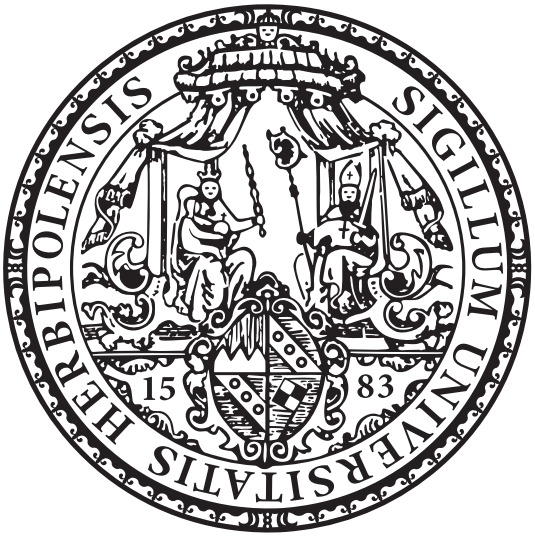
Bachelorarbeit

Tabu Search für das linienbasierte dynamische DARP

Lucas Hein

Abgabedatum:	02. Mai 2024
Betreuer:	Prof. Dr. Marie Schmidt
	Kendra Reiter, M. Sc.



Julius-Maximilians-Universität Würzburg
Lehrstuhl für Informatik I
Algorithmen und Komplexität

Zusammenfassung

Aufgrund geringer Bevölkerungsdichten ist klassischer öffentlicher Verkehr im ländlichen Raum teuer und ineffizient. Die Lösung für dieses Problem könnten sogenannte anfrage-orientierte öffentliche Bussysteme sein. Das zwei Phasen linienbasierte dynamische DARP (2P-LiDyDARP) ist ein solches System, bei dem Busse entlang einer Linie Kundenanfragen bedienen. Diese Arbeit untersucht den Einsatz einer Tabu-Search für die Lösung des 2P-LiDyDARPs. Neue Anfragen werden zuerst mithilfe einer Greedy Strategie in die bestehende Lösung integriert. Danach wird diese Lösung in einem zweiten Schritt durch eine Tabu-Search verbessert. Die entwickelte Tabu-Search wird mithilfe einer Simulation mit zwei Greedy Strategien verglichen. Es zeigt sich, dass die Tabu-Search im Vergleich zu den anderen beiden Strategien mehr Anfragen bearbeiten kann und dabei für die Bearbeitung dieser Anfragen weniger Fahrzeit benötigt. Für die untersuchten Testinstanzen terminierte die Tabu-Search im Schnitt unter einer Sekunde.

Abstract

Due to low population densities, traditional public transport in rural areas is expensive and inefficient. This problem can be solved through so-called demand-responsive public bus systems. The two-phase line-based dynamic DARP (2P-LiDyDARP) is such a system in which buses serve customer requests along a given line. This thesis investigates the use of a tabu-search for solving the 2P-LiDyDARP. When a request gets known, a simple greedy strategy is used for integrating this request into the existing solution. This solution is then improved using a tabu-search. With the help of a simulation, the proposed tabu-search is compared with two greedy strategies. The results indicate that the tabu-search can process more requests while also reducing the time driven by the buses. For the analyzed test instances, the tabu-search terminated in less than one second on average.

Inhaltsverzeichnis

1 Einleitung

Aus einer Befragung des ADACs zur Nutzung des öffentlichen Verkehrs (ÖV) im ländlichen Raum ergab sich, dass fast die Hälfte der Befragten den ÖV so gut wie überhaupt nicht nutzen.

Als Gründe für die Nichtnutzung wurden vor allem fehlende Direktverbindungen, hohe Fahrzeiten, eine zu geringe Taktung und zu hohe Fahrpreise genannt [AAk18].

Aufgrund der geringen Besiedlungsdichte ist klassischer ÖV für den ländlichen Raum teuer und ineffizient.

Nach Vansteenwegen et al. könnten sogenannte anfrage-orientierte öffentliche Bussysteme eine mögliche Lösung für den ÖV im ländlichen Raum sein. Diese Systeme sind nicht zwingend an feste Fahrpläne oder Fahrtrouten gebunden. Die Planung erfolgt stattdessen auf Informationen darüber, wann und wohin Kunden fahren möchten [VMA+22].

