)\}$ gesetzt. Wenn $\Theta_u(t)$ dadurch verbessert werden konnten, merken wir uns den größten Zeitpunkt t_{update}, zu dem Θ_u verbessert werden konnte. Ist u bereits in Q_{bw} enthalten, dann aktualieren wir den entsprechenden Eintrag. Ansonsten fügen wir (u, t_{update}) in die Queue ein.

Die Rückwärtssuche terminiert, wenn die Queue leer ist. Dann sind alle Rückwärtsprofile final. Wenn wir während dem Algorithmus einen Zeitpunkt t_{relax} aus der Queue entnehmen, dann sind die Rückwärtsprofile aller Knoten für die Zeitpunte $t \geq t_{relax}$ final.

Laufzeit und Korrektheit

Die Rückwärtssuche kann als Variante von *RouBIRT* betrachtet werden, indem wir die Kosten für das Warten und Fahren $W = \{w_0 = 1\}$ setzen. Wir suchen dabei zudem keine optimalen Pfade, sondern Pfade mit minimaler Reisedauer. Dadurch sind die Beweise für die Korrektheit und polynomielle Laufzeit der Rückwärtssuche analog zu den jeweiligen Beweisen des Grundalgorithmus. Sie werden daher im Folgenden nicht aufgeführt. Die Laufzeit ist (asymptotisch) nicht langsamer als die des *RouBIRT*-Algorithmus.

Algorithmus 4.1 : Berechnung der Pareto-Front mit Rückwärtssuche

Input : Graph $G = (V, E, \delta, \rho, \Phi)$, source node s, target node z, planning horizon $\mathcal{H} = [t_{min}, t_{max}]$, waiting costs $W = \{w_0, ..., w_r\}$
Data : Min-Priority queue Q_{fw}, Max-Priority Queue Q_{bw}
Output : Pareto-Set of arrival time and cost at z

```
// Initialization
forall v ∈ V do
    forall t ∈ H do
        C_v(t) ← ∞, Θ_v(t) ← ∞
forall t ∈ H do
    C_s(t) ← 0, Θ_z(t) ← 0
Q_fw.INSERT(s, t_min)
Q_bw.INSERT(z, t_max)
// Main loop
while Q_fw is not empty do
    if SETTLEFORWARD() then
        (u, t_relax) ← Q_fw.DELETEMIN()
        PRUNE(u,t_relax)
        GETPRELIMINARYTARGETCOSTS(u,t_relax)
        forall (u, v) ∈ E do
            C''_v ← link_fw(u,v,t_min)
            t_update ←MERGE(C_v, C''_v)
            if t_update ≠ ⊥ and t_update ≤ t_max then
                if Q_fw.CONTAINS(v) then
                    Q_fw.DECREASEKEY(v, t_update)
                else
                    Q_fw.INSERT(v, t_update)
    else
        (v, t_relax) ← Q_bw.DELETEMAX()
        forall (u, v) ∈ E do
            Θ'_u ← link_bw(u,v,t_relax)
            t_update ←MERGE(Θ_u, Θ'_v)
            if t_update ≠ ⊥ and t_update ≥ t_min then
                if Q_bw.CONTAINS(v) then
                    Q_bw.INCREASEKEY(v, t_update)
                else
                    Q_bw.INSERT(v, t_update)
return {(t, C_z(t)) | t ∈ H, C_z(t) < ∞, ∄t' ∈ H : t' < t ∧ C_z(t') ≤ C_z(t)}
```

4.2 Beschreibung

Wir erweitern den *RouBIRT*-Algorithmus um die soeben beschriebene Rückwärtssuche. Durch alleiniges Hinzunehmen der Rückwärtssuche bleiben die Korrektheit und die polynomielle Laufzeit erhalten. Dass dies auch für das Hinzufügen des Reisezeit-Prunings und der vorläufigen Zielkosten gilt, werden wir an den entsprechenden Stellen beweisen.
Die Initialisierung erfolgt wie in den beiden vorherigen Abschnitten beschrieben. In der Hauptschleife wird jeweils ein Knoten aus Q_{fw} oder Q_{bw} exploriert (entsprechend vorwärts bzw. rückwärts). Wir entscheiden nach in Abschnitt 7 näher beschriebenen Strategien, ob wir im aktuellen Schleifendurchlauf einen Knoten in der Vorwärts- oder Rückwärtssuche explorieren. Beim Explorieren eines Knotens in der Rückwärtssuche gehen wir wie in Abschnitt 4.1 beschrieben vor. Beim Explorieren eines Knotens in der Vorwärtssuche gehen wir mit zwei Ausnahmen wie in Kapitel 3 vor. Diese beiden Ausnahmen sind das Reisezeit-Pruning und das Berechnen von vorläufigen Kosten im Kostenprofil des Ziel-Knotens. In Pseudocode 4.1 sind diese Stellen rot hervorgehoben.
Im Folgenden beschreiben wir das Reisezeit-Pruning und die Berechnung der vorläufigen Zielkosten näher.

Target Pruning

In diesem Abschnitt bezeichnen wir das Target Pruning nur als Pruning. Wir erweitern das in Abschnitt 3.3.1 beschriebene Target Pruning. Mithilfe der Rückwärtssuche können wir unter bestimmten Bedingungen aussagen, dass es keinen optimalen Pfad gibt, der sich zum Zeitpunkt t am Knoten v befindet. Dies können wir für Zeitpunkte machen, zu denen das Rückwärtsprofil am Knoten v bereits final ist. Im Folgenden sei t_{bw} das Minimum unter allen Zeitpunkten, bei denen dies der Fall ist. Im Laufe des Algorithmus gilt stets, dass t_{bw} gleich dem Key des obersten Elementes von Q_{bw} ist. In Theorem 4.1 machen wir eine Aussage darüber, wann wir aussagen können, dass ein Pfad nicht optimal sein kann. Zudem zeigt der Beweis, dass die Korrektheit erhalten bleibt.

Theorem 4.1. *Sei $v \in V$ ein Knoten, C_v das zugehörige Kostenprofil. Der zum Kostenwert $C_v(t)$ gehörende Pfad ist nicht Teil eines optimalen Pfades, wenn $t \geq t_{bw}$ ist und eine der folgenden Bedingungen gilt:*

- $t + \Theta_v(t) > t_{max}$
- $t + \Theta_v(t) \leq t_{max}$ *und* $C_v(t) + dist_z(v) \cdot d > C_z(t + \Theta_v(t))$

Wir sagen dann, dass der Kostenwert $C_v(t)$ vernachlässigbar ist.

Beweis. Wenn die erste Bedingung erfüllt ist, dann ist es nicht möglich, innerhalb des Planungshorizonts am Ziel anzukommen. Also gibt es keinen optimalen Pfad, der sich zum Zeitpunkt t am Knoten v befindet (unabhängig vom Kostenwert $C_v(t)$).
Ansonsten sei p ein beliebiger Pfad vom Start zum Ziel, der zum Zeitpunkt t mit Kosten $C_v(t)$ am Knoten v ist. Dann hat p mindestens Kosten $C_v(t)+dist_z(v)\cdot d$, da ab dem Knoten v noch mindestens für $dist_z(v)$ Zeiteinheiten zum Ziel gefahren werden müssen. Außerdem kommt p frühestens zum Zeitpunkt $t + \Theta_v(t)$ am Ziel an. Sei p' der zum Kostenwert $C_z(t + \Theta_v(t)$ gehörende Pfad. Dann kommt p' nicht später als p am Ziel an und hat echt kleinere Kosten, also ist p nicht optimal. □

Ist ein Kostenwert $C_v(t)$ vernachlässigbar, dann setzen wir $C_v(t) = \infty$.
Das Pruning kann durch einen einfachen Sweepline-Algorithmus über C_v, Θ_v und C_z umgesetzt werden. Daher ist die Laufzeit linear in der summierten Größe von C_v, Θ_v und C_z, was die polynomielle Gesamtlaufzeit erhält.

Vorläufige Zielkosten

Wir berechnen beim Explorieren des Knotens v basierend auf dem Rückwärtsprofil gemäß Gleichung 4.2 einen vorläufigen Kostenwert im Ziel. Auf diese Weise können wir früher Kostenwerte am Ziel-Knoten erhalten.

$$\forall t \in \mathcal{H}, t \geq t_{relax}, t + \Theta_v(t) \leq t_{max} : C'_z(t + \Theta_v(t)) = C_v(t) + \Theta_v(t) \cdot d \tag{4.2}$$

Für alle anderen Zeitpunkte setzen wir $C'_z(t) = \infty$. Danach mergen wir C_z und C'_z.
In Theorem 4.2 zeigen wir, dass durch die vorläufigen Zielkosten keine ungültigen Kosten im Kostenprofil des Ziel-Knotens entstehen. Das heißt, dass für die so berechneten Kostenwerte stets ein Pfad von s nach z mit höchstens den jeweils berechneten Kosten existiert. Die Korrektheit bleibt also erhalten.

Theorem 4.2. *Sei $v \in V$ ein Knoten, $v \neq z$ und $t \in \mathcal{H}$ mit $t + \Theta_v(t) \leq t_{end}$. Dann existiert ein Pfad p von s nach z, der zum Zeitpunkt t an v ist, zum Zeitpunkt $t + \Theta_v(t)$ an z ankommt und dessen Kosten höchstens $C_v(t) + \Theta_v(t) \cdot d$ betragen.*

Beweis. Sei p_1 der zum Kostenwert $C_v(t)$ gehörige Pfad. Sei p_2 der zur Reisezeit $\Theta_v(t)$ gehörende Pfad. Der Pfad p_2 hat höchstens Kosten $\Theta_v(t) \cdot d$, da alle Wartekosten $w_i \leq w_0 = d$ sind. Wir erhalten p durch Verknüpfen der Pfade p_1 und p_2. Da p_1 und p_2 jeweils valide Pfade sind, ist auch p valide. Da p_2 zum Zeitpunkt $t + \Theta_v(t)$ an z ankommt, gilt dies auch für p. p befindet sich trivialerweise zum Zeitpunkt t am Knoten v. Für die Kosten des Pfades p gilt:

$$cost(p) = cost(p_1) + cost(p_2) \leq C_v(t) + \Theta_v(t) \cdot d \tag{4.3}$$

□

Die Berechnung der vorläufigen Zielkosten kann durch einen einfachen Sweepline-Algorithmus über C_v, Θ_v und C_z umgesetzt werden. Die Laufzeit ist also linear in der summierten Größe, wodurch die Gesamtlaufzeit polynomiell bleibt.

5. Modifizierter Basisalgorithmus

In diesem Kapitel modifizieren wir den *RouBIRT*-Algorithmus, sodass er das in Abschnitt 2.5 vorgestellte *RouBIRT-DB*-Problem löst. Den resultierenden Algorithmus nennen wir *RouBIRT-DB*-Algorithmus.

5.1 Beschreibung

Um das *RouBIRT-DB*-Problem zu lösen, modifizieren wir den *RouBIRT*-Algorithmus. Wir speichern uns an einem Knoten nicht mehr nur ein Profil, sondern mehrere. Jedes dieser Profile gehört dabei zu einer akkumulierten Fahrzeit seit der letzten Pause. Formal ausgedrückt gibt es für jeden Knoten v eine Menge A_v, die alle akkumulierten Fahrzeiten a enthält, für die es am Knoten v ein Profil mit akkumulierter Fahrzeit a gibt. Ein Profil mit zugehöriger akkumulierter Fahrzeit a schreiben wir als $C_{v,a}$. Wir speichern uns für jeden Knoten v eine Menge $S_v = \{C_{v,a} | a \in A_v\}$. Dabei soll stets gelten, dass jedes Profil $C_{v,a}$ nur an Zeitpunkten definiert ist, an denen es von keinem anderen Profil $C_{v,a'}$ dominiert wird. Definition 5.1 sagt aus, wann ein Profil ein anderes Profil zu einem Zeitpunkt t dominiert.

Definition 5.1. *Seien $C_{v,a}, C_{v,a'} \in S_v$, $a < a'$ und $t \in \mathcal{H}$. $C_{v,a}$ dominiert $C_{v,a'}$ zum Zeitpunkt t, falls $C_{v,a}(t) \leq C_{v,a'}(t)$ gilt.*

Der *RouBIRT-DB*-Algorithmus ist in Algorithmus 5.1 dargestellt.

In der Initialisierungsphase setzen wir $A_v, S_v = \emptyset$ für alle Knoten außer dem Startknoten. Für diese Knoten gibt es zu Beginn des Algorithmus noch keine Kostenprofile. Der Startknoten s hat ein Profil $C_{s,0}$ mit einer akkumulierten Fahrzeit 0, das zu allen Zeitpunkten gleich 0 ist. Wir fügen entsprechend 0 in A_s und $C_{s,0}$ in S_s ein. Anschließend fügen wir wie vorher (s, t_{min}) in die Queue ein.
In der Hauptschleife entnehmen wir wie vorher einen Knoten u mit einem Zeitpunkt t_{relax} aus der Queue. Wir relaxieren jede Kante mit jedem Kostenprofil in S_u. Für eine Kante $e = (u, v)$ und ein Kostenprofil $C_{u,a}$ berechnen wir dazu zunächst die neue akkumulierte Fahrzeit $a_{new} = a + \delta(e)$ am Knoten v. Falls $a_{new} > limit$ ist, dann ist das Relaxieren dieser Kante mit dem Profil $C_{u,a}$ nicht erlaubt und wir fahren mit dem nächsten Durchlauf der *for*-Schleife fort. Die Berechnung des neuen Kostenprofils $C''_{v,a_{new}}$ funktioniert so wie in Abschnitt 3.1 beschrieben. Wir entfernen aus $C''_{v,a_{new}}$ alle Zeitpunkte (setzen den zugehörigen Wert auf ∞), zu denen $C''_{v,a_{new}}$ von einem anderen Profil aus S_v dominiert wird. Wenn am Knoten v bereits ein Kostenprofil $C_{v,a_{new}}$ existiert, dann mergen wir $C_{v,a_{new}}$ und $C''_{v,a_{new}}$

und setzen t_{update} entsprechend. Sonst fügen wir, falls $C''_{v,a_{new}}$ nicht zu allen Zeitpunkten ∞ ist, $C''_{v,a_{new}}$ in S_v und a_{new} in A_v ein und setzen t_{update} auf den frühesten Zeitpunkt, zu dem $C''_{v,a_{new}}$ nicht ∞ ist.
Falls wir $C_{v,a_{new}}$ verbessern konnten und der Knoten v ein Parkplatz ist, berechnen wir aus $C_{v,a_{new}}$ ein neues Kostenprofil $\widetilde{C}_{v,0}$ mit akkumulierter Fahrtzeit 0. Dazu verschieben wir $C_{v,a_{new}}$ um $break$ nach rechts und um $break \cdot w_{\rho(v)}$ nach oben:

$$\forall t \in \mathcal{H}, t \geq t_{update} + break : \widetilde{C}_{v,0}(t) = C''_{v,a_{new}}(t - break) + break \cdot w_{\rho(v)} \tag{5.1}$$

Für alle anderen Zeitpunkte t setzen wir $\widetilde{C}_{v,0}(t) = \infty$. Falls S_v bereits ein Kostenprofil $C_{v,0}$ enthält, dann mergen wir $C_{v,0}$ und $\widetilde{C}_{v,0}$. Ansonsten fügen wir $\widetilde{C}_{v,0}$ in S_v und 0 in A_v ein. t_{update} kann hier nicht mehr kleiner werden, da wir $C_{v,0}$ frühestens zum Zeitpunkt $t_{update} + break$ verbessern können.
Wir entfernen nun aus allen anderen Profilen vom Knoten v die Zeitpunkte, zu denen das Profil von einem anderen Profil dominiert wird. Falls dadurch ein Profil zu allen Zeitpunkten gleich ∞ ist, dann entfernen wir dieses Profil aus S_v und die zugehörige akkumulierte Fahrzeit aus A_v. Im letzten Schritt fügen wir wie vorher den Knoten v, falls wir S_v verbessern konnten, in die Queue ein oder aktualisieren den Eintrag von v.
Der Algorithmus terminiert, wenn die Queue leer ist. Dann sind alle Kostenprofile final und wir können das Ergebnis ausgeben. Würden wir die Queue nicht leerlaufen lassen, dann kann es passieren, dass wir die Korrektheit verlieren und die berechnete Menge von Routen nicht maximal ist. Das kann passieren, wenn es eine Route in der maximalen Menge gibt, die erst zu einem sehr späten Zeitpunkt am Ziel ankommt. Für die maximale Menge der Routen ist uns die akkumulierte Fahrzeit am Ziel egal (sofern sie kleiner oder gleich *limit* ist) und berücksichtigen daher alle Profile $C_{z,a}$ am Ziel.
Das Finden eines Pfades zu einem Tupel $(t, C_{z,a}(t))$ funktioniert bis auf eine Änderung gleich wie das Finden des Pfades bei *RouBIRT*. Wir setzen zunächst $R = \{z\}$ und $A = D = \{t\}$ und merken uns den aktuellen Kostenwert $c = C_{z,a}(t)$. Dann führen wir die folgenden Schritte so lange iterativ aus, bis $R[1] = s$ ist.

- Finde Knoten u und Zeitpunkt t, sodass $(u, R[1]) \in E$, $t + \tau^{\rightarrow}_{(u,R[1])}(t) = A[1]$ und $C_{R[1],a}(A[1]) = C_u(t) + w_0 \cdot \tau^{\rightarrow}_e((u, R[1]))$ für ein $a \in A_u$.
- Füge u in R und t in D jeweils vorne ein und setze $c = C_{u,a}(t)$.
- Suche den kleinsten Zeitpunkt $t' \leq t$, für den $C_{u,a}(t') + (t - t') \cdot w_{\rho(u)} = C_{u,a}(t)$. Füge t' vorne in A ein und setze $c = C_{u,a}(t')$.
- Prüfe, ob am Knoten u eine Pause gemacht wurde. Prüfe dazu, ob u ein Parkplatz ist und es ein Profil $C_{u,a'} \in S_v$ gibt, sodass $C_{u,a'}(t' - break) + break \cdot w_{\rho(u)} = c$ gilt. Reduziere dann $A[1]$ um $break$ und setze $c = C_{u,a'}(t' - break)$.

Algorithmus 5.1 : Berechnung der Pareto-Front mit Lenk- und Ruhezeiten

Input : Graph $G = (V, E, \delta, \rho, \Phi)$, source node s, target node t, planning horizon $\mathcal{H} = [t_{min}, t_{max}]$, waiting costs W, driving limit $limit$, break time $break$
Data : Priority queue Q
Output : Pareto-Set of arrival time and cost at z

```
// Initialization
forall v ∈ V do
    S_v ← ∅
S_s ← {C_{s,0} ← 0}
Q.INSERT(s, t_min)
// Main loop
while Q is not empty do
    (u, t_relax) ← Q.DELETEMIN()
    forall (u, v) ∈ E , C_{u,a} ∈ S_u do
        a_new ← a + δ(u, v)
        if a_new > limit then
            continue
        C''_{v,a_new} ← LINK(u,v,t_relax,a)
        REMOVEDOMINATED(C''_{v,a_new})
        if A_v. CONTAINS(a_new) then
            t_update ← MERGE(C_{v,a_new}, C''_{v,a_new})
        else
            A_v.INSERT(a_new)
            S_v.INSERT(C''_{v,a_new})
            t_update ← min_{t∈H} C''_{v,a_new} < ∞
        if ρ(v) > 0 and t_update ≠ ⊥ and t_update ≤ t_max then
            C̃_{v,0} ← BREAKPROFILE(C''_{v,a_new})
            if A_v.CONTAINS(0) then
                MERGE(C_{v,0}, C̃_{v,0})
            else
                A_v.INSERT(0)
                S_v.INSERT(C̃_{v,0})
        forall C_{v,a'} ∈ S_v do
            REMOVEDOMINATED(C_{v,a'})
        if t_update ≠ ⊥ and t_update ≤ t_max then
            if Q.CONTAINS(v) then
                Q.DECREASEKEY(v, t_update)
            else
                Q.INSERT(v, t_update)
return
    {(t, C_{z,a}(t)) | t ∈ H, C_{z,a}(t) < ∞, ∄t' ∈ H, a' ∈ A_v : t' < t ∧ C_{z,a'}(t') ≤ C_{z,a}(t)}
```

5.2 Korrektheit

Für die Korrektheit von *RouBIRT-DB* stellen wir Lemma 5.2 auf und beweisen dieses durch Induktion.

Lemma 5.2. *Sei $s \in V$ der Startknoten und $v \in V$ ein beliebiger Knoten. Sei $C^*_{z,a}$ ein finales Kostenprofil der finalen Profilfolge S^*_z eines Algorithmus $\mathcal{A}^*$, der das RouBIRT-DB-Problem korrekt löst. Sei p der zum Kostenwert $C^*_{z,a}(t)$ gehörende Pfad (für beliebiges t). Dann findet auch der RouBIRT-DB-Algorithmus den Pfad p.*

Beweis. Wir zeigen diese Aussage durch Induktion über die Anzahl der Pausen i von p. *Induktionsanfang*: Sei $i = 0$, dann finden wir p aufgrund der Korrektheit des *RouBIRT*-Algorithmus. *Induktionsschluss*: Sei $i > 0$. Sei nun v_0 der letzte Knoten auf p, auf dem der Pfad eine Pause macht. Wir unterteilen p am Knoten v_0 in zwei Teilpfade p_1 und p_2. Dann enthält p_1 $i-1$ Pausen und wir finden p_1 nach *Induktionsvoraussetzung* (*IV*), p_2 finden wir aufgrund der Korrektheit des *RouBIRT*-Algorithmus, da die zu p_2 gehörenden Kostenwerte entlang von p_2 nicht dominiert werden (sonst gäbe es einen besseren Kostenwert für $C_{z,a}$ zum Zeitpunkt t). Da wir auf jedem Parkplatz eine Pause ermöglichen und v_0 ein Parkplatz ist (sonst würde p auf v_0 keine Pause machen), finden wir auch die Pause. Insgesamt findet der *RouBIRT-DB*-Algorithmus also den ganzen Pfad p. □

Offensichtlich berechnet der *RouBIRT-DB*-Algorithmus keine Routen, bei denen der Fahrer länger als *limit* am Stück fährt. Dominierte Routen werden ebenfalls nicht berechnet, da nach jeder Kantenrelaxierung die Invariante wiederhergestellt wird, dass jeder Kostenwert $c < \infty$ nicht dominiert wird und wir am Ende für die Menge der optimalen Routen alle Profile $C_{z,a}$ am Zielknoten betrachten. Zusammen mit Lemma 5.2 folgt die Korrektheit von *RouBIRT-DB*.

5.3 Laufzeit

Die *worst-case* Laufzeit von *RouBIRT-DB* ist mindestens exponentiell in der Anzahl der Knoten, da es an einem Knoten exponentiell viele Profile geben kann, die sich paarweise nicht dominieren. Wir betrachten dazu den Graph in Abbildung 5.1 mit $n+1$ Knoten, der aus [Wag19] entnommen ist. Am Knoten v_{n+1} gibt es dann exponentiell viele Profile, was wir in diesem Abschnitt beweisen werden.

Wir betrachten im Folgenden nur Pfade, die kein Warten auf Knoten beinhalten und bei v_1 beginnen. Der Graph ist so konstruiert, dass das Nehmen der oberen Kante immer das Abwarten der vollständigen Sperrung beinhaltet. Zudem ist mit einer geringeren akkumulierten Fahrzeit immer eine spätere Ankunft am Knoten v_{n+1} verbunden.
Wir stellen Pfade als Wörter p über dem Alphabet $\{o, u\}^n$ dar. Für $p = p_1 p_2 ... p_n$ ist $p_i = o$ genau dann, wenn p vom Knoten v_i zum Knoten v_{i+1} die obere Kante nimmt. Zudem sei

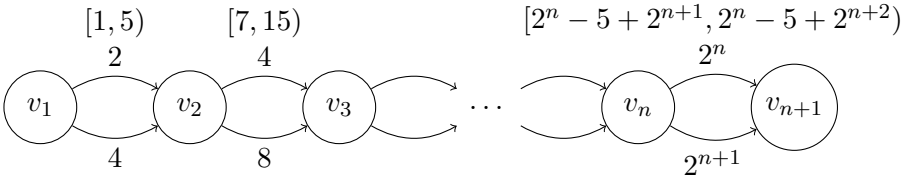


Abbildung 5.1: Graph mit Sperrungen. Die oberen Kanten sind in den angegebenen Intervallen gesperrt. Kein Knoten ist ein Parkplatz. Der Planungshorizont startet zum Zeitpunkt 0.

$arr(p)$ der Ankunftszeitpunkt von p am Knoten $v_{|p|+1}$, $cost(p)$ der dazugehörige Kostenwert und $dt(p)$ die akkumulierte Fahrzeit seit dem Start.
Wir stellen nun zunächst einige Hilfsaussagen auf, um später zu zeigen, dass es am Knoten v_{n+1} exponentiell viele Profile gibt. Für jeden der im Folgenden betrachteten Pfade p soll stets $|p| \leq n$ gelten.

Lemma 5.3. *Sei $i \leq n$ beliebig. Der Pfad o^i kommt zum Zeitpunkt $2^{i+1} - 6 + 2^{i+2}$ am Knoten v_{i+1} an, der Pfad u^i zum Zeitpunkt $\sum_{k=1}^{i} 2^{k+1}$.*

Beweis. Für den Pfad o^i führen wir einen Induktionsbeweis über die Länge des Pfades i.
Induktionsanfang: Für $i = 1$ muss die Sperrung vollständig abgewartet werden. Daher kommt o^1 zum Zeitpunkt $2 + 4 = 6 = 2^2 - 6 + 2^{2+1}$ am Knoten v_2 an.
Induktionsschluss: Sei $i \geq 2$. Der Pfad o^{i-1} kommt nach *IV* zum Zeitpunkt $2^i - 6 + 2^{i+1}$ am Knoten v_i an. Wir fahren nun eine Zeiteinheit, müssen dann die Sperrung der Länge 2^{i+1} abwarten und fahren über den Rest der Kante. Die Ankunftszeit von o^i am Knoten v_{i+1} ist dann $2^i - 6 + 2^{i+1} + 2^i + 2^{i+1} = 2^{i+1} - 6 + 2^{i+2}$.
Auf dem Pfad u^i enthält keine Kante eine Sperrung, weshalb die Ankunft $arr(u^i)$ die Summe der Kantengewichte ist. □

Aus Lemma 5.3 folgt, dass für die Ankunftszeit jedes Pfades p $arr(u^i) \leq arr(p) \leq arr(o^i)$ gilt ($i = |p|$). Wenn wir vom Knoten v_i aus die obere Kante nehmen, dann fahren wir vor Beginn der Sperrung los und es gilt $arr(u^i) + 2^{i+1} = \sum_{k=1}^{i} 2^{k+1} + 2^{i+1} > 2^{i+1} - 5 + 2^{i+2}$. Da $arr(u^i)$ der frühest mögliche Abfahrtszeitpunkt am Knoten v_i ist, können wir also nicht vor Beginn der Sperrung am Knoten v_{i+1} ankommen. Wir müssen also die gesamte Sperrung abwarten und es folgt, dass das Nehmen der oberen Kante immer das Abwarten der gesamten Sperrung beinhaltet.

Lemma 5.4. *Für zwei unterschiedliche Pfade a, b ($|a| = |b| = i$) gilt $|arr(a) - arr(b)| < 2^{i+1}$ und $|dt(a) - dt(b)| < 2^{i+1}$.*

Beweis. Aus Lemma 5.3 folgt $|arr(a) - arr(b)| \leq arr(o^i) - arr(u^i) = 2^{i+1} - 6 + 2^{i+2} - \sum_{k=1}^{i} 2^{k+1} = \cdots = 2^{i+1} - 3 < 2^{i+1}$.
Wir zeigen $|dt(a) - dt(b)| < 2^{i+1}$ per Induktion über i.
Induktionsanfang: Sei $i = 1$. Dann ist $|dt(a) - dt(b)| = 2 < 4 = 2^{1+1}$.
Induktionsschluss: Sei nun $i \geq 2$ und $a = a'x, b = b'y$ zwei Pfade mit $x, y \in \{o, u\}$. Für a', b' gelte die *IV* ($|a'| = |b'| = i - 1$). Wenn $x = y$ ist, dann folgt die Behauptung direkt aus der *IV*. Wenn $x \neq y$ ist, dann wird die absolute Differenz um höchstens $2^{i+1} - 2^i = 2^i$ größer. Es folgt $|dt(a'x) - dt(b'y)| \leq |dt(a') - dt(b')| + 2^i < 2^i + 2^i = 2^{i+1}$. □

Lemma 5.5. *Sei $i \leq n$ beliebig und seien a, b zwei unterschiedliche Pfade ($|a| = |b| = i$). Dann gilt folgende Äquivalenz:*

$$dt(a) < dt(b) \Leftrightarrow arr(a) > arr(b)$$

Beweis. Wir führen einen Induktionsbeweis über i.
Induktionsanfang: Sei $i = 1$. Wenn $dt(a) < dt(b)$ gilt, dann nimmt a die obere Kante, b die untere Kante und es folgt $arr(a) > arr(b)$. Wenn umgekehrt $arr(a) > arr(b)$ ist, dann hat a die obere Kante genommen, b die untere Kante genommen und es folgt $dt(a) < dt(b)$.
Induktionsschluss: Sei $i \geq 2$ und die *IV* gelte für a', b' ($|a'| = |b'| = i - 1$). Wir erweitern diese Pfade zu zwei Pfaden $a'x, b'y$ mit $x, y \in \{o, u\}$. Wenn $x = y$ ist, folgt die Behauptung für $a'x, b'y$ direkt aus der *IV*. Sei nun also $x \neq y$. O.B.d.A. sei $dt(a'x) < dt(b'y)$. Wenn $x = u$ (und somit $y = o$) wäre, dann würde $dt(a'u) = dt(a') + 2^{i+1} < dt(b') + 2^i = dt(b'o)$ und somit $dt(b') - dt(a') - 2^i > 0$ gelten, was im Widerspruch zu $|dt(a') - dt(b')| < 2^i$ (Lemma 5.4)

steht. Also muss $x = o$ und $y = u$ sein. Nun unterscheiden wir zwei Fälle. Im ersten Fall ist $dt(a') < dt(b')$ und es folgt $arr(a') > arr(b')$, woraus wiederum $arr(a'x) > arr(b'x)$ folgt. Im zweiten Fall ist $dt(a') > dt(b')$ und es folgt $arr(a') < arr(b')$, woraus $arr(a'x) > arr(b'y)$ folgt. Für die Implikation $arr(a'x) > arr(b'y) \Rightarrow dt(a'x) < dt(b'x)$ verfahren wir analog. □

Lemma 5.6. *Sei $i \leq n$ beliebig. Für zwei unterschiedliche Pfade a, b ($|a| = |b| = i$) gilt stets $arr(a) \neq arr(b)$ und $dt(a) \neq dt(b)$.*

Beweis. Wir zeigen per Induktion über i, dass $dt(a) \neq dt(b)$ ist. Mit Lemma 5.5 folgt dann $arr(a) \neq arr(b)$.
Induktionsanfang: Sei $i = 1$. Da a, b unterschiedlich sind, muss ein Pfad die obere und der andere die untere Kante nehmen. Daher ist $dt(a) \neq dt(b)$.
Induktionsschluss: Sei $i \geq 2$. Für a', b' gelte die *IV* ($|a'| = |b'| = i - 1$). Wir zeigen nun, dass dann auch $dt(a'x) \neq dt(b'y)$ gilt ($x, y \in \{o, u\}$). Wenn $x = y$, können wir direkt die *IV* anwenden. Im Folgenden ist also $x \neq y$. Nach Lemma 5.4 gilt, dass $|dt(a') - dt(b')| < 2^i$ ist. Da die Fahrzeitdifferenz der jeweils letzten Kanten von $a'x$ und $b'y$ genau 2^i ist, gilt auch $dt(a'x) \neq dt(b'y)$. Falls die IV für a', b' nicht gilt, dann ist $a' = b'$ und für $a'x, b'y$ gilt die Behauptung, falls $x \neq y$ (also $a'x \neq b'y$) ist. □

Diese Hilfsaussagen verwenden wir nun, um Theorem 5.7 zu formulieren und zu beweisen.

Theorem 5.7. *Seien a, b zwei unterschiedliche Pfade zum Knoten v_{n+1}. Dann gilt, dass weder $C_{v_{n+1},dt(a)}$ von $C_{v_{n+1},dt(b)}$ dominiert wird noch die Umkehrung.*

Beweis. Nach Lemma 5.6 gilt $dt(a) \neq dt(b)$, o.B.d.A sei $dt(a) < dt(b)$. Daraus folgt mit Lemma 5.5, dass $arr(a) > arr(b)$ ist. Daher gilt zum Zeitpunkt $arr(b)$ für die entsprechenden Kostenprofile $C_{v_{n+1},dt(a)}(arr(b)) = \infty > arr(b) \cdot d = C_{v_{n+1},dt(b)}(arr(b))$. Also wird $C_{v_{n+1},dt(b)}$ nicht von $C_{v_{n+1},dt(a)}$ dominiert. Umgekehrt kann $C_{v_{n+1},dt(a)}$ nicht von $C_{v_{n+1},dt(b)}$ dominiert werden, da $dt(b) > dt(a)$. □

Es gibt insgesamt $|\{o, u\}^n| = 2^n$ Pfade von v_1 zu v_{n+1}. Für jeden dieser Pfade gibt es nach Lemma 5.6 ein eigenes Profil am Knoten v_{n+1}. Nach Theorem 5.7 gibt es für jedes dieser Profile mindestens einen Zeitpunkt t, zu dem das Profil von keinem anderen Profil dominiert wird. Es kann also keines dieser Profile gelöscht werden und wir erhalten am Knoten v_{n+1} exponentiell viele Profile.

5.4 Problem-Komplexität

In Abschnitt 5.3 haben wir gesehen, dass das *RouBIRT-DB*-Problem nicht polynomiell lösbar ist, da die Anzahl der Profile an einem Knoten exponentiell in der Anzahl Knoten groß werden kann. Das wirft die Frage auf, ob das *RouBIRT-DB*-Problem $\mathcal{NP}$-vollständig ist. Wir definieren dazu das folgende Entscheidungsproblem.

Definition 5.8 (Routing-Entscheidungsproblem mit Straßensperrungen und Reisekosten(RouBIRT-DB-Dec))**.**

Gegeben: *Eine Anfrage bestehend aus einem Graphen G (wie in Abschnitt 2.1 beschrieben), einem Startknoten s, einem Zielknoten z, einem Planungshorizont $\mathcal{H}$ sowie zwei Parametern limit und break.*

Frage: *Existiert eine Route von s nach z, die zum Zeitpunkt t_{dep} an s losfährt, spätestens zum Zeitpunkt t_{arr} an z ankommt und höchstens limit fährt?*

Wenn wir die Beschränkung durch Lenk- und Ruhezeiten weglassen, dann können wir RouBIRT-DB-Dec mit dem *RouBIRT*-Algorithmus in polynomieller Zeit lösen. Wenn wir die temporären Sperrungen der Kanten weglassen, können wir RouBIRT-DB-Dec mit folgendem Algorithmus in polynomieller Zeit lösen. Dabei steht Erreichbarkeit für Erreichbarkeit durch Fahren von höchstens *limit*.

- Suche die Menge P_s aller Parkplätze, die von s erreichbar sind.
- Suche die Menge P_z aller Parkplätze, von denen z erreichbar ist.
- Suche für jeden Parkplatz u die Menge P_u aller Parkplätze, die von u erreichbar sind.
- Konstruiere einen Overlay-Graph G' mit den Knoten s,z und allen Parkplätzen.
- Füge für jeden Parkplatz u eine Kante zu jedem Parkplatz $v \in P_u$ mit Gewicht $dist(u, v) + break$ in G' ein.
- Füge für jeden Parkplatz $u \in P_s$ eine Kante (s, u) mit Gewicht $dist(s, u) + break$ in G' ein.
- Füge für jeden Parkplatz $u \in P_z$ eine Kante (u, z) mit Gewicht $dist(u, z)$ in G' ein.
- Bestimme in G' den kürzesten Weg von s nach z.

Für einen $\mathcal{NP}$-Vollständigkeitsbeweis von RouBIRT-DB-Dec müssen wir also sowohl temporäre Sperrungen als auch Lenk- und Ruhezeiten berücksichtigen (falls $\mathcal{P} \neq \mathcal{NP}$). Ob RouBIRT-DB-Dec tatsächlich $\mathcal{NP}$-vollständig ist, werden wir hier nicht näher untersuchen.

6. Versuchsaufbau

Um in den folgenden Kapiteln die beiden Algorithmen *BD-RouBIRT* und *RouBIRT-DB* zu evaluieren, geben wir in diesem Kapitel einige Informationen zur verwendeten Maschine, dem Graphen, den Queries sowie den Kostenparametern. Dabei stimmen das Straßennetzwerk und die Queries mit denen überein, die in [Brä18], [Wag19] und [KSWZ20] verwendet wurden.

6.1 Hardware und Kompiler

Für unsere Experimente verwenden wir eine virtuelle Maschine mit Ubuntu 16.04 als Betriebssystem. Zum Kompilieren verwenden wir *g++* Version 5.5.0. Unsere VM verfügt über vier Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz Prozessoren mit einer Turbo-Taktfrequenz von 3.3GHz und jeweils vier CPU-Kernen und 30MB Cache. Die Größe des RAM beträgt 64GB und es sind ca. 300GB Swap-Space verfügbar.

6.2 Straßennetzwerk

Der verwendete Graph enthält das Straßennetzwerk der Länder Deutschland, Frankreich, Italien, Liechtenstein, Luxemburg, Österreich und Schweiz. Der Graph besteht aus 30.069.809 Knoten, von denen 15.315 Parkplätze sind, und 47.635.384 Kanten. Durch das Entfernen von Knoten ohne ein- und ausgehende Kanten reduziert sich die Anzahl der Knoten auf 21.903.924. Der Graph enthält Fahrverbote für Sattelzüge mit einem Gesamtgewicht von 40 Tonnen. Diese sind in Tabelle 6.1 dargestellt. Zusätzlich gilt in Italien im Juli ein Samstagsfahrverbot von 8.00 - 16.00 Uhr.

Für die Kostenparameter verwenden wir das in [KSWZ20] als *default* festgelegte Kostenset. Die Parkplätze sind dabei je nach Anzahl der dort verfügbaren Stellplätze in 5 Kategorien unterteilt. Je mehr Stellplätze verfügbar sind, desto höher ist das Rating des Parkplatzes. Die Kostenparameter sind in Tabelle 6.2 aufgeführt.

Wie bereits in Abschnitt 2.1 erwähnt, erhalten Nicht-Parkplätze das Rating 0 und das Fahren kostet so viel wie das Warten auf einem Nicht-Parkplatz.

Tabelle 6.1: Überblick der Wochenend- und Nachtfahrverbote für Sattelzüge mit 40 Tonnen Gesamtgewicht in 2018 (Quelle: [Brä18]).

Land	Samstagsfahrverbot	Sonntagsfahrverbot	Nachtfahrverbot
Österreich[1]	15.00 – 24.00	00.00 – 22.00	22.00 – 05.00
Frankreich[2]	22.00 – 24.00	00.00 – 22.00	–
Deutschland[3]	–	00.00 – 22.00	–
Italien[4]	8.00 - 16.00 (Juli)	07.00 – 22.00 (Jun. – Sep.) 09.00 – 22.00 (Okt. – Mai)	–
Liechtenstein[5]	–	00.00 – 24.00	22.00 – 05.00
Luxemburg[6]	21.30 – 24.00 oder[7] 23.30 – 24.00	00.00 – 21.45	–
Schweiz[8]	–	00.00 – 24.00	22.00 – 05.00

[1] Österreichische Straßenverkehrsordnung: StVO §42(1)+(6), 12. Juli 2018
[2] Amtsblatt der Republik Frankreich: Journal officiel de la République française: n°0302 du 28 décembre 2017, texte n°120
[3] Deutsche Straßenverkehrsordnung: StVO §30(3), 06. Oktober 2017
[4] Ministerium für Infrastruktur und Verkehr Italien: Decreto Ministeriale numero 571 del 19-12-2017, Art. 1, 1a)+b)
[5] Liechtensteinische Straßenverkehrsordnung: VRV Art. 89 1)+2), 01. Juli 2018
[6] Luxemburgische Straßenverkehrsordnung: Réglement grand-ducal modifié du 19 juillet 1997 relatif aux limitations de la circulation des poids lourds pendant les dimanches et jours fériés Art. 1
[7] 21.30 – 24.00 Richtung Frankreich oder 23.30 – 24.00 Richtung Deutschland
[8] Schweizer Straßenverkehrsordnung: SVG Art. 2(2), 01. Januar 2018

Tabelle 6.2: Übersicht über das verwendete Kostenset

Rating	0	1	2	3	4	5
Kostenwert	14	7	6	5	4	3

6.3 Queries

Wir verwenden vier verschiedene Mengen von Queries, die wir von 1 bis 4 nummerieren. Diese Querymengen bezeichnen wir im Folgenden auch als Testmengen. Jede Testmenge hat ihren eigenen Start des Planungshorizontes. Diese frühesten Abfahrtszeiten sind in Tabelle 6.3 dargestellt.