Das in diesem Bereich wohl am meisten untersuchte Problem ist das Dial-A-Ride-Problem (DARP). Das DARP bietet einen Haus-zu-Haus-Transportdienst für Personen mit mehreren Fahrgästen an [DS14].

Das eigentliche Problem des DARPs besteht darin, für eine feste Anzahl von Fahrzeugen Fahrzeugrouten und Fahrgastzuweisungen zu ermitteln, sodass eine Menge von Kundenanfragen bestmöglich bedient wird [GKP+23].

Abhängig davon, wann diese Anfragen bekannt sind, heißt ein DARP statisch oder dynamisch [DS14]. Der statische Fall tritt dann auf, wenn alle Anfragen bereits vor der Planung der Routen bekannt sind [DS14]. Bei der dynamischen Problemvariante können zusätzliche Informationen während der Planungsphase oder sogar während der Ausführungsphase offengelegt werden [MBC17]. Das dynamische DARP bietet Kunden mehr Flexibilität, da diese auch noch während des Betriebes Anfragen stellen können.

Das *linienbasierte DARP* (LiDARP) ist ein Sonderfall des DARPs. Es kombiniert Elemente einer klassischen fahrplanbasierten Buslinie mit Elementen eines DARPs.

Beim LiDARP fährt eine Flotte von Bussen entlang einer festen Linie mit Haltestellen. Die Linie darf in beide Richtungen befahren werden. Haltestellen müssen dabei immer in der Reihenfolge angefahren werden, in der sie auf der Linie liegen. Abkürzungen sind nicht erlaubt. Ein Bus darf seine Fahrtrichtung nur wechseln, wenn er aktuell keine Passagiere transportiert.

Das LiDARP kombiniert durch diesen Aufbau die Flexibilität eines DARPs bei der Bestimmung Fahrpläne, behält aber eine gewisse Planbarkeit durch die Beschränkung auf eine Linie bei.

Analog zum DARP gibt es für das LiDARP eine statische und eine dynamische Variante. Das *statische LiDARP* (LiStDARP) und das *dynamische LiDARP* (LiDyDARP). Das LiDyDARP wurde erstmals von Puppe [Pup23b] beschrieben.

Diese Arbeit untersucht das *zwei Phasen LiDyDARP* (2P-LiDyDARP). Das 2P-LiDyDARP kombiniert das LiStDARP mit dem LiDyDARP. In Phase-1 fahren noch keine Busse, Kunden können aber bereits Anfragen stellen. Anfragen, die in dieser Phase eingehen, werden am Ende von Phase-1 als LiStDARP gelöst. In Phase-2 fahren Busse und bedienen Anfragen. Kunden können in dieser Phase weiterhin Anfragen stellen. Phase-2 wird als LiDyDARP betrachtet, da sich hier die Anfragesituation im laufenden Betrieb verändert.

Diese Arbeit untersucht, ob sich eine Tabu-Search für die Lösung des 2P-LiDyDARPs eignet. Dazu wird eine Tabu-Search Strategie entwickelt und durch Simulation mit zwei Greedy Strategien verglichen.

Die vorliegende Arbeit ist wie folgt aufgebaut: Abschnitt 2 definiert das 2P-LiDyDARP formal, Abschnitt 3 gibt eine Literaturübersicht, Abschnitt 4 erklärt den grundlegenden Aufbau einer Tabu-Search, Abschnitt 5 beschreibt die verwendete Simulation und die Funktionsweise der Tabu-Search, Abschnitt 6 stellt die Ergebnisse der Untersuchungen vor und Abschnitt 7 diskutiert die gefundenen Ergebnisse.

2 Problembeschreibung

In diesem Abschnitt wird das *zwei Phasen linienbasierte dynamische DARP* (2P-LiDyDARP) formal definiert.

Gegeben ist eine *Buslinie* L, die k *Haltestellen* $H = \{h_1, \ldots, h_k\}$ bedient. Die Buslinie $L = \{\{h_i, h_{i+1}\} \mid 1 \leq i \leq k-1\}$ besteht aus Kanten, die benachbarte Haltestellen verbinden. Busse müssen immer entlang dieser nicht kreisförmigen festen Linie fahren und dürfen keine anderen Kanten verwenden. Die Buslinie wird in beide Richtungen befahren und bei Bedarf dürfen Busse frühzeitig die Richtung ändern. Jede Verbindung $e \in L$ besitzt eine feste *Fahrzeit* T_e. Daneben gibt es eine feste und für alle Haltestellen gleiche *minimale Haltezeit* t_s.