Die Start- und Zielpunkte der ersten drei Testmengen sind identisch. Sie unterscheiden sich lediglich durch den Start des Planungshorizonts. Wir verwenden hier 100 verschiedene Startpunkte, die nördlich der Alpen zufällig gewählt sind. Zu jedem dieser Startpunkte gibt es jeweils 18 verschiedene Zielpunkte, die alle einen anderen Dijkstra-Rank haben. Ein Zielpunkt hat dabei den Dijkstra-Rank x, wenn er bei Dijkstra's Algorithmus als x-ter Knoten aus der Queue genommen wird. Wir verwenden dabei Dijkstra-Ranks 2^i

Tabelle 6.3: Früheste Abfahrtszeit für jede Testmenge. Diese sind angelehnt an [Brä18], die Planungshorizonte beginnen jedoch eine Stunde später. (Quelle: [Wag19])

Testmenge	Startzeit
1	Mo, 2. Juli 2018, 06.00
2	Fr, 6. Juli 2018, 06.00
3	Fr, 6. Juli 2018, 18.00
4	Mo, 2. Juli 2018, 18.00

Abbildung 6.1: Alle Start- und Zielknotenpaare in der Testmenge 4 befinden sich in den Gebieten A und B, jeweils einer der beiden Knoten in einem Gebiet. A erstreckt sich von 49° N 4° E an der linken oberen Ecke bis zu 47° N 18° E in der rechten unteren Ecke. Die nordwestliche Spitze des Gebiets B liegt bei 46° N 4° E, das südöstliche Ende ist an 42° N 18° E (Quelle: [Brä18]).

mit $i \in \{7, 8, ..., 24\}$. Für die Testmengen 1-3 ergeben sich damit insgesamt jeweils 1800 Queries.
Testmenge 4 enthält insgesamt 200 Queries, die alle unterschiedliche Startpunkte besitzen. Die Bereiche, aus denen die Start- und Zielpunkte stammen, sind in Abbildung 6.1 dargestellt. Der Bereich A enthält dabei die Startpunkte und Bereich B die Zielpunkte.

7. Evaluation des beschleunigten Basisalgorithmus

In diesem Kapitel evaluieren wir den *BD-RouBIRT*-Algorithmus quantitativ.

7.1 Planungshorizont und Strategien

Für die quantitative Evaluation des *BD-RouBIRT*-Algorithmus verwenden wir vier verschiedene Längen des Planungshorizonts: 12h, 24h, 36h und 48h. Längere Planungshorizonte werden wir nicht evaluieren, da für längere Planungshorizonte der RAM der VM zu klein ist und es zu Memory-Swapping kommt. Dadurch wird die Laufzeit des Algorithmus verfälscht und Vergleiche sind dann nicht mehr möglich.
In Abschnitt 4 haben wir noch die Frage offen gelassen, nach welcher Strategie wir entscheiden, ob wir als nächstes einen Knoten der Vorwärts- oder der Rückwärtssuche explorieren. Dazu verwenden wir die Strategie *ratio_x*. Wir explorieren immer erst eine bestimmte Anzahl x von Knoten in der Vorwärtssuche, bevor wir einen Knoten in der Rückwärtssuche explorieren. Dabei verwenden wir für x die Werte 1, 2, 5 und 10. Falls nichts näheres angegeben ist, verwenden wir die alternierende Strategie *ratio_1*, um zu entscheiden, ob wir einen Knoten der Vorwärts- oder Rückwärtssuche explorieren. Dies ist der Fall, wenn wir die Rückwärtssuche limitieren. Hier ist es dann am besten, wenn wir die Profile der Rückwärtssuche möglichst schnell berechnen, um früher prunen zu können.
Der Grund für die Notwendigkeit der im Folgenden beschriebenen Strategien ist der Folgende: Bei einer herkömmlichen bidirektionalen Suche in Dijkstra's Algorithmus bringt die Rückwärtssuche einen direkten Gewinn für die Berechnung des Ergebnisses. Es gibt ein Abbruchkriterium, wann die Suche beendet werden kann. In unserem Fall ist eine solche bidirektionale Suche nicht möglich, was in [Wag19] beschrieben ist. Es kann dann dazu kommen, dass optimale Pfade verloren gehen und sich somit das Ergebnis verändert. Daher verwenden wir eine andere Form der bidirektionalen Suche: die Rückwärtssuche trägt nicht direkt zur Berechnung dabei, sondern hilft nur dabei, den Suchraum der Vorwärtssuche einzuschränken, indem sie stärkeres Pruning ermöglicht. Daher sind die folgenden Strategien nötig, um jeweils eine andere Dosierung der Rückwärtssuche zu erreichen und herauszufinden, wann sich die beste Laufzeit ergibt.
Die Rückwärtssuche ist breitet sich kreisförmig aus. Dadurch werden im Laufe des Algorithmus immer mehr Knoten in der Rückwärtssuche exploriert, die für die Vorwärtssuche nur wenig Gewinn bringen. Diese Knoten liegen vom Start aus gesehen hinter dem Ziel und bringen lediglich ein verringern des Zeitpunktes t_{bw}, ab dem die Rückwärtsprofile final sind.

Im Lauf der Suche müssen in der Rückwärtssuche immer mehr Knoten exploriert werden, um t_{bw} um eine Zeiteinheit zu verringern. Daher limitieren wir die Rückwärtssuche nach den folgenden drei Strategien.

Harte Limitierung

Wir definieren eine Strategie *hardlimit_x*, bei der wir die Rückwärtssuche beenden, sobald in der Rückwärtssuche x Knoten exploriert wurden. Dabei verwenden wir für x die Parameter 100.000, 500.000, 1.000.000, 3.000.000, 5.000.000 und 7.000.000.

Weiche Limitierung

Wir definieren eine Strategie *softlimit_x*, bei der wir die Rückwärtssuche beenden, sobald mehr als x Knoten exploriert werden müssen, um t_{bw} um 1000 Zeiteinheiten zu verbessern. In unserer Implementierung entspricht eine Zeiteinheit einer Sekunde. Wir verwenden dabei für x die Werte 10.000, 100.000 und 1.000.000 .

Startdistanz-Limitierung

Wir definieren eine Strategie *potlimit*, bei der wir die Rückwärtssuche beenden, sobald $t_{max} - t_{bw} > dist_z(s)$ ist. Dabei ist t_{max} das Ende des Planungshorizonts und t_{bw} der Zeitpunkt, ab dem die Rückwärtsprofile aller Knoten final sind. Die Bezeichnung *potlimit* steht für *potential limit* und kommt daher, dass wir in unserer Implementierung $dist_z$ als Potentialfunktion für A* verwenden.

Falls keine dieser drei Strategien angegeben ist, läuft die Rückwärtssuche implizit bis zum Ende des Algorithmus.

Um schnelle Queries nicht zu verlangsamen, ist es sinnvoll, die Rückwärtssuche nicht bereits zu Beginn des Algorithmus zu starten. Daher verwenden wir die folgenden Strategien, um zu entscheiden, wann wir die Rückwärtssuche starten.

Key-Start

Für triviale Queries, bei denen der kürzeste Pfad auch mit Sperrungen gültig ist, sucht unser Algorithmus aufgrund der A*-Potentiale nur entlang des kürzesten Wegs. Für den Key k der Vorwärts-Queue Q_{fw} gilt dann bis zum Erreichen des Ziel-Knotens stets $k = t_{min} + dist_z(s)$. Für diese Queries bringt die Rückwärtssuche keinen Gewinn, da wir die Nachbarknoten der Knoten auf dem kürzesten Weg bereits mit dem Target Pruning aus Abschnitt 3.3.1 prunen können. Wenn k vor Erreichen des Ziels größer als $t_{min} + dist_z(s)$ wird, dann ist der kürzeste Weg unter der Berücksichtigung von Sperrungen nicht gültig und der Algorithmus muss einen oder mehrere alternative Wege suchen, was die Laufzeit erhöht. Daher definieren eine Strategie *keystart*, bei der wir die Rückwärtssuche starten, falls $k > t_{min} + dist_z(s)$ ist.

Distanz-Run

Eine weitere Möglichkeit zu erkennen, dass das Umfahren von Sperrungen nötig ist, ist das Beobachten der Potentiale der Knoten, die aus der Vorwärts-Queue entnommen werden. Bei trivialen Queries sind diese Potentiale bis zum Erreichen des Ziels monoton fallend. Wenn der kürzeste Weg eine Sperrung beinhaltet, dann ist das nicht der Fall. Daher definieren wir eine Strategie *potrun_x*, bei der wir die Rückwärtssuche starten, sobald das erste Mal x aufeinanderfolgende Potentiale der Knoten aus der Vorwärts-Queue monoton steigend sind. Wir verwenden dabei für x die Werte $4, 6$ und 8.

Tabelle 7.1: Übersicht über alle Strategien und die jeweils verwendeten Parameter

Strategie	Parameter
ratio_x	1, 2, 5, 10
hardlimit_x	100k, 500k, 1M, 3M, 5M, 7M
softlimit_x	1k, 1k, 1M
potlimit	-
keystart	-
potrun_x	4, 6, 8

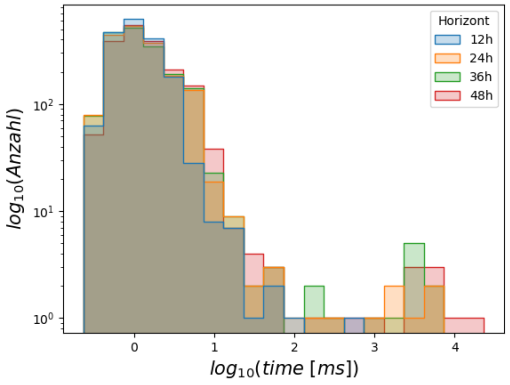
Falls keine dieser beiden Strategien angegeben ist, startet die Rückwärtssuche implizit zu Beginn des Algorithmus.
In Tabelle 7.1 sind alle aufgeführten Strategien zusammen mit den verwendeten Parametern aufgelistet.

7.2 Laufzeit Basisalgorithmus

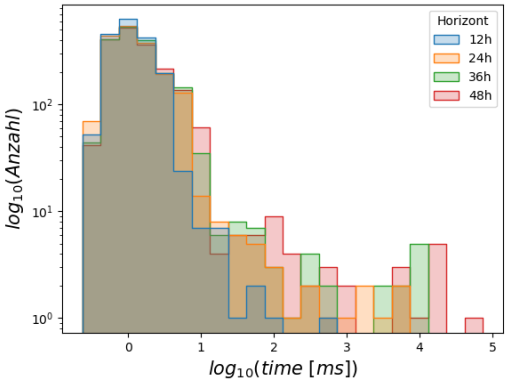
Das Ziel der Rückwärtssuche ist es, die Queries zu beschleunigen, bei denen Sperrungen umfahren werden müssen und es somit zu mehreren Ergebnispfaden kommt. Diese Queries führen bei großer Entfernung zum Ziel (Dijkstra-Rank 23 oder 24) zu hohen Laufzeiten von 1s bis teilweise fast 2min. Die Rückwärtssuche sollte dabei schnelle Queries nicht wesentlich verlangsamen. Daher ist es sinnvoll, die Strategien auf Queries mit niedriger und hoher Laufzeit getrennt zu evaluieren.
Um die Queries sinnvoll aufzuteilen, betrachten wir zunächst die Laufzeitverteilungen der einzelnen Testmengen auf den vier Horizonten 12h, 24h, 36h und 48h. Diese sind in Abbildung 7.1 aufgeführt.

Bei den Testmengen 1 & 2 haben fast alle Queries eine Laufzeit von 0.1-10ms. Das gilt für alle Horizontlängen. Queries mit einer Laufzeit von 10ms oder mehr gibt es fast nicht. Bei Testmenge 3 sieht die Verteilung ähnlich aus: Es gibt sehr viele Queries mit Laufzeit 0.1-10ms. Der Unterschied zu Testmenge 1 & 2 ist die höhere Anzahl Queries mit Laufzeit 100ms und höher. Es gibt hier auch ein paar Queries mit 3s und mehr Laufzeit. Bei Testmenge 4 sieht die Verteilung ganz anders aus: Abgesehen von 12h Horizont lassen sich jeweils zwei Maxima erkennen. Das erste liegt bei $3,16$ms und das zweite bei 1000ms. Das Minimum dazwischen liegt bei 10ms bzw. $31,6$ms.
Aufgrund dieser Beobachtungen können wir die Testmengen anhand eines Schwellwerts aufteilen. Für die Testmengen 1-3 wählen wir dazu als Schwellwert 10ms, bei Testmenge 4 $31,6$ms. Wir nummerieren die Untermengen von Testmenge i mit Untermenge i.1 (Laufzeit kleiner als Schwellwert) und Untermenge i.2 (Laufzeit größer als Schwellwert).

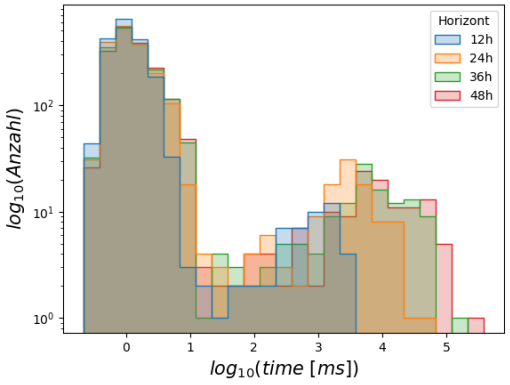
Abbildung 7.1: Anzahl der Queries mit Laufzeit *time*.



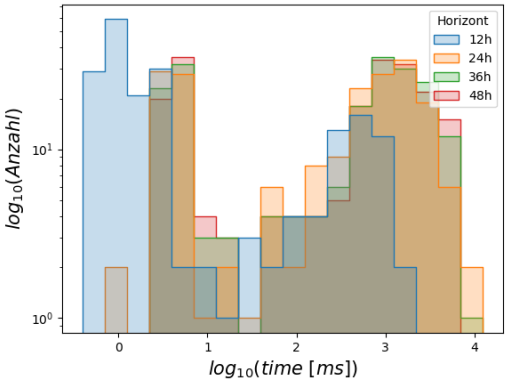
(a) Testmenge 1



(b) Testmenge 2



(c) Testmenge 3



(d) Testmenge 4

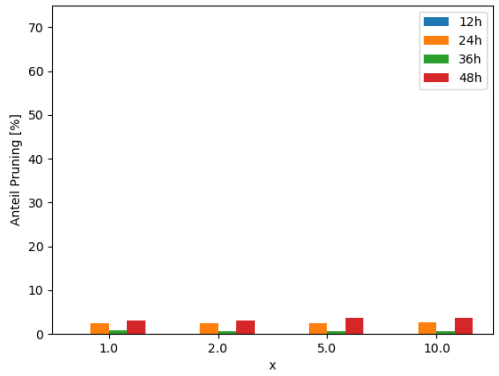
7.3 Vollständige Rückwärtssuche

In diesem Abschnitt betrachten wir kurz die Laufzeit der Vorwärtssuche, wenn vorher die Rückwärtssuche vollständig durchgelaufen ist. Das bedeutet, dass für alle Knoten das zugehörige Rückwärtsprofil zu allen Zeitpunkten final ist. Wir beschränken uns dabei auf eine ausgewählte Menge von 10 Queries aus Testmenge 3. Die Laufzeiten dieser Queries liegen bei *RouBIRT* zwischen 21s und 55s. Mit einer vollständigen Rückwärtssuche reduziert sich die Laufzeit der Vorwärtssuche auf 1.5s - 12s. Dadurch ergibt sich eine durchschnittliche Beschleunigung der Vorwärtssuche um den Faktor 5.7. Insgesamt laufen diese Queries dann mehrere Minuten, weshalb eine vollständige Rückwärtssuche nicht praktikabel ist.

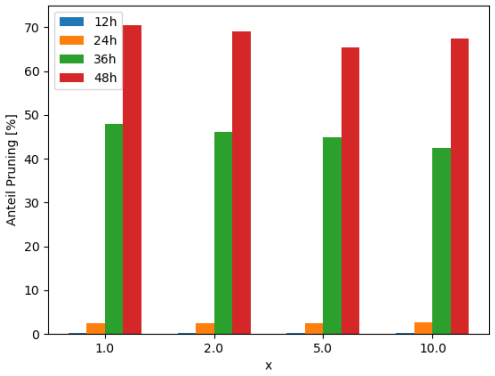
7.4 Target Pruning

Bevor wir auf die Laufzeiten der einzelnen Strategien eingehen, wollen wir zunächst die Wirkung der Rückwärtssuche im Bezug auf das Pruning betrachten. Wir beschränken uns in diesem Abschnitt immer auf die Strategien *ratio_x*, da sich für die restlichen Strategien ähnliche Werte ergeben. Wir untersuchen zunächst, bei wie vielen der Knoten, die in der Vorwärtssuche exploriert werden, wir Segmente prunen können. Dazu sind in Abbildung 7.2 die Anteile der Knoten aufgelistet, bei denen wir ein oderer mehrere Segmente prunen können.

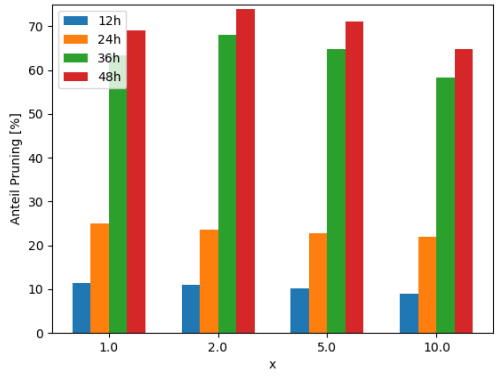
Abbildung 7.2: Durchschnittlicher Anteil der Knoten, bei denen ein oder mehrere Segmente geprunt werden konnten, an der Gesamtzahl der explorierten Knoten in der Vorwärtssuche mit Strategie *ratio_x*. In jedem Diagramm ist dabei der Anteil für eine Testmenge auf vier verschiedenen Horizontlängen dargestellt.



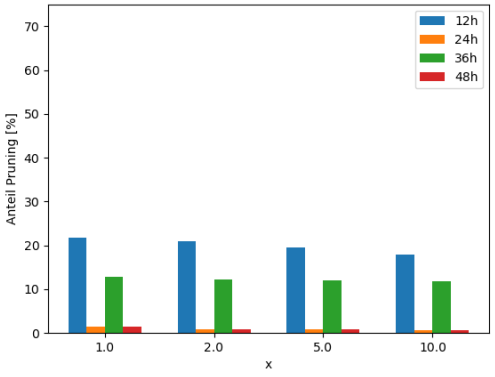
(a) Testmenge 1



(b) Testmenge 2



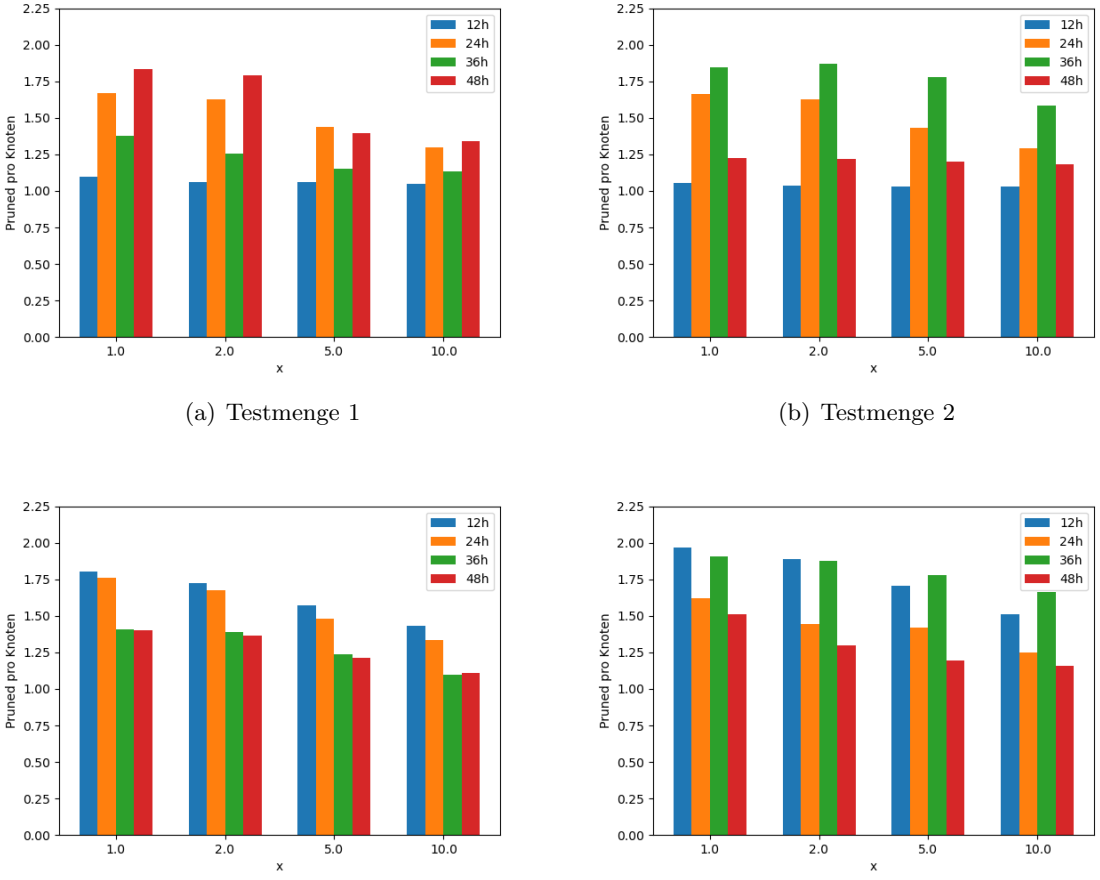
(c) Testmenge 3



(d) Testmenge 4

Bei Testmenge 1 ist der Anteil mit unter 4% nur sehr gering. Bei Testmenge 2 können wir bei 36h Horizont bei 40-50% und bei 48h Horizont bei 60-70% der Knoten Segmente prunen. Bei Testmenge 3 zeigt sich ein ähnliches Bild. Der Unterschied besteht darin, dass die Anteile bei 24h und 36h Horizont höher sind. Bei Testmenge 4 können wir bei 12h Horizont am meisten prunen (ca. 20%), bei 36h Horizont noch etwa 12% und bei 24h bzw. 48h Horizont ist der Anteil nur noch sehr gering. Diese Durchschnittswerte sind bei allen Testmengen für die Untermengen ähnlich, weshalb wir uns hier auf den Durchschnitt der gesamten Testmenge beschränken.
Um zu untersuchen, wie effektiv das Pruning an den Knoten jeweils ist, sind in Abbildung 7.3 die durchschnittlichen Anzahlen der Segmente, die wir prunen können, dargestellt. Dabei ignorieren wir Knoten, bei denen kein Segment geprunt werden kann.

Abbildung 7.3: Durchschnittliche Anzahl der Segmente, die pro Knoten geprunt werden konnten, mit Strategie *ratio_x*. In jedem Diagramm ist dabei die Anzahl für eine Testmenge auf vier verschiedenen Horizontlängen dargestellt.



(a) Testmenge 1

(b) Testmenge 2

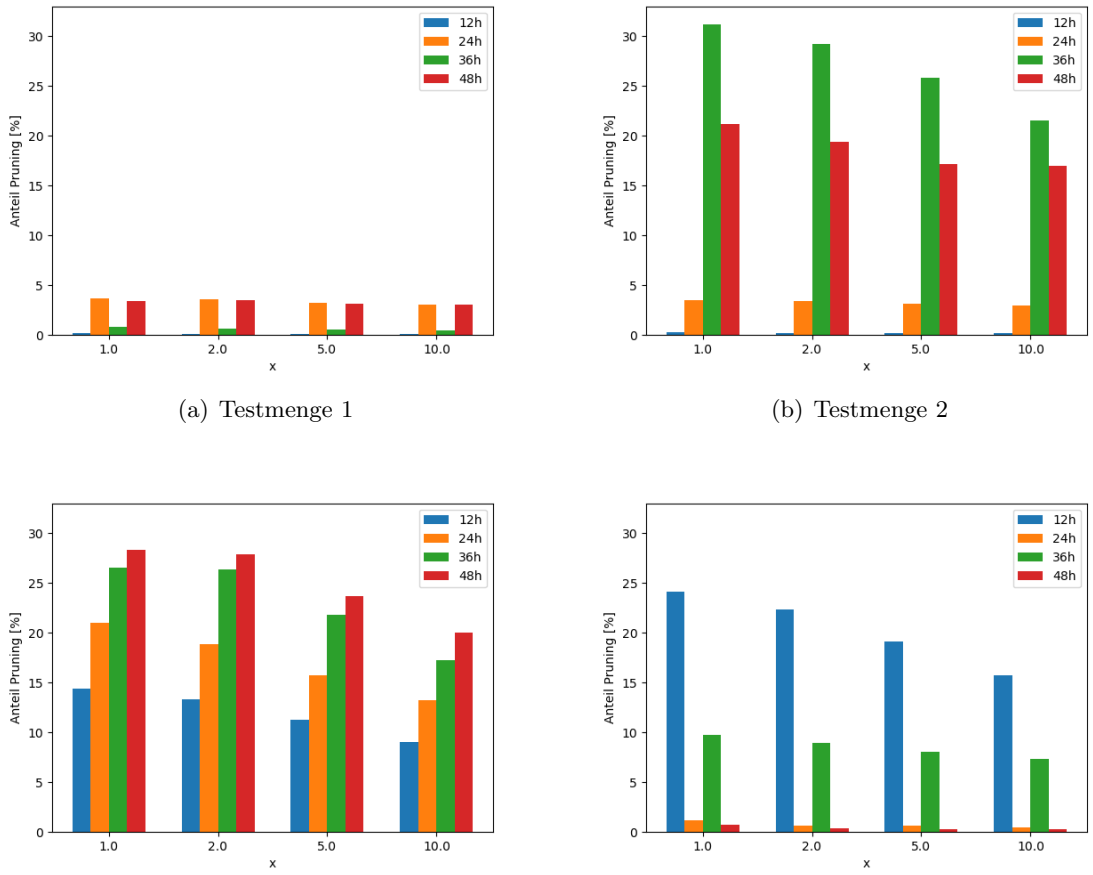
(c) Testmenge 3

(d) Testmenge 4

Bei allen Testmengen liegt das Maximum bei 1.8 bis 1.9 Segmenten pro Knoten, die Horizonte beim Auftreten des Maximum sind jedoch unterschiedlich. Bei Testmenge 1 liegt das Maximum bei 48h Horizont, bei Testmenge 3 können wir bei 48h Horizont am wenigsten Segmente prunen. Bei Testmenge 3 & 4 liegt das Maximum jeweils bei 12h Horizont, bei Testmenge 2 liegt das Maximum bei 36h Horizont. Insgesamt können wir aber durchschnittlich nie mehr als zwei Segmente pro Knoten prunen. Zudem ist zu sehen, dass die Zahl der Segmente, die pro Knoten geprunt werden können, mit steigendem x fällt. Das bedeutet, dass bei weniger explorierten Knoten in der Rückwärtssuche die Anzahl der

geprunten Segmente pro Knoten fällt. Diese Aussage trifft auch auf die anderen Strategien zu.
Um diese absoluten Zahlen der geprunten Segmente in Relation zur Gesamtzahl der Segmente zu setzen, ist in Abbildung 7.4 jeweils der Anteil der Segmente dargestellt, die geprunt werden konnten.

Abbildung 7.4: Durchschnittlicher Anteil der Segmente, die geprunt wurden, an der Gesamtzahl der Segmente mit Strategie *ratio_x*. In jedem Diagramm ist dabei der Anteil für eine Testmenge auf vier verschiedenen Horizontlängen dargestellt.



(a) Testmenge 1

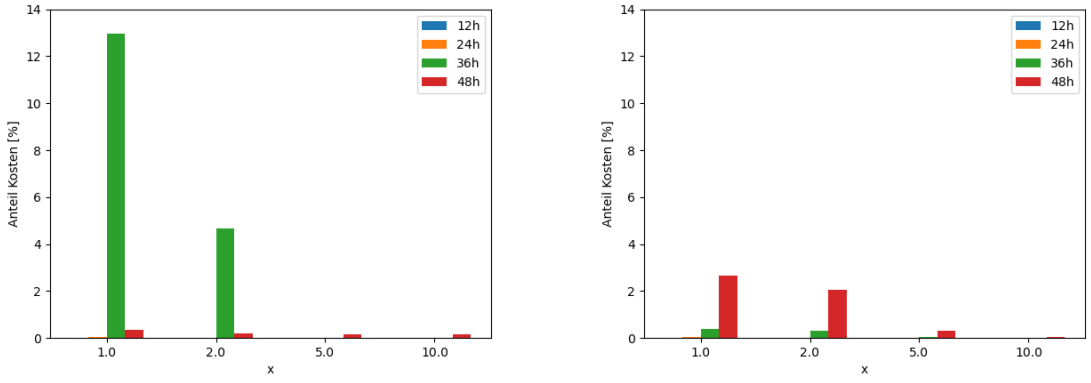
(b) Testmenge 2

(c) Testmenge 3

(d) Testmenge 4

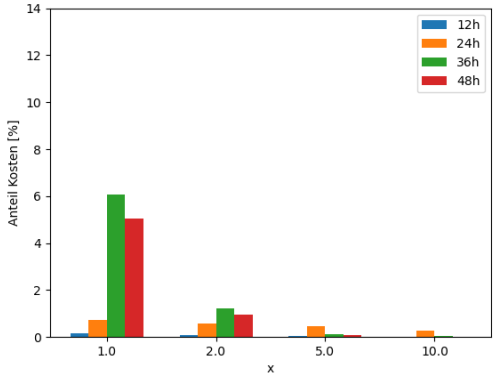
Für die Gesamtzahl der geprunten Segmente p und die Anzahl der am Ende der Suche vorhandenen Segmente s ergibt sich der Anteil dabei durch $\frac{p}{s+p}$. Bei Testmenge 1 liegen die Anteile immer unter 5%. Bei Testmenge 2 sind die Anteile bei 36h und 48h Horizont deutlich höher: Die Anteile liegen hier bei ca. 30% bzw. 20%. Bei Testmenge 3 ist der Anteil umso höher, je länger der Horizont ist. Hier bewegen sich die Anteile zwischen knapp 15% und ca. 28%. Bei Testmenge 4 sind die Anteile für 24h und 48h Horizont sehr gering, bei 36h Horizont liegen sie bei ca. 10%. Bei 12h Horizont sind sie mit ca. 24% am höchsten.
In Theorem 4.1 haben wir zwei verschiedene Möglichkeiten für das Target-Pruning mithilfe der Rückwärts-Profile eingeführt: Nach frühestmöglicher Ankunft nach Ende des Horizonts oder nach Mindestkostenwert und frühester Ankunft am Ziel. Wir bezeichnen die zweite Möglichkeit als Pruning nach Kosten. Im Folgenden untersuchen wir, welchen Anteil das Pruning nach Kosten am Pruning hat. Dazu sind in Abbildung 7.5 für die vier Testmengen die Anteile der Segmente, die nach Kosten geprunt wurden, an der Gesamtzahl der geprunten Segmente aufgeführt.

Abbildung 7.5: Durchschnittlicher Anteil der Segmente, die nach Kosten geprunt wurden, an der Gesamtzahl der Segmente mit Strategie *ratio_x*. In jedem Diagramm ist dabei der Anteil für eine Testmenge auf vier verschiedenen Horizontlängen dargestellt.

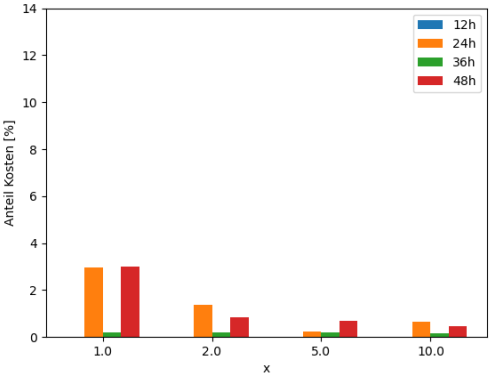


(a) Testmenge 1

(b) Testmenge 2



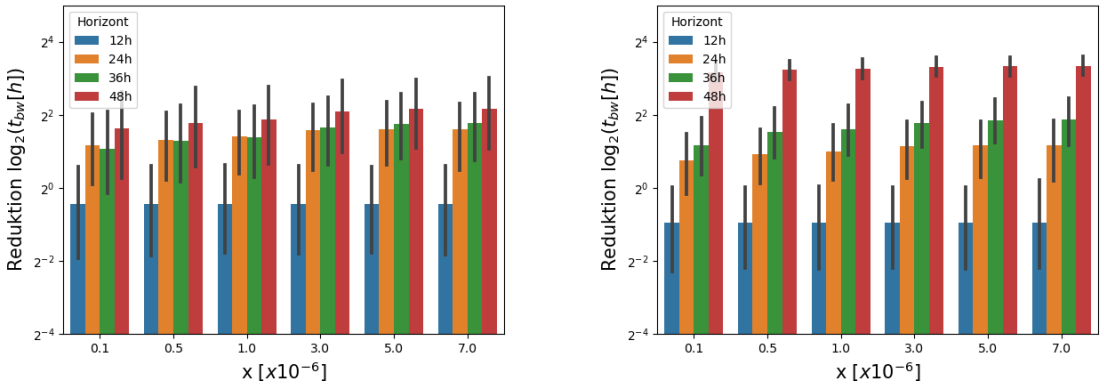
(c) Testmenge 3



(d) Testmenge 4

Außer bei *ratio_1* ist der Anteil immer kleiner als 5%, meistens sogar unter 2%. Für die Strategien, die in den folgenden Abschnitten noch evaluiert werden, sind die Anteile teilweise ähnlich, meistens aber sogar noch geringer. Das bedeutet, dass das Pruning nach Kosten für das Pruning keine große Rolle spielt. Wie viele Segmente im Laufe der Suche geprunt werden können, hängt also im Wesentlichen davon ab, wie stark die Rückwärtssuche den Zeitpunkt t_{bw}, ab dem die Rückwärtsprofile final sind, reduzieren kann. Daher werden wir für die Evaluation von *ratio_x* und der noch folgenden Strategien jeweils die Anzahl der Zeiteinheiten, die die Rückwärtssuche t_{bw} reduzieren kann, als Erklärung für die auftretenden Laufzeiten verwenden.
Bevor wir die Laufzeiten evaluieren, werden wir daher anhand der Strategien *hardlimit_x* untersuchen, wie stark die Rückwärtssuche t_{bw} reduzieren kann. Dazu betrachten wir von den vier Testmengen jeweils die zweite Untermenge mit den langsameren Queries, da es bei der anderen Untermenge häufig dazu kommt, dass die Suche beendet wird, bevor die maximale Anzahl an Knoten in der Rückwärtssuche exploriert wurde. In Abbildung 7.6 sind für die vier Untermengen die Anzahl der Zeiteinheiten dargestellt, um die die Rückwärtssuche t_{bw} reduzieren kann. Eine Zeiteinheit entspricht dabei einer Sekunde und ein Tag hat 86400s.

Abbildung 7.6: Durchschnittliche Anzahl Stunden, die die Rückwärtssuche t_{bw} reduzieren konnte, mit Strategie *hardlimit_x*. In jedem Diagramm ist dabei die Anzahl für eine Testmenge auf vier verschiedenen Horizontlängen dargestellt. Die schwarzen vertikalen Striche stellen die Standardabweichung dar.



(a) Untermenge 1.2

(b) Untermenge 2.2

(c) Untermenge 3.2

(d) Untermenge 4.2

Für alle Untermengen ist ein sättigender Verlauf der Punkte erkennbar. Das kommt daher, dass die Suchfront der Rückwärtssuche im Lauf des Algorithmus immer größer wird und somit immer mehr Knoten für die gleiche Reduzierung von t_{bw} exploriert werden müssen. Bei Untermenge 1.2 ist die Reduzierung am geringsten, bei Untermenge 2.2 ist sie schon etwa doppelt so groß. Bei Untermenge 3.2 ist die Reduzierung mit Abstand am größten. Hier wird t_{bw} im Schnitt um fast einen Tag reduziert (bei 48h Horizont). Die Unterschiede zwischen den Untermengen kommen daher, dass es bei Untermenge 3.2 mehr Queries gibt, wo das Limit der maximalen Anzahl an explorierten Knoten in der Rückwärtssuche tatsächlich erreicht wird. Der sättigende Verlauf kommt bei den Untermengen 1.2 und 2.2 auch daher, dass sich die Anzahl der explorierten Knoten in der Rückwärtssuche mit steigendem x immer weniger steigt. Bei Untermenge 4.2 steigen die Werte am Anfang sehr stark, diese Steigung flacht aber sehr schnell ab. Die Reduzierung ist hier kleiner als bei den Untermengen 2.2 und 3.2. Mit diesen Erkenntnissen werden wir im folgenden Abschnitt die Laufzeiten der einzelnen Strategien evaluieren.

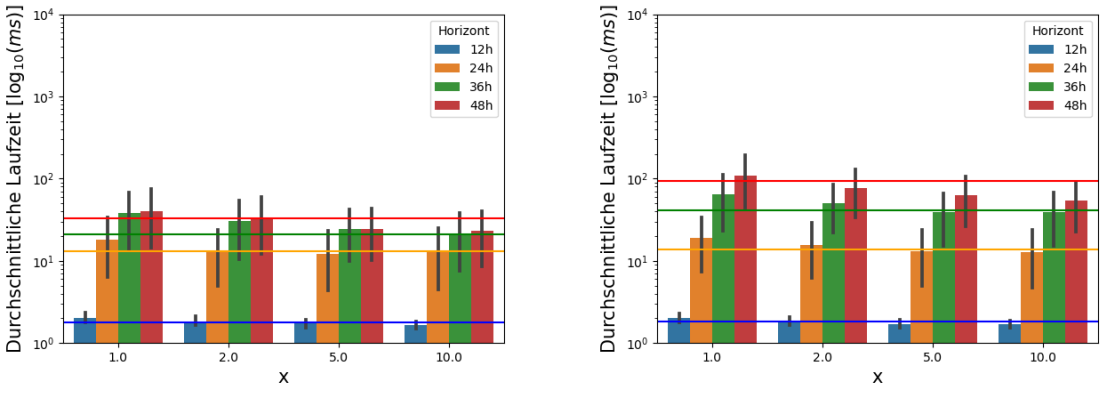
7.5 Laufzeiten der Strategien

In diesem Abschnitt werden wir die Laufzeiten der einzelnen Strategien evaluieren und für jede Strategie den Parameter x auswählen, für den die beste Laufzeit erzielt wird.

7.5.1 Keine Limitierung

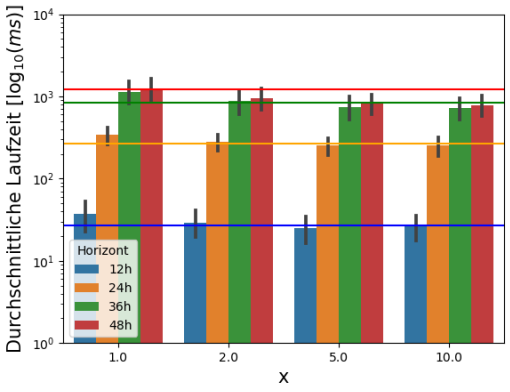
Für die Evaluation der Strategien *ratio_x* sind in Abbildung 7.7 die durchschnittlichen Laufzeiten auf den unterschiedlichen Testmengen dargestellt.

Abbildung 7.7: Durchschnittliche Laufzeit der Strategie *ratio_x*. In jedem Diagramm ist dabei die Laufzeit für eine Testmenge auf vier verschiedenen Horizontlängen dargestellt. Die durchgezogenen Linien stellen die durchschnittliche Laufzeit von *RouBIRT* dar. Die schwarzen vertikalen Striche stellen die Standardabweichung dar.

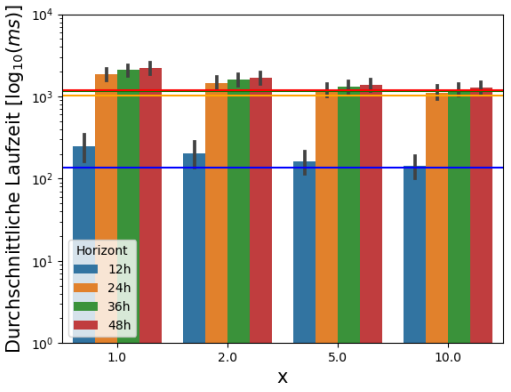


(a) Testmenge 1

(b) Testmenge 2



(c) Testmenge 3



(d) Testmenge 4

Für fast alle Horizontlängen lässt sich dabei eine Kurve erkennen, die bei *ratio_1* ihr Maximum hat und für steigendes x immer weiter abfällt. Zudem lässt sich vor allem bei 36h und 48h Horizontlänge eine Sättigung erkennen. Im Vergleich zur alten durchschnittlichen Laufzeit weisen die Testmengen 1-3 für die Horizontlänge 48h ab einem bestimmten Wert von x eine Verbesserung auf. Der Wert für x liegt für Testmenge 1 bei $x = 5$ und bei den Testmengen 2 & 3 bei $x = 2$. Bei Testmenge 4 gibt es keinen (getesteten) Parameter für x, für den eine Beschleunigung erreicht wird. Für $x = 1$ sind die Laufzeiten teilweise fast doppelt so lang, für $x = 10$ kommen sie sehr nah an die alte Laufzeit heran. Daraus folgt direkt, dass für eine Beschleunigung bei Testmenge 4 eine Limitierung der Rückwärtssuche nötig ist. Für die jeweiligen Untermengen sehen die Kurven sehr ähnlich aus, für die jeweils erste Untermenge der Testmengen 1-3 kann jedoch keine Beschleunigung erreicht werden. Bei *ratio_1* liegt die Laufzeit bei 2.2ms bis 2.8ms. Die alte Laufzeit liegt jeweils zwischen 1.7ms und 1.9ms (48h Horizont). Für *ratio_10* kommt die Laufzeit (ähnlich zu Testmenge

4) sehr nahe an die alte Laufzeit heran. Insgesamt wählen wir *ratio_10* als beste dieser Strategien aus.

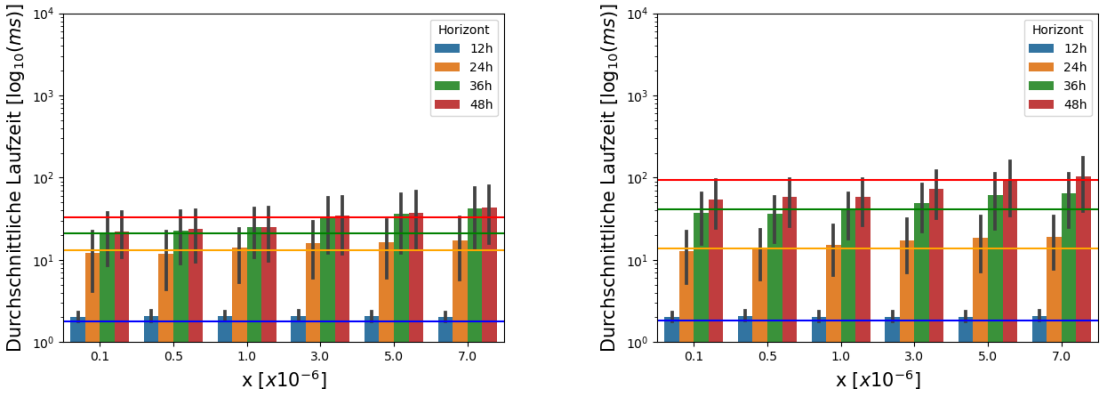
7.5.2 Limitierung

In diesem Abschnitt werden wir die Strategien *hardlimit_x*, *softlimit_x* und *potlimit* evaluieren.