Weiter ist eine homogene *Flotte* $B = \{b_1, \ldots, b_n\}$ gegeben, die aus n Minibussen besteht. Die Anzahl an Personen, die ein Minibus gleichzeitig transportieren kann, wird als *Kapazität* C bezeichnet.

Kunden können mithilfe von *Anfragen* einen Fahrtwunsch bekunden. Die Menge aller Anfragen wird mit $R = \{r_1, \ldots, r_m\}$ beschrieben, wobei m die Anzahl der Anfragen ist. Die Menge aller Anfragen R teilt sich in zwei disjunkte Teilmengen R_1 und R_2. Die Menge R_1 beinhaltet alle Anfragen, die in Phase-1 und R_2 alle Anfragen, die in Phase-2 gestellt werden. Eine Anfrage $r_i \in R$ mit $1 \leq i \leq m$ besteht aus einer *Starthaltestelle* r_i^o, einer *Zielhaltestelle* r_i^d, einem *Anfragezeitpunkt* r_i^a, einer gewünschten *Abfahrtszeit* r_i^g und einer *spätesten Ankunftszeit* r_i^l. Dem Kunden einer Anfrage wird das Versprechen gemacht, dass er maximal die Zeit $W_{\max}$ an der Starthaltestelle warten muss. Eine Anfrage r muss also bis spätestens $r^g + W_{\max}$ abgeholt werden, sonst zählt sie als abgelehnt. Die späteste Ankunftszeit unter Berücksichtigung von $W_{\max}$ kann berechnet werden als:

$$r^l := r^g + W_{\max} + 2 \cdot T_{\{r^o, r^d\}} \tag{2.1}$$

Das Problem besteht darin, eine möglichst gute Zuordnung von Anfragen zu Bussen zu finden und für jeden Bus einen Fahrplan aufzustellen. Anfragen können bei dieser Zuordnung akzeptiert oder abgelehnt werden.

Eine solche Lösung wird in dieser Arbeit mithilfe von sogenannten *Aktionsplänen* dargestellt. Jeder Bus erhält dabei seinen eigenen Aktionsplan. Ein Aktionsplan A ist eine Liste von Aktionen, die der Bus der Reihe nach ausführt. Eine *Aktion* a besteht aus einem Typ a^{typ}, einer Haltestelle $a^h \in H$, einem Zeitpunkt a^t und einer Menge a^r, die angibt, welche Anfragen von dieser Aktion mitgenommen werden sollen. Beim Typ a^{typ} wird zwischen Ankunfts- und Abfahrtsaktionen unterschieden. Für alle Ankunftsaktionen gilt $a^r = \emptyset$.

Der Betrieb der Buslinie wird in zwei Phasen eingeteilt. In *Phase-1* fahren noch keine Busse. Kunden können aber bereits Anfragen stellen. Eine Anfrage r, die in dieser Phase

gestellt wird, erhält als Anfragezeitpunkt $r^a = -1$. In *Phase-2* fahren Busse und bedienen Anfragen. Kunden können in dieser Phase weiterhin Anfragen stellen. Um Anfragen aus Phase-2 besser planen zu können, muss zwischen dem Anfragezeitpunkt r^a und der gewünschten Abfahrtszeit r^g ein Mindestabstand eingehalten werden. Dieser Mindestabstand wird in dieser Arbeit als MinAnfragefenster bezeichnet. Das MinAnfragefenster ist für alle Anfragen gleich. Anfragen aus Phase-1 müssen diesen Mindestabstand nicht einhalten. Der gewünschte Abfahrtszeitpunkt für Anfragen aus R_1 darf daher bereits zum Zeitpunkt 0 sein.

3 Literatur

Ziel dieses Abschnitts ist es, das 2P-LiDyDARP zu klassifizieren, existierende DARP Lösungsansätze aufzuzeigen und verwandte wissenschaftliche Arbeiten vorzustellen.