Harte Limitierung

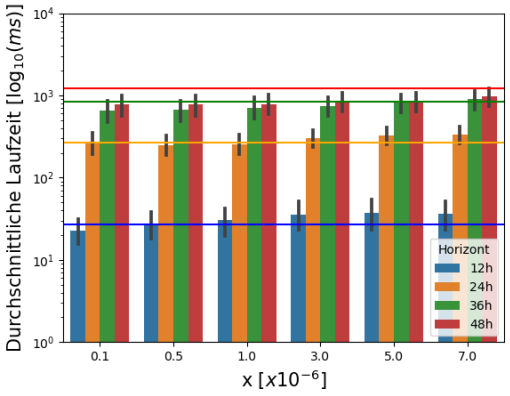
Für die Evaluation der Strategie *hardlimit_x* sind in Abbildung 7.8 die durchschnittlichen Laufzeiten für die einzelnen Testmengen und Horizontlängen dargestellt.

Abbildung 7.8: Durchschnittliche Laufzeit der Strategie *hardlimit_x*. In jedem Diagramm ist dabei die Laufzeit für eine Testmenge auf vier verschiedenen Horizontlängen dargestellt. Die durchgezogenen Linien stellen die durchschnittliche Laufzeit von *RouBIRT* dar. Die schwarzen vertikalen Striche stellen die Standardabweichung dar.

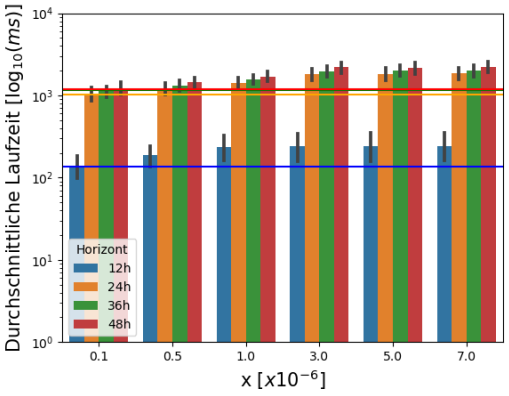


(a) Testmenge 1

(b) Testmenge 2



(c) Testmenge 3



(d) Testmenge 4

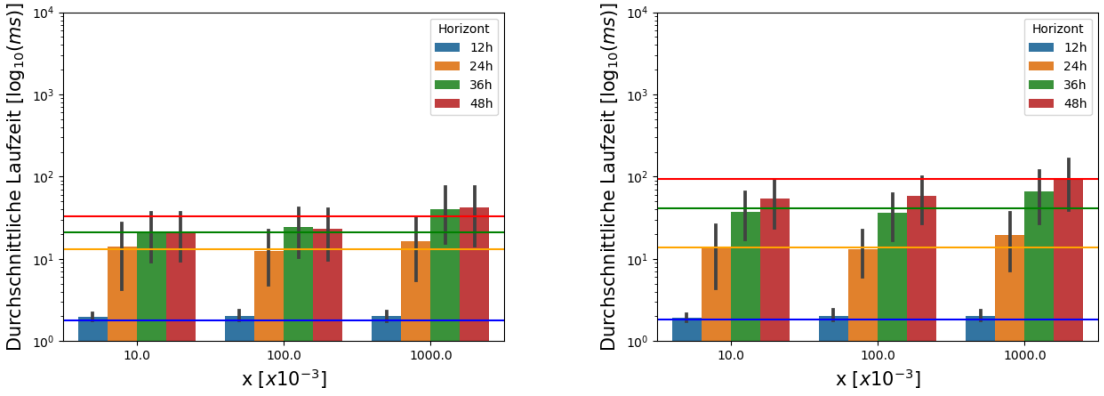
In jedem Schaubild lässt sich erkennen, dass mit wachsendem x die durchschnittliche Laufzeit steigt. Das Minimum ist dabei meist bei $x = 100.000$, sonst bei $x = 500.000$. Bei den Testmengen 1-3 wird für x bis 1.000.000 immer eine Beschleunigung erreicht (Horizontlänge 48h), während für $x = 7.000.000$ nur noch in Testmenge 3 eine Beschleunigung erreicht wird. Für die anderen Horizontlängen lässt sich ähnliches beobachten, das x, ab dem keine

Beschleunigung mehr erreicht wird, ist jedoch unterschiedlich. Für Testmenge 4 hat die erkennbare Kurve eine ähnliche Form, allerdings liegt hier für kein x eine Beschleunigung vor. Für $x = 100.000$ kommt die Laufzeit sehr nah an die alte Laufzeit heran, für $x = 7.000.000$ ist sie jedoch teilweise fast doppelt so groß. Für die jeweils zweite Untermenge ist bei allen Testmengen der Verlauf qualitativ jeweils gleich. Für die jeweils erste Untermenge der Testmengen 1-3 ist die Laufzeit für alle x nahezu konstant und es wird keine Beschleunigung erreicht. Die Laufzeiten bewegen sich bei 48h Horizont zwischen 2.2ms und 2.8ms, was weniger als zweimal so groß wie die jeweils alte Laufzeit ist. Daher können wir diesen Laufzeitverlust vernachlässigen. Für die Untermenge 4.1 gilt das abgesehen von 36h Horizont ebenso. Bei diesem Horizont liegt das Minimum bei $x = 500.000$ und danach steigen die Laufzeiten. Für $x \leq 1.000.000$ kann dabei eine Beschleunigung erreicht werden. Die Laufzeiten liegen zwischen 100ms und 140ms, die alte Laufzeit liegt bei ca. 130ms. Wir wählen *hardlimit_100k* als beste dieser Strategien aus, da hier unter Berücksichtigung aller Testmengen die Laufzeit am besten ist.

Weiche Limitierung

Für die Evaluation der Strategie *softlimit_x* sind in Abbildung 7.9 die durchschnittlichen Laufzeiten dargestellt.

Abbildung 7.9: Durchschnittliche Laufzeit der Strategie *softlimit_x*. In jedem Diagramm ist dabei die Laufzeit für eine Testmenge auf vier verschiedenen Horizontlängen dargestellt. Die durchgezogenen Linien stellen die durchschnittliche Laufzeit von *RouBIRT* dar. Die schwarzen vertikalen Striche stellen die Standardabweichung dar.



(a) Testmenge 1

(b) Testmenge 2

(c) Testmenge 3

(d) Testmenge 4

Wir sehen, dass bei den Testmengen 1-3 für $x = 10.000$ oder $x = 100.000$ die minimale Laufzeit erreicht wird. Das Maximum wird immer bei $x = 1.000.000$ erreicht. Eine wesentliche Beschleunigung erreichen wir hier nur für die Horizontlänge 48h bei $x = 10.000$ und $x = 100.000$. Für die anderen Horizontlängen ist entweder die Beschleunigung sehr gering oder die Laufzeit sogar größer. Für Testmenge 4 können wir für keine Horizontlänge eine Beschleunigung erreichen. Für $x = 10.000$ kommen die Laufzeiten sehr nah an die alten Laufzeiten heran, für $x = 1.000.000$ ist die Laufzeit jedoch fast doppelt so groß. Für die jeweils zweite Untermenge ist der Verlauf qualitativ jeweils gleich. Für die jeweils erste Untermenge sind die Verläufe ähnlich wie mit den Strategien *hardlimit_x*. Wir wählen *softlimit_10k* als beste dieser Strategien aus, da hier unter Berücksichtigung aller Testmengen die Laufzeit am besten ist.

Startdistanz-Limitierung

Für die Evaluation der Strategie *potlimit* sind die durchschnittlichen Laufzeiten in Tabelle 7.2 dargestellt.

Tabelle 7.2: Durchschnittliche Laufzeit der Strategie *potlimit* verglichen mit der Laufzeit von *RouBIRT* in ms. In jeder Tabelle ist dabei die Laufzeit für eine Testmenge auf vier verschiedenen Horizontlängen dargestellt.

(a) Testmenge 1

Horizontlänge	*RouBIRT*	*potlimit*
12h	1.8	2.0
24h	13.0	17.5
36h	21.0	38.2
48h	32.8	37.3

(b) Testmenge 2

Horizontlänge	*RouBIRT*	*potlimit*
12h	1.8	2.1
24h	13.9	18.1
36h	40.7	64.0
48h	94.5	94.6

(c) Testmenge 3

Horizontlänge	*RouBIRT*	*potlimit*
12h	26.7	37.3
24h	264.7	328.4
36h	835.9	952.0
48h	1227.6	941.8

(d) Testmenge 4

Horizontlänge	*RouBIRT*	*potlimit*
12h	136.9	237.5
24h	1024.6	1835.7
36h	1155.8	1916.5
48h	1186.3	2207.9

Bei den Testmengen 1 & 2 können wir jeweils keine Beschleunigung erreichen, die Laufzeit wird absolut betrachtet aber auch nicht wesentlich langsamer. Bei Testmenge 3 können wir bei 48h Horizontlänge eine Beschleunigung um einen Faktor von ca. 1.3 erreichen. Bei den anderen Horizontlängen ist die Laufzeit größer, relativ betrachtet aber höchstens um 0.25 höher. Bei Testmenge 4 können wir keine Verbesserung erzielen und die Laufzeiten werden teilweise fast doppelt so lang. Für die beiden Untermengen sind die Laufzeiten qualitativ (auch im Vergleich mit den alten Laufzeiten) jeweils gleich.
Um einen Vergleich zu den Strategien *hardlimit_x* herzustellen, sind in Tabelle 7.3 die durchschnittlichen Anzahlen der explorierten Knoten in der Rückwärtssuche dargestellt.

Tabelle 7.3: Übersicht über die Anzahl der explorierten Knoten in der Rückwärtssuche bei der Strategie *potlimit*.

(a) Testmenge 1

Horizontlänge	Anzahl [$x10^{-3}$]
12h	1.0
24h	9.4
36h	19.0
48h	17.5

(b) Testmenge 2

Horizontlänge	Anzahl [$x10^{-3}$]
12h	1.0
24h	10.1
36h	24.9
48h	23.1

(c) Testmenge 3

Horizontlänge	Anzahl [$x10^{-3}$]
12h	16.5
24h	116.0
36h	187.9
48h	58.8

(d) Testmenge 4

Horizontlänge	Anzahl [$x10^{-3}$]
12h	132.8
24h	953.6
36h	861.7
48h	930.8

Interessant ist dabei vor allem die vierte Testmenge: Hier werden (abgesehen von 12h Horizont) im Durchschnitt fast 1 Millionen Knoten in der Rückwärtssuche exploriert, was, wie wir bei den Strategien *hardlimit_x* bereits gesehen haben, zu viel ist, um eine Beschleunigung zu erreichen. Das erklärt die fast um Faktor 2 schlechteren Laufzeiten. Für die jeweils zweite Untermenge der Testmengen 1-3 gilt dasselbe wie für die Testmenge 4. Bei den jeweils ersten Untermengen sind die Anzahlen meist unter 1000, was aber trotzdem nicht zu einer Beschleunigung führt.
Die Strategie *potlimit* ist also auf jeden Fall schlechter als die Strategie *hardlimit_100k*, weshalb wir sie im Folgenden nicht weiter betrachten.

7.5.3 Verzögerter Start

In diesem Abschnitt werden wir die Strategien *potrun_x* und *keystart* evaluieren. Bei der Ausführung haben wir diese Strategien jeweils mit jeder den Strategien *hardlimit_x*, *ratio_x*, *soft_x* kombiniert. Die qualitative Lage der Punkte zueinander in den Diagrammen ist dabei dieselbe wie in den Diagrammen der jeweiligen Strategie aus den Abschnitten 7.5.1 und 7.5.2. Insbesondere ist damit der beste Parameter x für eine der Strategien *hardlimit_x*, *ratio_x* oder *soft_x* auch bei Kombination mit einer Strategie zum späteren Start der Rückwärtssuche der Parameter, der die beste durchschnittliche Laufzeit erzielt. Wir werden im Folgenden die Strategien *potrun_x* und *keystart* in Kombination mit *softlimit_10k* vergleichen, da sich in Kombination mit den anderen Strategien jeweils qualitativ gleiche Verläufe ergeben.

Distanz-Run

Zunächst wollen wir für die Strategien *potrun_x* untersuchen, bei wie vielen Queries die Rückwärtssuche überhaupt gestartet wird und zu welchem Zeitpunkt sie gestartet wird. Als Zeitpunkt ist dabei ein prozentualer Zeitpunkt gemeint. Wenn der *BD-RouBIRT*-Algorithmus die Rückwärtssuche nach a explorierten Knoten in der Vorwärtssuche startet und insgesamt b Knoten in der Vorwärtssuche exploriert werden, dann ist der prozentuale Zeitpunkt $\frac{a}{b}$. Da wir mit diesen Strategien die Rückwärtssuche im Idealfall nur bei Queries aus den jeweils zweiten Untermengen starten wollen, werden wir die Untermengen im Folgenden jeweils getrennt betrachten.

In Tabelle 7.4 sind für die Strategien *potrun_x* die Anteile der Queries, bei denen die Rückwärtssuche gestartet wird, aufgeführt.

Tabelle 7.4: Übersicht über den Start der Rückwärtssuche bei den Strategien *potrun_x*. Die Spalten 2-4 enthalten dabei die Anteile für den jeweiligen Parameter x in %.

(a) Untermenge 1.1

Horizont	4	6	8
12h	0.92	0.31	0.24
24h	1.14	0.45	0.45
36h	2.89	2.15	2.15
48h	3.00	2.38	2.38

(b) Untermenge 1.2

Horizont	4	6	8
12h	100.00	83.33	0
24h	100.00	77.42	29.03
36h	100.00	81.82	48.48
48h	97.06	82.35	52.94

(c) Untermenge 2.1

Horizont	4	6	8
12h	0.49	0.18	0.18
24h	0.69	0.40	0.40
36h	2.75	2.40	2.40
48h	6.24	5.84	5.61

(d) Untermenge 2.2

Horizont	4	6	8
12h	100.00	72.73	9.09
24h	91.67	68.75	37.50
36h	92.31	82.69	48.08
48h	92.45	92.45	86.79

(e) Untermenge 3.1

Horizont	4	6	8
12h	1.44	1.00	0.81
24h	3.12	2.70	2.58
36h	6.07	5.65	5.29
48h	6.07	5.65	5.29

(f) Untermenge 3.2

Horizont	4	6	8
12h	100.00	92.86	66.07
24h	97.67	95.35	80.62
36h	97.78	97.04	94.07
48h	97.79	97.06	94.12

(g) Untermenge 4.1

Horizont	4	6	8
12h	3.03	0	0
24h	10.00	5.00	5.00
36h	12.90	8.06	6.45
48h	14.52	11.29	9.68

(h) Untermenge 4.2

Horizont	4	6	8
12h	98.33	93.33	45.00
24h	100.00	100.00	84.06
36h	100.00	100.00	88.41
48h	100.00	100.00	90.58

Wir betrachten zunächst nur den Parameter $x = 4$. Für die Untermenge 1.1 stellen wir fest, dass die Rückwärtssuche nur bei wenigen Queries gestartet wird. Bei Untermenge 2.1 & 3.1 ist der Anteil der Queries, bei denen die Rückwärtssuche gestartet wird, für 48h Horizont höher im Vergleich zur Untermenge 1.1. Je länger der Horizont wird, bei desto mehr Queries wird die Rückwärtssuche gestartet. Bei Untermenge 4.1 wird bei fast 15% aller Queries die Rückwärtssuche gestartet, was wesentlich mehr ist als bei den Untermengen 1.1 - 3.1. Mit steigendem x nimmt bei allen ersten Untermengen die Anzahl der Queries, bei denen die Rückwärtssuche gestartet wird, ab. Die Anzahl nimmt dabei meist um weniger als die Hälfte ab. Für die jeweils zweiten Untermengen liegen die Anteile bei 48h Horizont immer bei ca. 90%. Bei den Untermengen 3.2 und 4.2 gilt das für alle Werte von x. Bei den Untermengen 1.2 und 2.2 fallen die Anteile auf ca. 50%. Um die Zeitpunkte, zu denen die Rückwärtssuche gestartet wird, zu untersuchen, sind in Tabelle 7.5 die Zeitpunkte aufgeführt, zu denen die Rückwärtssuche jeweils durchschnittlich gestartet wird.

Tabelle 7.5: Übersicht über den Zeitpunkt des Starts der Rückwärtssuche bei den Strategien *potrun_x*. Die Spalten 2-4 enthalten dabei die Zeitpunkte für den jeweiligen Parameter x in %. Dabei werden Queries, bei denen die Rückwärtssuche nicht gestartet wurde, vernachlässigt.

(a) Untermenge 1.1

Horizont	4	6	8
12h	16.56	79.95	87.15
24h	26.20	62.10	62.29
36h	50.88	77.35	77.46
48h	50.96	77.92	78.02

(b) Untermenge 1.2

Horizont	4	6	8
12h	4.42	28.82	0
24h	0.30	1.56	18.94
36h	0.21	1.48	30.58
48h	0.21	1.57	29.05

(c) Untermenge 2.1

Horizont	4	6	8
12h	32.27	63.82	65.32
24h	46.62	76.56	77.13
36h	70.00	81.49	81.67
48h	79.56	84.60	84.66

(d) Untermenge 2.2

Horizont	4	6	8
12h	4.93	31.08	99.49
24h	0.45	2.05	24.33
36h	0.26	1.41	16.32
48h	0.23	1.46	15.70

(e) Untermenge 3.1

Horizont	4	6	8
12h	14.27	28.88	25.21
24h	35.24	50.26	52.29
36h	56.28	67.00	67.62
48h	56.28	67.00	67.62

(f) Untermenge 3.2

Horizont	4	6	8
12h	0.19	1.48	42.37
24h	0.08	0.44	14.52
36h	0.05	0.29	11.26
48h	0.04	0.25	11.31

(g) Untermenge 4.1

Horizont	4	6	8
12h	13.77	0	0
24h	19.73	56.73	79.87
36h	0.28	0.45	0.53
48h	21.34	54.13	74.37

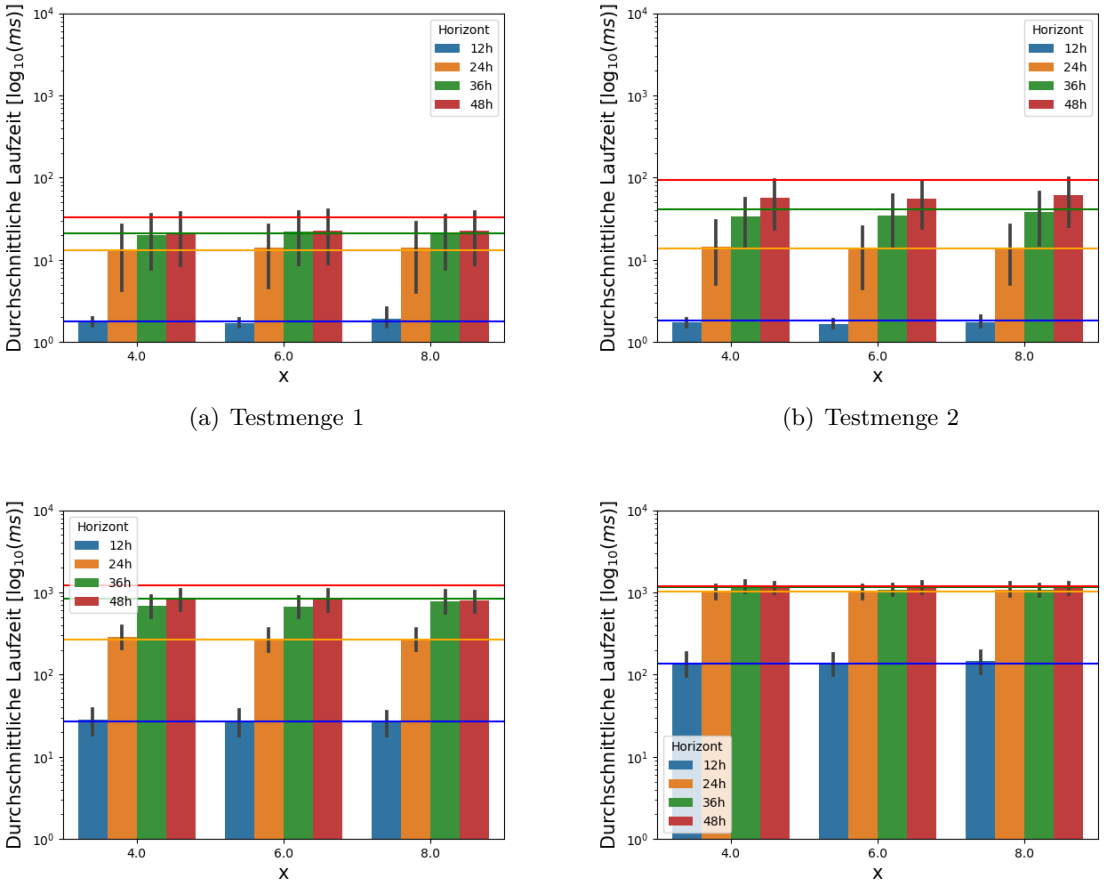
(h) Untermenge 4.2

Horizont	4	6	8
12h	0.34	2.35	41.13
24h	0.12	0.92	30.78
36h	0.13	1.04	35.34
48h	0.13	1.01	38.69

Bei Untermenge 1.1 liegen die Zeitpunkte zwischen 50% und 80% (48h Horizont). Bei Untermenge 2.1 sind die Zeitpunkte leicht höher und bei $x = 4$ sogar um ca. 30% höher. Bei Untermenge 3.1 sind die Zeitpunkte insgesamt kleiner als bei den Untermengen 1.1 und 2.1. Sie liegen bei ca. 67% ($x = 6, 8$). Bei Untermenge 4.1 sind die Zeitpunkte bei $x = 4$ deutlich niedriger. Bei $x = 8$ sind sie wieder im Bereich der Werte der der Untermengen 3.1,3.2 und 3.3. Auffällig sind hier die sehr kleinen Werte bei 36h Horizont. Um diese zu erklären, müssen wir zunächst die durchschnittlichen Laufzeiten der Untermenge 4.1 betrachten. Hier zeigt sich, dass die Laufzeiten für alle Horizontlängen außer 36h bei unter 10ms und bei 36h Horizont bei über 100ms liegen. Der Grund dafür sind Ergebnisrouten, die bei 48h Horizont erst nach 36h nach Beginn des Horizonts am Ziel ankommen. Diese Ergebnisrouten existieren bei 36h Horizont nicht, weshalb die Suche lange nach einer weiteren Ergebnisroute sucht, diese aber nicht findet. Das Target-Pruning greift dabei erst spät. Bei 48h werden diese Ergebnisrouten gefunden und das Target-Pruning kann dann anhand von den zugehörigen Kostenwerten im Ziel prunen. Die absoluten Zeitpunkte (also nach wie vielen relaxierten Knoten in der Vorwärtssuche), zu denen die Rückwärtssuche

gestartet wird, sind bei allen Horizontlängen ähnlich. Durch die höhere Laufzeit bei 36h Horizont wird der relative Zeitpunkt hier kleiner als bei den übrigen Horizonten. Bei den jeweils zweiten Untermengen sind die Zeitpunkte im vergleich zu den ersten Untermengen viel kleiner. Bei $x = 4$ und $x = 6$ liegen die Zeitpunkte meist unter 2%, bei $x = 8$ sind sie bei Untermenge 4.2 mit ca. 40% am höchsten und bei Untermenge 3.2 mit ca. 11% am geringsten.
Mit steigendem x wird die Rückwärtssuche immer später gestartet. Um die Auswirkungen des späteren Starts der Rückwärtssuche zu untersuchen, sind in Abbildung 7.10 die Laufzeiten der Strategien *potrun_x* kombiniert mit der Strategie *softlimit_10k* dargestellt.

Abbildung 7.10: Durchschnittliche Laufzeit der Strategie *potrun_x* kombiniert mit *softlimit_10k*. In jedem Diagramm ist dabei die Laufzeit für eine Testmenge auf vier verschiedenen Horizontlängen dargestellt. Die durchgezogenen Linien stellen die durchschnittliche Laufzeit von *RouBIRT* dar. Die schwarzen vertikalen Striche stellen die Standardabweichung dar.



(a) Testmenge 1

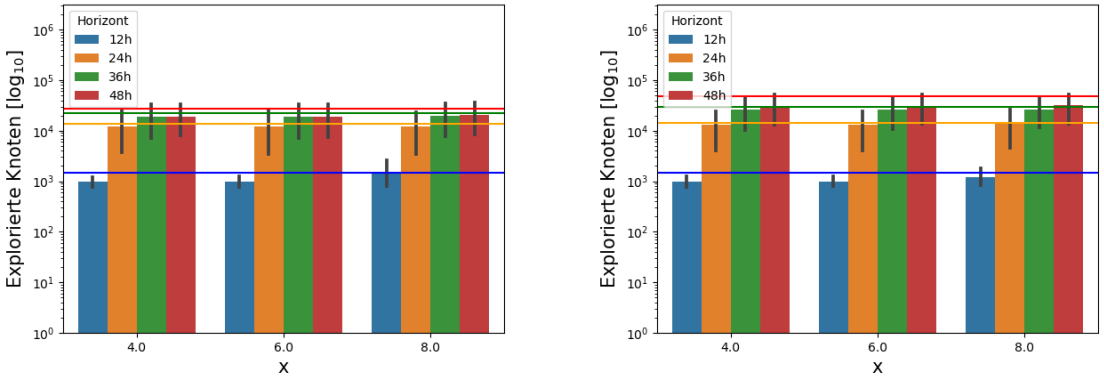
(b) Testmenge 2

(c) Testmenge 3

(d) Testmenge 4

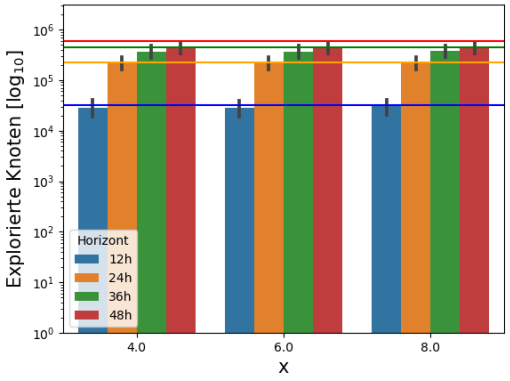
Wir stellen fest, dass bei den Testmengen 1 & 2 die Laufzeit mit steigendem x meistens nur leicht ansteigt, während sie bei den Testmengen 3 & 4 sogar leich fällt. Die Laufzeiten verhalten sich also entgegen der Erwartung, dass (falls wir die Rückwärtssuche starten) ein früherer Start für die Laufzeit besser ist als ein später Start.
Um dieses Verhalten zu erklären, sind in Abbildung 7.11 die Anzahlen der explorierten Knoten in der Vorwärtssuche aufgeführt.

Abbildung 7.11: Durchschnittliche Anzahl explorierter Knoten bei der Strategie *potrun_x* kombiniert mit *softlimit_10k*. In jedem Diagramm ist dabei die jeweilige Anzahl für eine Testmenge auf vier verschiedenen Horizontlängen dargestellt. Die durchgezogenen Linien stellen jeweils die Anzahl der explorierten Knoten im *RouBIRT*-Algorithmus dar. Die schwarzen vertikalen Striche stellen die Standardabweichung dar.

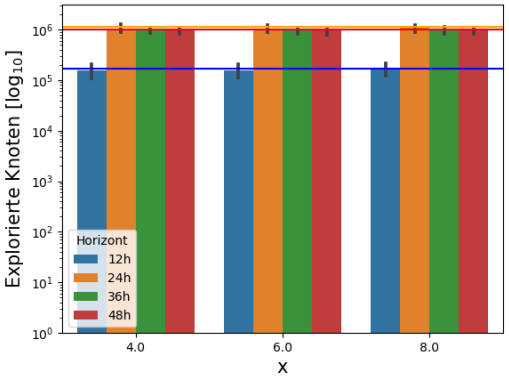


(a) Testmenge 1

(b) Testmenge 2



(c) Testmenge 3



(d) Testmenge 4

Es ist zu sehen, dass sich die Anzahl der explorierten Knoten in allen Testmengen relativ nur sehr wenig verändert. Für die Untermengen sehen die Diagramme qualitativ jeweils gleich aus. Daraus wird deutlich, dass wir durch einen früheren Start der Rückwärtssuche den Suchraum nicht wesentlich stärker einschränken können als durch einen späteren Start. Wir wählen daher die Strategie *potrun_8* als beste dieser Strategien aus.

Key-Start

In diesem Abschnitt evaluieren wir die Strategie *keystart*. Dazu sind in Tabelle 7.6 die Anteile der Queries aufgeführt, bei denen die Rückwärtssuche auf den jeweils ersten Untermengen gestartet wird, und zu welchem Zeitpunkt der Suche sie durchschnittlich gestartet wird.

Tabelle 7.6: Übersicht über den Start der Rückwärtssuche bei der Strategie *keystart*. Die Spalte *Anteil* enthält den Anteil der Queries in %, bei denen die Rückwärtssuche gestartet wurde. Die Spalte *Zeitpunkt* enthält den durchschnittlichen prozentualen Zeitpunkt, zu dem die Rückwärtssuche gestartet wurde (in %). Dabei werden Queries, bei denen die Rückwärtssuche nicht gestartet wurde, vernachlässigt.

(a) Untermenge 1.1

Horizont	Anteil	Zeitpunkt
12h	1.41	15.67
24h	1.59	19.76
36h	3.96	51.59
48h	3.96	51.33

(b) Untermenge 1.2

Horizont	Anteil	Zeitpunkt
12h	100.00	3.21
24h	100.00	0.22
36h	100.00	0.16
48h	97.06	0.16

(c) Untermenge 2.1

Horizont	Anteil	Zeitpunkt
12h	0.92	29.34
24h	1.20	40.82
36h	3.84	66.89
48h	7.44	76.44

(d) Untermenge 2.2

Horizont	Anteil	Zeitpunkt
12h	100.00	3.69
24h	91.67	0.37
36h	92.31	0.22
48h	92.45	0.19

(e) Untermenge 3.1

Horizont	Anteil	Zeitpunkt
12h	1.88	13.98
24h	3.91	33.93
36h	6.97	53.38
48h	6.97	53.37

(f) Untermenge 3.2

Horizont	Anteil	Zeitpunkt
12h	100.00	0.13
24h	97.67	0.06
36h	97.78	0.04
48h	97.79	0.03

(g) Untermenge 4.1

Horizont	Anteil	Zeitpunkt
12h	3.03	13.18
24h	10.00	19.13
36h	12.90	0.27
48h	14.52	20.85

(h) Untermenge 4.2

Horizont	Anteil	Zeitpunkt
12h	100.00	0.26
24h	100.00	0.09
36h	100.00	0.10
48h	100.00	0.10

Die Anteile der Queries, bei denen die Rückwärtssuche gestartet wird, sind bei jeder der ersten Untermengen leicht geringer als bei *potrun_4*. Im Vergleich zur Strategie *potrun_8* sind sie leicht erhöht. Die Zeitpunkte, zu denen die Rückwärtssuche gestartet wird, sind leicht später als bei der Strategie *potrun_4*. Bei den jeweils zweiten Testmengen wird die Rückärtssuche bei allen Queries nach durchschnittlich weniger als 1% der Suche gestartet. Die Anteile liegen dabei immer über 90%, meist sogar bei 100%. Daher ist die Strategie *keystart* sinnvoll in dem Sinne, dass die Rückwärtssuche bei langsamen Queries immer gestartet wird, um diese zu beschleunigen.
Um die Laufzeiten der Strategie *keystart* mit der Strategie *potrun_8* zu vergleichen, sind in Tabelle 7.7 die Laufzeiten auf den jeweiligen Testmengen aufgeführt. Wir wählen die Strategie *potrun_8* zum Vergleich, da diese unter allen Strategien *potrun_x* die besten Laufzeiten erzielt hat.

Tabelle 7.7: Übersicht über den Start der Rückwärtssuche bei der Strategie *keystart*. Die Spalte *Anteil* enthält den Anteil der Queries, bei denen die Rückwärtssuche gestartet wurde. Die Spalte *Zeitpunkt* enthält den durchschnittlichen prozentualen Zeitpunkt, zu dem die Rückwärtssuche gestartet wurde. Dabei werden Queries, bei denen die Rückwärtssuche nicht gestartet wurde, vernachlässigt.

(a) Testmenge 1

Horizont	*potrun_8*	*keystart*
12h	1.9	1.7
24h	14.0	13.6
36h	20.6	19.5
48h	22.7	21.3

(b) Testmenge 2

Horizont	*potrun_8*	*keystart*
12h	1.8	1.7
24h	14.3	13.7
36h	38.4	35.8
48h	60.8	53.6

(c) Testmenge 3

Horizont	*potrun_8*	*keystart*
12h	26.7	26.2
24h	271.1	261.8
36h	785.6	699.3
48h	794.0	815.4

(d) Testmenge 4

Horizont	*potrun_8*	*keystart*
12h	147.0	134.8
24h	1080.4	1091.9
36h	1086.0	1146.3
48h	1133.1	1179.7

Auf den Testmengen 1 & 2 ist die Strategie *keystart* auf allen Horizontlängen leicht schneller als die Strategie *potrun_8*. Auf Testmenge 3 gilt das für die Horizontlängen 12h, 24h und 36h ebenso. Bei Horizontlänge 48h ist die Laufzeit leicht höher. Auf Testmenge 4 ist die Laufzeit mit Strategie *keystart* (mit Ausnahme von 12h Horizont) immer leicht höher als mit Strategie *potrun_8*. Für die jeweiligen Untermengen ergeben sich qualitativ die gleichen Laufzeit-Verhältnisse.

Insgesamt können wir also festhalten, dass eine Limitierung der Rückwärtssuche bessere Laufzeiten erzielt als die Nicht-Limitierung der Rückwärtssuche. Dabei erzielt die Strategie *soft_10k* die besten Laufzeiten. Zusätzlich ist es für die Laufzeit der schnellen Queries gut, die Rückwärtssuche erst später zu starten. Dabei ist es je nach Query-Menge besser, die Strategie *potrun_8* oder die Strategie *keystart* zu verwenden. Dabei schafft es die Rückwärtssuche aber auch mit der besten (getesteten) Strategie nicht, die langsamen Queries wesentlich zu beschleunigen. Der Speedup liegt bei den Testmengen 1-3 zwischen 1.3 und 1.5, bei Testmenge 4 liegt er bei ca. 1.05.

7.6 Parallelisierung

Wenn wir die Strategien außer acht lassen, bleibt noch eine Möglichkeit, die Rückwärtssuche in den *RouBIRT*-Algorithmus einzubinden: Wir führen die Rückwärtssuche in einem eigenen Thread aus. Dabei muss lediglich der Zugriff auf die Rückwärtsprofile synchronisiert werden, falls die Rückwärtssuche gerade in ein Profil schreibt, das die Vorwärtssuche liest. Ansonsten sind alle Operationen auf gemeinsamen Daten Lese-Operationen, für die keine Synchronisierung notwendig ist. Die Laufzeiten des parallelisierten *BD-RouBIRT* sind in Tabelle 7.8 für die jeweils ersten Untermengen dargestellt.

Tabelle 7.8: Übersicht über die durchschnittlichen Laufzeiten des parallelisierten *BD-RouBIRT* für die ersten Untermengen verglichen mit den Laufzeiten von *RouBIRT* und *BD-RouBIRT* mit den Strategien *soft_10k* und *potrun_8*. Die Laufzeiten sind jeweils in ms angegeben.

(a) Untermenge 1.1

Horizont	*RouBIRT*	*potrun_8_soft_10k*	parall. *BD-RouBIRT*
12h	1.3	1.4	1.5
24h	1.6	1.8	1.9
36h	1.6	1.8	1.9
48h	1.8	1.9	1.9

(b) Untermenge 2.1

Horizont	*RouBIRT*	*potrun_8_soft_10k*	parall. *BD-RouBIRT*
12h	1.3	1.4	1.5
24h	1.6	1.7	1.8
36h	1.7	1.8	1.9
48h	1.9	2.1	2.1

(c) Untermenge 3.1

Horizont	*RouBIRT*	*potrun_8_soft_10k*	parall. *BD-RouBIRT*
12h	1.2	1.4	1.5
24h	1.5	1.6	1.7
36h	1.7	2.4	1.9
48h	1.7	2.3	1.9

(d) Untermenge 4.1

Horizont	*RouBIRT*	*potrun_8_soft_10k*	parall. *BD-RouBIRT*
12h	2.3	2.5	2.6
24h	4.3	4.6	4.8
36h	129.1	106.9	70.5
48h	5.1	5.4	5.6

Bei den Untermengen 1.1, 2.1 und 3.1 sind die Laufzeiten überall sehr ähnlich. Wir können hier weder mit der Strategie *potrun_8_soft_10k* noch mit der Parallelisierung ein Beschleunigung erreichen. Das gilt mit Ausnahme von 36h Horizont auch für due Untermenge 4.1. Bei 36h Horizont können wir mit der Parallelisierung im Vergleich zur Strategie *potrun_8_soft_10k* nochmal eine zusätzliche Beschleunigung, sodass wir die Laufzeit insgesamt fast halbieren können.
In Tabelle 7.9 sind die Laufzeiten des parallelisierten *BD-RouBIRT* für die zweiten Untermengen aufgeführt.

Tabelle 7.9: Übersicht über die durchschnittlichen Laufzeiten des parallelisierten *BD-RouBIRT* verglichen mit den Laufzeiten von *RouBIRT* und *BD-RouBIRT* mit den Strategien *soft_10k* und *potrun_8*. Die Laufzeiten sind jeweils in ms angegeben.

(a) Untermenge 1.2

Horizont	*RouBIRT*	*potrun_8_soft_10k*	parall. *BD-RouBIRT*
12h	27.5	29.5	15.4
24h	606.4	647.6	456.0
36h	1027.0	994.7	1158.5
48h	1645.4	1102.0	1108.3

(b) Untermenge 2.2

Horizont	*RouBIRT*	*potrun_8_soft_10k*	parall. *BD-RouBIRT*
12h	19.1	14.3	11.5
24h	418.5	428.2	321.8
36h	1326.3	1245.2	1114.8
48h	3148.1	1996.2	1608.0

(c) Untermenge 3.2

Horizont	*RouBIRT*	*potrun_8_soft_10k*	parall. *BD-RouBIRT*
12h	338.0	336.9	243.7
24h	3485.3	3568.0	2216.2
36h	11042.5	10368.1	7638.1
48h	16226.8	10480.0	7979.2

(d) Untermenge 4.2

Horizont	*RouBIRT*	*potrun_8_soft_10k*	parall. *BD-RouBIRT*
12h	197.4	211.9	179.9
24h	1483.0	1563.8	1408.3
36h	1617.1	1525.9	1546.6
48h	1717.0	1639.8	1725.9

Bei Testmenge 1 bringt die Parallelisierung bei allen Horizontlängen eine Beschleunigung. Im Vergleich zur Strategie *potrun_8_soft_10k* ist der parallelisierte *BD-RouBIRT* bei der Horizontlänge 36h leicht langsamer. Ein möglicher Grund dafür ist das häufigere Berechnen von vorläufigen Zielkosten, da hier mehr Knoten in der Rückwärtssuche exploriert werden. Bei Testmenge 2 bringt die Parallelisierung sowohl im Vergleich zur Laufzeit von *RouBIRT* als auch im Vergleich mit der Strategie *potrun_8_soft_10k* eine Beschleunigung. Diese beträgt bei 48h Faktor 2. Bei Testmenge 3 bringt die Parallelisierung die größte Beschleunigung. Hier erreichen wir bei 48h eine Halbierung der durchschnittlichen Laufzeit von 1.2s auf 600ms. Bei Testmenge 4 bewirkt die Parallelisierung entweder gar keine oder nur eine sehr geringe Beschleunigung. Im Vergleich mit der Strategie *potrun_8_soft_10k* bringt die Parallelisierung teilweise gar keine Beschleunigung. Diese Aussagen gelten bei allen Testmengen für die jeweils zweite Untermenge qualitativ ebenso. Für die jeweils erste Untermenge ist die durchschnittliche Laufzeit jeweils leicht höher als die Laufzeit von *RouBIRT*, allerdings sind diese Laufzeitdifferenzen nie größer als 1ms.

Insgesamt bringt die Parallelisierung von *BD-RouBIRT* bei den Testmengen 1-3 eine weitere Beschleunigung, bei Testmenge 4 bleibt die Laufzeit in etwa gleich.

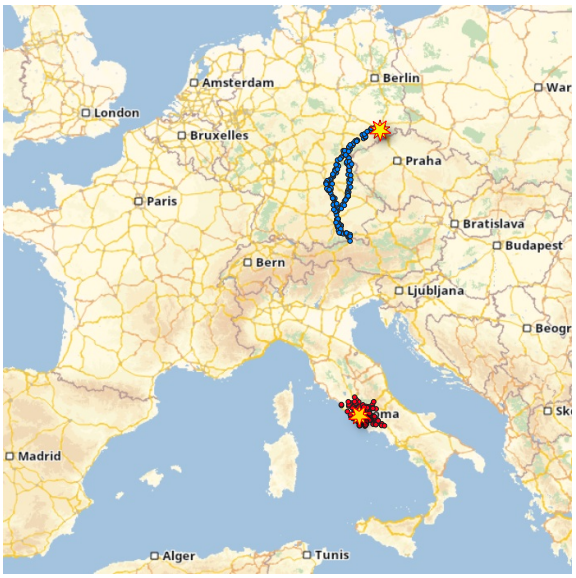
7.7 Beispiel-Query

In diesem Abschnitt untersuchen wir anhand einer Query, warum die Rückwärtssuche die Laufzeit der langsamen Queries nicht auf einen Wert unter 10s bringen kann. Dazu verwenden wir die Strategie *ratio_1* und keine Limitierung. Zudem läuft die Rückwärtssuche bereits ab dem Start des Algorithmus. Die betrachtete Query ist aus Testmenge 3. Der Planungshorizont beginnt am Freitag, dem 06. Juli 2018, und dauert 48h. Der Start der Query liegt in der Nähe von Dresden und das Ziel liegt in Rom. In den folgenden Abbildungen sind Start und Ziel jeweils durch einen gelben Stern markiert.
Zunächst ist in Abbildung 7.12 der Suchraum der Rückwärtssuche am Ende der Query visualisiert. Im Folgenden werden die in der Rückwärtssuche explorierten Knoten immer rot dargestellt sein. Zudem sind die Knoten gesampelt, sodass nur jeder 1000. Knoten in der Abbildung zu sehen ist.

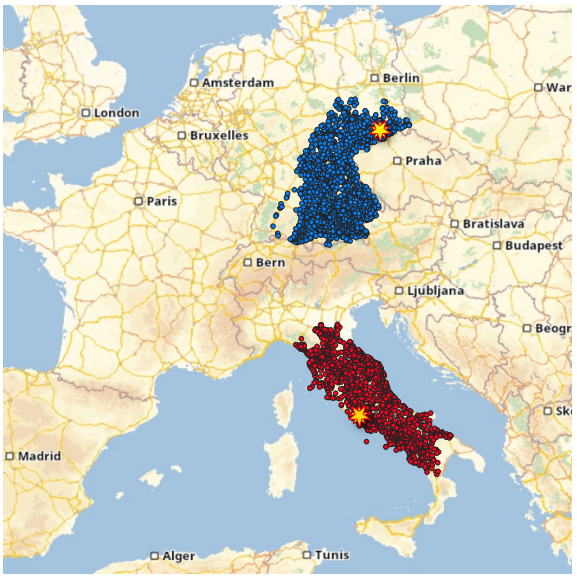
Abbildung 7.12: Suchraum der Rückwärtssuche der Beispiel-Query aus Testmenge 3. Die gelben Sterne stellen Start (oben) und Ziel (unten) dar.

Wir sehen, dass sich die Rückwärtssuche kreisförmig ausbreitet, da sie nicht zielgerichtet ist. Im Folgenden betrachten wir den Suchraum der Vorwärtssuche im Lauf des Algorithmus zu verschiedenen Zeitpunkten. In Abbildung 7.13 sind die Suchräume zu den jeweiligen Zeitpunkten dargestellt. Die Knoten der Vorwärtssuche sind im Folgenden immer blau dargestellt und sind wie die Rückwärtsknoten gesampelt. Zudem sind in grün die Knoten dargestellt, bei denen wir nach Kosten prunen können. Diese sind nicht gesampelt.

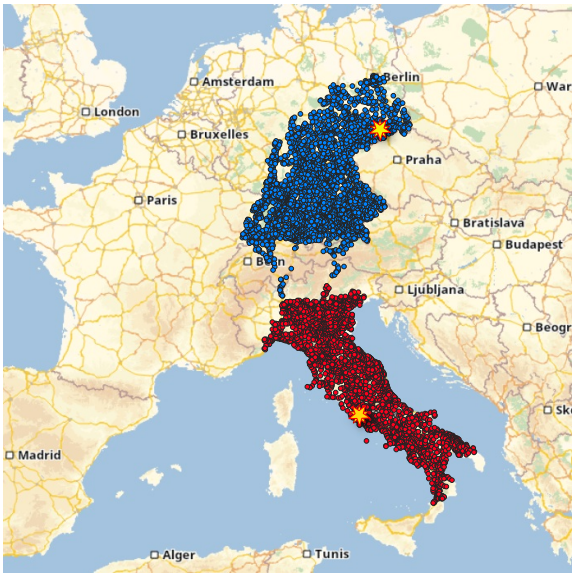
Abbildung 7.13: Visualisierung der Suchräume für die Beispiel-Query aus Testmenge 3



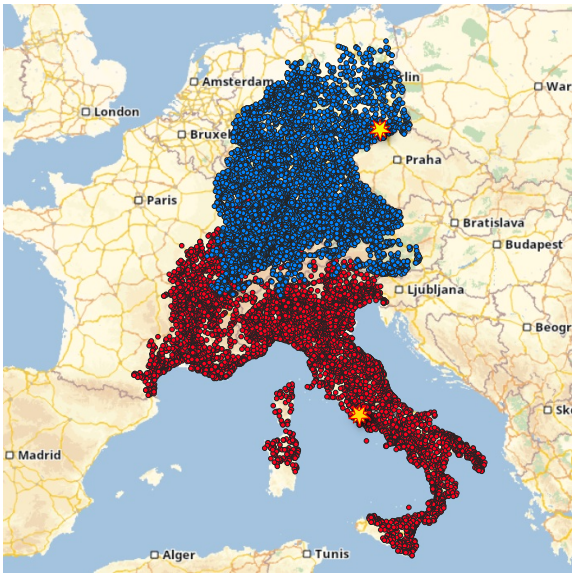
(a) Zeitpunkt 1



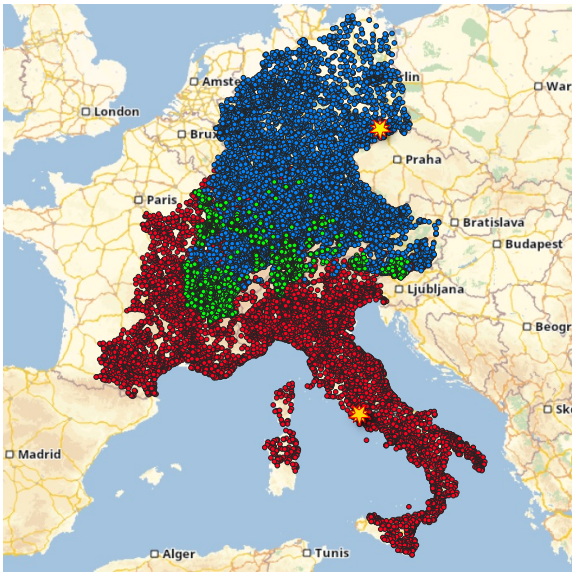
(b) Zeitpunkt 2



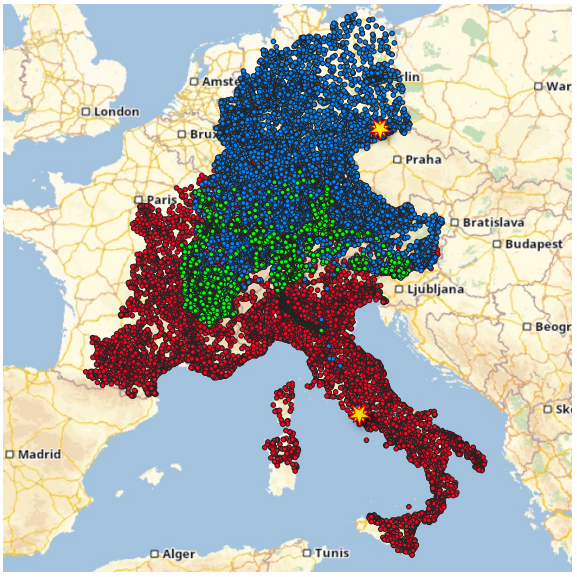
(c) Zeitpunkt 3



(d) Zeitpunkt 4



(e) Zeitpunkt 5



(f) Zeitpunkt 6

Wir sehen, dass die Vorwärtssuche durch die A*-Potentiale sehr schnell an die deutsch-österreichische Grenze gelangt (Zeitpunkt 1), wo zum Ankunfts-Zeitpunkt das Nachtfahrverbot in Österreich gilt. Daher sind zu diesem Zeitpunkt bereits zwei Pfade erkennbar, entlang denen die Vorwärtssuche sucht. Die Vorwärtssuche versucht dann zunächst, das Nachtfahrverbot in Österreich und der Schweiz zu umfahren (Zeitpunkt 2). Dadurch breitet sich der Suchraum nach Westen aus und die Suche geht zunächst nicht direkt in Richtung Ziel weiter. Diese Ausbreitung setzt sich bis zum Ende der Suche fort. Ab dem vierten Zeitpunkt überlagern sich die Suchräume der Vorwärts- und Rückwärtssuche. Dadurch sind zum fünften Zeitpunkt Knoten zu sehen, bei denen nach Kosten geprunt werden kann. Der sechste Zeitpunkt stellt die Suchräume am Ende der Suche dar. Wir sehen, dass sich die Knoten, bei denen nach Kosten Segmente geprunt werden können, nur um die Schweiz und Österreich konzentrieren. Insbesondere können wir nördlich davon nur nach Ankunftszeit prunen, was die weite Ausbreitung des Vorwärtssuchraums nach Westen und Norden nicht verhindert. Das ist der Grund, warum die Rückwärtssuche die langsamen Queries nicht auf unter 10s beschleunigen kann.

8. Evaluation des modifizierten Basisalgorithmus

In diesem Kapitel evaluieren wir den *RouBIRT-DB*-Algorithmus. Dabei gehen wir darauf ein, wie sich die Ergebnisse durch die Modellerweiterung verändern. Zudem führen wir eine eingeschränkte quantitative Evaluation durch.

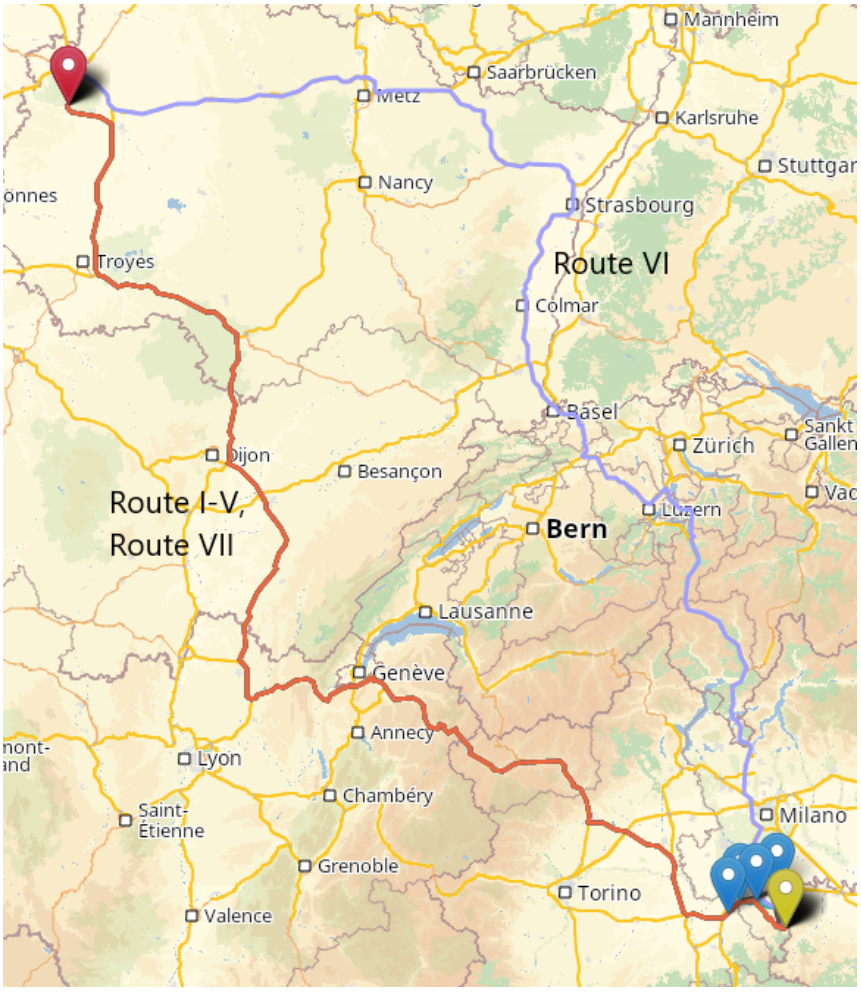
8.1 Planungshorizont, Testmengen und Parameter

Für die Parameter *limit* und *break* verwenden wir die Werte 9h und 11h. Dies entspricht der Regelung für LKW-Fahrer in Europa gemäß Verordnung (EG) Nr.561/2006. Als Horizontlängen verwenden wir 24h und 36h. Für 12h Horizontlänge bleiben die Ergebnispfade entweder erhalten (falls die Fahrzeit höchstens 9h beträgt) oder die ursprünglichen Ergebnispfade sind nicht mehr in der Lösung enthalten (falls die Fahrzeit mehr als 9h beträgt), da eine Fahrzeit von mehr als 9h und eine Pause von 11h innerhalb des 12-stündigen Horizonts nicht möglich sind. Bei 48h Horizontlänge kommt es bei einigen Queries zu Memory-Swapping. Da diese Queries bis zur Abgabe der Arbeit nicht vollständig ausgeführt worden wären, werden wir diese Horizontlänge ebenfalls vernachlässigen.
Die Laufzeit der einzelnen Queries ist teilweise sehr viel länger als die ursprüngliche Laufzeit von *RouBIRT* und kann bis zu einigen Stunden groß werden. Daher haben wir für die qualitative Evaluation von *RouBIRT-DB* die Größe der Testmengen reduziert. Bei den Testmengen 1-3 verwenden wir nur die ersten 100 Queries, bei Testmenge 4 die ersten 50 Queries.
Um die RAM-Auslastung weiter zu reduzieren, unterscheiden wir nicht zwischen Nicht-Parkplätzen und Parkplätzen verschiedener Kategorien, sondern nur noch zwischen Nicht-Parkplätzen und Parkplätzen. Dabei sind die Kosten für das Warten an einem Nicht-Parkplatz 14 und die Kosten für das Warten an einem Parkplatz 3 (jeweils pro Zeiteinheit).

8.2 Ergebnisveränderung

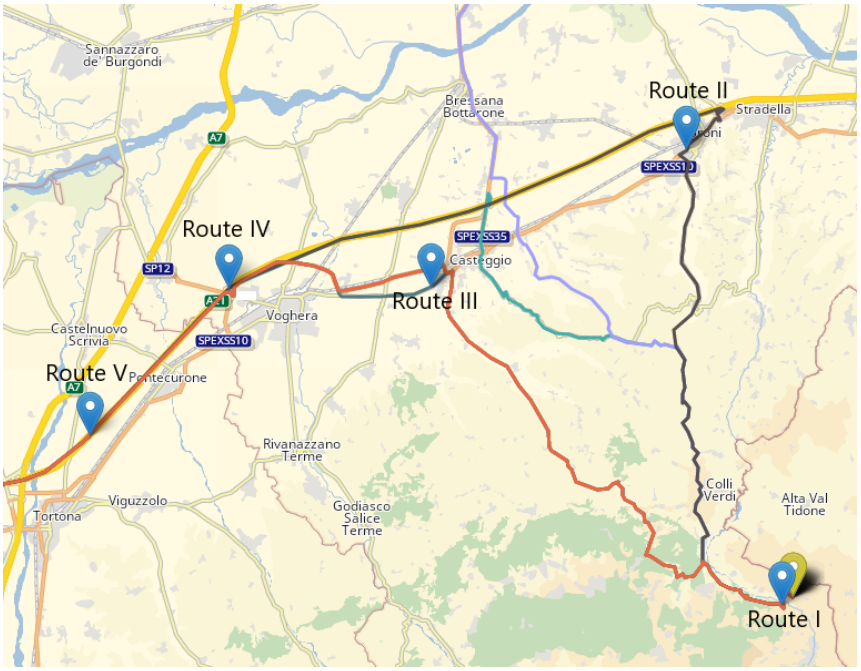
Wir evaluieren nun anhand von zwei Beispiel-Queries, wie sich das Ergebnis durch die Modellerweiterung ändert. Dabei stellen wir jeweils zunächst die Ergebnisse des *RouBIRT*-Algorithmus vor, erklären, wie diese entstehen, und beschreiben anschließend, wie sich die Ergebnisse beim *RouBIRT-DB*-Algorithmus verändern.
Wir betrachten zunächst eine Query aus Testmenge 3, die in Abbildung 8.1 dargestellt ist.

Abbildung 8.1: Query aus Testmenge 3 mit Start östlich von Paris, Ziel in der Nähe von Mailand und 7 Ergebnisrouten



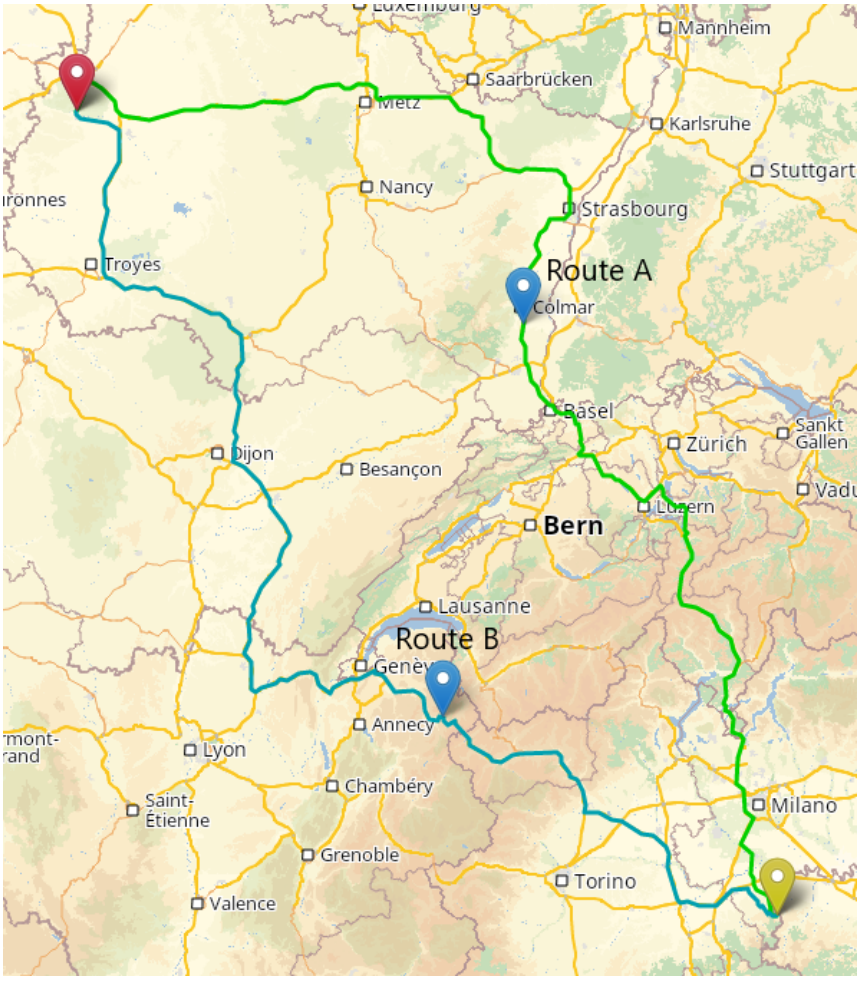
Der Start liegt östlich von Paris (roter Punkt) und das Ziel in der Nähe von Mailand (gelber Punkt). Diese Query hat ursprünglich sieben Ergebnisrouten, die alle eine reine Fahrzeit von ca. 14h besitzen. Jede dieser Routen ist also unter der Berücksichtigung der maximalen Fahrzeit von 9h nicht ohne Pause fahrbar. Der Planungshorizont beginnt am 06. Juli 2018 (Freitag) um 18.00 Uhr und dauert 36h. Alle Routen beinhalten daher das Abwarten des Samstagsfahrverbots in Italien in den Sommermonaten von 8h. Fünf der Ergebnisrouten (Routen I - V) unterscheiden sich nur kurz vor dem Ziel darin, wo sie das Samstagsfahrverbot abwarten. Diese Routen fahren vom Start aus zunächst nach Süden, anschließend westlich um die Schweiz und dann zum Ziel. Daher beinhalten diese Routen nicht das Abwarten des Nachtfahrverbots in der Schweiz. In Abbildung 8.2 ist der Bereich um das Ziel vergrößert dargestellt.

Abbildung 8.2: Vergrößerter Ziel-Ausschnitt der Query aus Abbildung 8.1



Dort sind die Unterschiede dieser ersten fünf Routen sichtbar. Hier sieht man zudem, dass die erste dieser fünf Routen kurz vor dem Ziel auf einer Kante wartet, also gar keinen Parkplatz anfährt. Dadurch ist diese Route früher als alle anderen Pfade am Ziel. Zusätzlich gibt es eine weitere Route (Route VI), die zunächst nach Osten und anschließend nach Süden zum Ziel fährt. Diese Route startet am 07. Juli (Samstag) um 04.29:02 Uhr und muss somit ebenfalls nicht das Nachtfahrverbot in der Schweiz abwarten. Dieser Pfad befindet sich durch seine spätere Abfahrt erst nach Ablauf des Samstagsfahrverbots in Italien und wartet daher nicht vor dem Ziel. Die siebte Route (Route VII) verläuft wie die ersten fünf Routen und startet am 07. Juli um 07.03:12 Uhr. Durch die spätere Abfahrt befindet sich diese Route ebenfalls erst nach Ablauf des Samstagsfahrverbots in Italien und muss daher kurz vor dem Ziel nicht warten.
Bis auf die letzten beiden Routen kommen alle Pfade mehr als 11h vor Ende des Planungshorizonts am Ziel an und haben somit die Möglichkeit, zusätzlich eine Pause von 11h auf einem Parkplatz zu machen, um die Fahrzeitlimitierung von 9h nicht zu verletzen. Warum der *RouBIRT*-Algorithmus die Pfade I-V berechnet und nicht nur einen Pfad anstelle dieser 5 Pfade, werden wir im Folgenden noch erläutern.
Der *RouBIRT-DB*-Algorithmus berechnet für diese Query nur noch zwei Ergebnisrouten. Diese sind in Abbildung 8.3 dargestellt.

Abbildung 8.3: Ergebnisrouten unter Berücksichtigung von Lenk- und Ruhezeiten der Query aus Abbildung 8.1

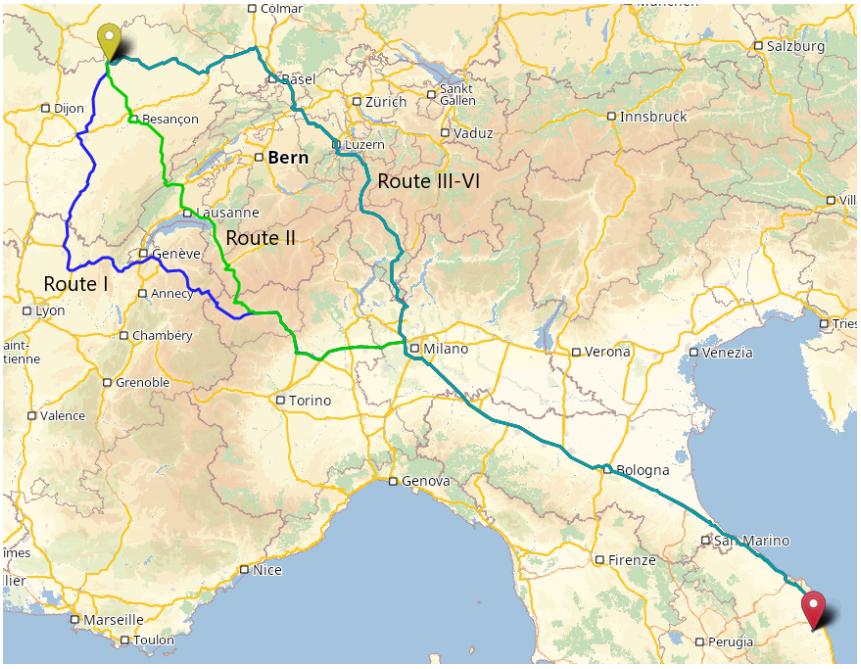


Die erste Route (Route A, grün) kann Route VI aus den ursprünglichen Routen zugeordnet werden. Während Route VI erst am 07. Juli um 03.29:02 Uhr startet, um nicht in das Nachtfahrverbot der Schweiz zu fahren, startet Route A bereits am 06. Juli um 18.00 Uhr zu Beginn des Planungshorizontes. Route A macht nahe der deutsch-französischen Grenze 11h Pause, um die Fahrzeitlimitierung einzuhalten, und ist dann erst nach Ende des Nachtfahrverbots in der Schweiz. Außerdem ist die Route A erst nach Ende des Samstagsfahrverbots in Italien und fährt ebenso wie Route VI vor dem Ziel keinen Parkplatz mehr an.
Die zweite Route (Route B, blau) kann den Routen I-V sowie VII zugeordnet werden. Sie startet am 06. Juli 2018 um 20.03:12 Uhr und macht dann südöstlich von Genf eine Pause von 11h, um die Fahrzeitlimitierung einzuhalten. Durch die spätere Abfahrt und die Pause ist die Route B ebenso wie die Route A erst nach Ende des Samstagsfahrverbot in Italien. Dadurch ist auf der Route B kurz vor dem Ziel kein Warten mehr nötig und die Routen I-V sowie VII fallen zu einer einzelnen Route zusammen.
Jetzt erläutern wir, warum der *RouBIRT*-Algorithmus nicht auch nur die Route B (mit späterer Abfahrt und kürzerer Pause) berechnet hat, sondern stattdessen die Routen I-V. Die Routen I-V entstehen durch den folgenden Trade-Off: Fahre einen größeren Umweg, warte dafür das Samstagsfahrverbot an einem Parkplatz ab, der sich näher am Ziel befindet, um dann früher am Ziel zu sein. Bei Route I ist der “Parkplatz” eine Kante, die näher am Ziel ist als alle Parkplätze, die auf den Routen II-V angefahren werden. Diese Routen werden nicht von Route VII dominiert, da sich die Parkplätze in Italien befinden. Sie liegen

also innerhalb des während des Samstagsfahrverbots gesperrten Gebiets. Daher kommt es hier zu mehreren Routen, die im Wesentlichen gleich sind, und sich sowohl in ihren Kosten als auch in ihrer Abfahrts- und Ankunftszeit nur leicht unterscheiden.

Im Folgenden betrachten wir noch die Änderung des Ergebnisses anhand einer Query aus Testmenge 4. In Testmenge 4 gibt es noch weitere Queries, bei denen eine ähnliche Änderung des Ergebnisses auftritt.
Die Query ist gemeinsam mit den Ergebnisrouten ohne Berücksichtigung der Fahrzeitlimitierung in Abbildung 8.4 dargestellt.

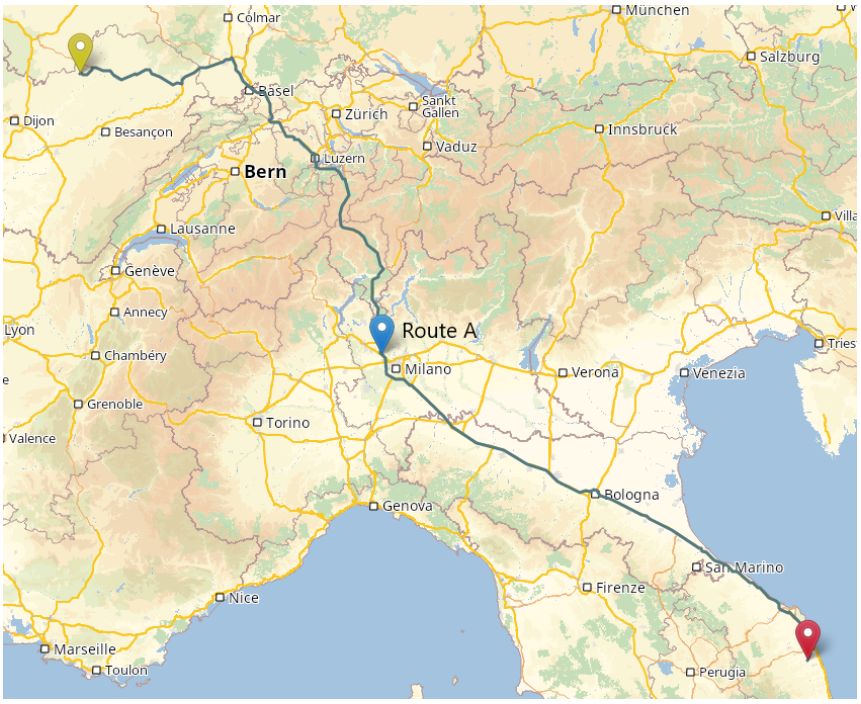
Abbildung 8.4: Query aus Testmenge 4 mit Start im Osten von Italien, Ziel in der Nähe von Besancon und 6 Ergebnisrouten



Der Start liegt im Osten von Italien (roter Punkt) und das Ziel liegt in der Nähe von Besancon in Frankreich (gelber Punkt). Der Planungshorizont beginnt am 2. Juli 2018 (Montag) um 18.00 Uhr und dauert 36h. Der direkte Pfad führt durch die Schweiz und ist aufgrund des Nachtfahrverbots in der Schweiz nicht gültig. Daher berechnet der *RouBIRT*-Algorithmus die im folgenden Absatz beschriebenen Routen.
Die erste Route (dunkelblau, Route I) startet zu Beginn des Planungshorizontes und umfährt die Schweiz. Die zweite Route (grün, Route II) startet knapp 40 min nach Beginn des Planungshorizontes, fährt zunächst wie die dunkelblaue Route und fährt dann durch die Schweiz zum Ziel. Aufgrund der späteren Abfahrt ist das Nachtfahrverbot bereits vorbei, wenn die grüne Route in die Schweiz fährt. Die Routen III-VI unterscheiden sich nur geringfügig und sind daher in der Abbildung nur als eine Route (türkis) sichtbar. Sie fahren im Zeitraum von 3h bis 6h nach Beginn des Planungshorizontes los, fahren zu Beginn wie die dunkelblaue Route und fahren dann weiter östlich als die grüne Route durch die Schweiz zum Ziel. Diese Routen sind aufgrund der Sperrung eines Autobahnabschnitts kurz vor der schweizer Grenze alle optimal. Sie umfahren die Sperrung entweder vollständig (Route III) oder fahren einen Umweg, um zum Ende der Sperrung bereits näher am Ziel bzw. bereits innerhalb des vorher gesperrten Abschnitts zu sein (Route IV-VI). Der Trade-Off ist bei den Routen IV-VI also ein größerer Umweg für eine frühere Ankunft am Ziel. Keine der sechs Routen enthält Warten auf dem Weg (die Routen II-VI warten jeweils am Start) und

alle Routen haben eine Fahrzeit zwischen 15h und 17h. Daher sind unter Berücksichtigung der Fahrzeitlimitierung alle Routen nicht ohne Pause fahrbar. Zudem kommen alle Routen früher als 24h nach Beginn des Planungshorizonts am Ziel an, wodurch auf allen Routen eine Pause von 11h möglich wäre, um die Fahrzeitlimitierung einzuhalten.
Das Ergebnis unter Berücksichtigung der Fahrzeitlimitierung ist in Abbildung 8.5 dargestellt.

Abbildung 8.5: Ergebnisroute der Query aus Abbildung 8.4 unter Berücksichtigung der Fahrzeitlimitierung



Wir sehen, dass es jetzt nur noch eine Ergebnisroute gibt (Route A). Die Route A startet zu Beginn des Planungshorizontes und macht fast direkt vor der schweizer Grenze eine Pause von 11h, um die Fahrzeitlimitierung einzuhalten. Die Pause wird indirekt auch dafür genutzt, das Nachtfahrverbot in der Schweiz abzuwarten. Dadurch muss die Route A die Schweiz im Gegensatz zu den Routen I und II nicht (teilweise) umfahren, sondern kann ohne weitere Sperrungen auf dem Weg direkt zum Ziel fahren.

Anhand dieser beiden Beispiele können wir sehen, dass sich bei Queries, bei denen das Abwarten einer Sperrung nötig ist und die Fahrzeit länger als 9h ist, die Anzahl der Ergebnisrouten durch die Berücksichtigung der Fahrzeitlimitierung reduziert. Das Ergebnis wird einfacher und oft fallen die Routen sogar zu einer einzelnen Route zusammen. Bei den getesteten Queries treten Veränderungen anderer Art auf, wie z.B., dass die Anzahl der Ergebnispfade wächst, nicht auf.

8.3 Laufzeit

In diesem Abschnitt evaluieren wir die Laufzeitveränderung zwischen *RouBIRT* und *RouBIRT-DB*. Der *RouBIRT-DB*-Algorithmus braucht auf vielen Queries mehr als 64GB RAM, wodurch es zu Swapping kommt. Daher hatten wir beim Testen für jede Query ein Zeitlimit von 20min. Die Laufzeit auf kleineren Instanzen zu messen war dabei keine Option, da es hier nur selten dazu kommt, dass Sperrungen umfahren oder abgewartet werden müssen. Hier könnten wir die zusätzliche Laufzeit also nur bedingt evaluieren. Für

die folgenden Betrachtungen betrachten wir nur die Horizontlänge von 36h, da bei 24h einige der Queries, die bei *RouBIRT* über 10s gedauert haben, keine Pfade mehr enthalten, die vor Ende des Planungshorizonts am Ziel ankommen. Dadurch sind die Laufzeiten bei diesen Queries im Millisekunden-Bereich und ein sinnvoller Vergleich ist nicht möglich.
Wir betrachten zunächst die Queries, bei denen der kürzeste Pfad auch unter Berücksichtigung von Sperrungen gültig ist. Hier ist die Laufzeit von *RouBIRT-DB* sehr klein (einige ms). Hier vergrößert sich die Laufzeit also nicht wesentlich. Die maximale Anzahl an Profilen pro Knoten ist hierbei meistens 1 oder 2. Nun betrachten wir die Queries, bei denen Sperrungen umfahren oder abgewartet werden müssen. Bei Testmenge 3 sind die Laufzeiten von Queries, die vorher zwischen 10s und 30s gebraucht haben, jetzt bei bis zu 6min. Die maximale Anzahl an Profilen an einem Knoten liegt hier meist bei ca. 110. Bei einer Query gibt es einen Knoten mit 290 Profilen. Bei Testmenge 4 ergeben sich bei Queries teilweise Laufzeiten von ca. 15min, wo die Laufzeit von *RouBIRT* bei ca. 4s lag. Die maximale Anzahl an Profilen an einem Knoten liegt hier teilweise bei etwas weniger als 600. Bei den Queries, die das Limit von 20min erreicht haben, liegt die maximale Anzahl an Profilen an einem Knoten meist über 1000, bei einer Query sogar bei ca. 2000. Das ergibt an diesem Knoten insgesamt ca. 3400 Segmente, wenn man alle Profile dieses Knotens betrachtet.

8.4 Modell

In diesem Abschnitt diskutieren wir das verwendete Modell. Unser Modell erlaubt Warten, das zur Einhaltung der Fahrzeitlimitierung dient, nur auf Parkplätzen. Insbesondere zählt es also nicht als Pause, auf einem Nicht-Parkplatz oder einer Kante zu warten. Würden wir das Warten auf Nicht-Parkplätzen (keine Kanten) zulassen, dann entsteht bei allen Nicht-Parkplätzen mindestens ein zusätzliches Profil mit akkumulierter Fahrzeit 0. Wir würden zudem viel mehr Profile mit akkumulierter Fahrzeit 0 berechnen. Daher verschlechtert sich hier die Laufzeit. Die Ergebnisrouten würden sich auf den getesteten Queries nur darin unterscheiden, wo die Pause abgewartet wird. Die Anzahl der Ergebnisrouten würde sich vermutlich nicht ändern, da sich entlang von langen Routen meist ein Parkplatz ohne einen großen Umweg erreichen lässt. Wenn wir das Warten auf Nicht-Parkplätzen und Kanten auch zur akkumulierten Fahrzeit zählen würden, dann entstehen beim Relaxieren einer Kanten e mit einer Sperrung der Länge x von einem Profil mit akkumulierter Fahrzeit a ausgehend nicht mehr nur ein Profil, sondern zwei Profile: das erste Profil hat akkumulierte Fahrzeit $a + \delta(e)$, das zweite akkumulierte Fahrzeit $a + \delta(e) + x$. Diese Profile dominieren sich gegenseitig nicht, weshalb hier pro Knoten mehr Profile entstehen. Daher ist auch hier die Laufzeit höher. Das Ergebnis würde sich nur bei Queries ändern, bei denen im Ergebnis von *RouBIRT-DB* Routen enthalten sind, bei denen eine Sperrung auf einem Nicht-Parkplatz oder einer Kante abgewartet wird. Hier führt das Warten möglicherweise dazu, dass eine weitere Pause auf einem Parkplatz nötig ist. Das führt dann (falls noch weitere Routen im Ergebnis enthalten sind) eventuell dazu, dass diese Route aufgrund der weiteren Pause nicht mehr optimal ist und somit nicht mehr im Ergebnis enthalten ist. Alle Routen, die nicht auf einem Nicht-Parkplatz oder einer Kante warten, bleiben erhalten.
Die Komplexität des *RouBIRT-DB*-Problems wird also durch die Abwandlungen des Modells auf jeden Fall nicht niedriger. Da wir im Rahmen dieser Arbeit (wie in Abschnitt 5.4 bereits erwähnt) nicht näher auf die Komplexität des *RouBIRT-DB*-Problems eingehen, werden wir auch nicht näher untersuchen, ob die Komplexität durch die Abwandlungen des Modells höher wird.

9. Fazit

Wir haben LKW-Routing mit zeitunabhängigen Fahrdauern und Parkplätzen mit unterschiedlicher Qualität betrachtet. Wir haben gesehen, wie man den *RouBIRT*-Algorithmus zu einer bidirektionalen Suche erweitern kann, um die Laufzeit durch stärkeres Pruning zu verbessern. Wir haben verschiedene Strategien untersucht, um die Rückwärtssuche in *RouBIRT* einzubinden. Dabei haben sich für die Strategie *potrun_8* kombiniert mit *softlimit_10k* die besten Laufzeiten ergeben. Zusätzlich haben wir die Laufzeit untersucht, wenn wir die Rückwärtssuche in einem eigenen Thread ausführen. Diese parallelisierte Version hat dabei die besten Laufzeiten erzielt. Insgesamt hat die parallelisierte Version die Laufzeiten einiger langsamen Queries im Durchschnitt halbieren können, viele andere Queries, bei denen Sperrungen berücksichtigt werden mussten, konnten aber nicht wesentlich beschleunigt werden. Der Suchraum der Vorwärtssuche konnte also nicht weit genug eingeschränkt werden, um alle Queries in praktikabler Zeit beantworten zu können.
Des weiteren haben wir den *RouBIRT*-Algorithmus so modifiziert, dass er Lenk- und Ruhezeiten berücksichtigen kann. Dabei hat sich bei einigen Queries, wo die Gesamtfahrdauer mehr als das Fahrtlimit beträgt, die Anzahl der Ergebnisrouten stark reduziert. Dort gab es dann meist nur noch eine Ergebnisroute, die unterwegs eine Pause enthält, um die Fahrzeitlimitierung einzuhalten und gleichzeitig eine Sperrung abzuwarten. Die Laufzeiten und der Speicherverbrauch sind bei Queries, die ohne Lenk- und Ruhezeiten bereits nicht in praktikabler Laufzeit beantwortbar waren, sehr stark angestiegen.
Um bei diesen Queries die Laufzeit zu reduzieren, ist es möglich, die Rückwärtssuche auch in den modifizierten *RouBIRT*-Algorithmus einzubauen. Dadurch ist ggf. ein stärkeres Pruning möglich und die Laufzeit (und der Speicherverbrauch durch die Kostenprofile der Vorwärtssuche) könnte sich stark reduzieren.

Literaturverzeichnis

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

[Brä18] Christian Bräuer: *Route Planning with Temporary Road Closures*. Master Thesis, Karlsruhe Institute of Technology, 2018.

[GSSV12] Robert Geisberger, Peter Sanders, Dominik Schultes und Christian Vetter: *Exact Routing in Large Road Networks Using Contraction Hierarchies*. Transportation Science, 46(3):388–404, August 2012.

[HNR68] Peter E. Hart, Nils Nilsson und Bertram Raphael: *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*. IEEE Transactions on Systems Science and Cybernetics, 4:100–107, 1968.

[KBS+17] Alexander Kleff, Christian Bräuer, Frank Schulz, Valentin Buchhold, Moritz Baum und Dorothea Wagner: *Time-Dependent Route Planning for Truck Drivers*, Seiten 110–126. Springer International Publishing, Cham, 2017, ISBN 978-3-319-68496-3.

[KSWZ20] Alexander Kleff, Frank Schulz, Jakob Wagenblatt und Tim Zeitz: *Efficient Route Planning with Temporary Driving Bans, Road Closures, and Rated Parking Areas*. In: Simone Faro und Domenico Cantone (Herausgeber): *Proceedings of the 19th International Symposium on Experimental Algorithms (SEA'20)*, 2020. `http://www.sea2020.dmi.unict.it/LIPIcs160/LIPIcs-SEA-2020-17.pdf`.

[Str19] Ben Strasser: *RoutingKit*, 2019. `https://github.com/RoutingKit/RoutingKit`, besucht am 22.02.2021.

[SZ21] Ben Strasser und Tim Zeitz: *A Fast and Tight Heuristic for A* in Road Networks*. In: David Coudert und Emanuele Natale (Herausgeber): *Proceedings of the 19th International Symposium on Experimental Algorithms (SEA'21)*, Band 190 der Reihe *Leibniz International Proceedings in Informatics*, June 2021. To appear.

[vdTdWB18] Marieke van der Tuin, Mathijs de Weerdt und G. Batz: *Route Planning with Breaks and Truck Driving Bans Using Time-Dependent Contraction Hierarchies*. Proceedings of the International Conference on Automated Planning and Scheduling, 28(1), Jun. 2018. `https://ojs.aaai.org/index.php/ICAPS/article/view/13912`.

[Wag19] Jakob Wagenblatt: *Routenplanung mit temporären Straßensperrungen und ortsabhängigen Wartekosten*. Bachelor Thesis, Karlsruhe Institute of Technology (KIT), 2019. `https://i11www.iti.kit.edu/_media/teaching/theses/ba-wagenblatt-19.pdf`.