Die Übersichtsarbeit von Vansteenwegen et al. [VMA+22] stellt ein Klassifikationsmodell für nachfrageorientierte öffentliche Bussysteme vor. Das 2P-LiDyDARP kann nach diesem Klassifikationsmodell als dynamisch-online, many-to-many und semi-flexibles öffentliches Bus-System eingestuft werden. Wenn sich der Grundfahrplan eines jeden Dienstes, der auf einer Linie oder einem Gebiet betrieben wird, während des Betriebs ändern kann, so gilt ein öffentliches Bus-System als dynamisch-online. Dienste dürfen sich dabei auch ändern, wenn sie bereits in Betrieb sind. Many-to-many beschreibt hingegen, dass Passagiere an allen Haltestellen ein- und aussteigen können. Ein öffentliches Bussystem gilt als semi-flexible, wenn für die verschiedenen Dienste eine Standardroute und ein Standardfahrplan vorgegeben sind, die Busse aber von diesen Standards abweichen können.

Molenbruch et al. [MBC17] geben in ihrer Arbeit einen Überblick über existierende DARP Lösungsverfahren. Die Autoren unterteilen die Verfahren in *exakte* und *approximierende* Methoden. Da das DARP NP-schwer [MBC17] ist, können exakte Verfahren in der Regel nur für relativ kleine Instanzen eingesetzt werden. Die meisten Probleme realistischer Größe werden daher mithilfe approximierender Methoden gelöst [MBC17]. Zu diesen Methoden gehören Local-Search Verfahren. Nach Pirlot ist die grundlegende Idee hinter einer Local-Search die, dass man sich von einer Lösung zu einer anderen in der Nachbarschaft bewegt. Dabei muss die Nachbarschaft einer Lösung als Teilmenge aller Lösungen definiert sein. In jedem Schritt i wird eine neue Lösung s_{i+1} aus der Nachbarschaft von s_i ausgewählt. Für die nächste Lösung s_{i+1} wird hierbei am häufigsten die beste Lösung aus der Nachbarschaft von s_i gewählt. Diese Strategie hat das Problem, dass sie nur schwer aus lokalen Optima entkommen kann [Pir96].

Die Tabu-Search ist eine Variante der Local-Search, die gezielt dafür entworfen wurde, um das Problem der lokalen Optima zu überwinden [Pir96]. Die Tabu-Search wurde auch bereits erfolgreich für die Lösung von DARP Problemen eingesetzt (vergleiche dazu [MBC17]). Deswegen wird die Tabu-Search als Verfahren für diese Arbeit gewählt.

Nachfolgend sollen nun noch einige wissenschaftliche Arbeiten vorgestellt werden, die mit dieser Arbeit verwandt sind.

Gaul et al. [GKS21] stellen einen Rolling-Horizon Ansatz für das dynamische DARP vor. Beim Eintreffen neuer Anfragen wird eine bestehende Lösung durch Lösen einer MILP-Formulierung erweitert. Die vorliegende Arbeit löst zwar kein MILP, erweitert aber ebenfalls beim Eintreffen neuer Anfragen die bestehende Lösung.

Crainic et al. [CMNG05] stellen in ihrer Arbeit einen demand-responsive Busservice vor, bei dem genau ein Bus auf einer kreisförmigen Linie fährt. Kunden können zusätz-

liche Anfragen stellen, um an weiteren optionalen Orten abgeholt zu werden. Für die Lösung dieses Problems wurde unter anderem eine Tabu-Search untersucht, die laut den Autoren gute Lösungen mit begrenztem Rechenaufwand ermitteln konnte. Der Hauptunterschied zwischen dem Problem von Crainic et al. und dem LiDyDARP liegt in der betrachteten Linie. Das LiDyDARP besitzt, anders als das Problem von Crainic et al., eine nicht kreisförmige Linie, die in beide Richtungen befahren werden kann. Auch besitzt das LiDyDARP keine optionalen Haltestellen außerhalb der definierten Linie. In der von Crainic et al. betrachteten Version des Problems wird außerdem nur ein einziger Bus betrachtet. Das LiDyDARP erlaubt hingegen eine ganze Flotte von Bussen.

Attansio et al. [ACGL04] zeigen, wie sich eine bereits existierende sequenzielle Tabu-Search für das statische DARP mithilfe von verschiedenen Parallelisierung-Ansätzen verbessern lässt. Die dynamischen und parallelen Lösungsansätze der Autoren basieren darauf, dass eine statische Lösung des Problems zu Beginn des Planungszeitraums mit allen bis dahin bekannten Anfragen erzeugt wird. Wird eine neue Anfrage bekannt, wird zuerst überprüft, ob eine gültige Lösung mit dieser Anfrage gefunden werden kann. Wird eine solche Lösung nicht gefunden, so wird die Anfrage abgelehnt. Wenn hingegen eine Lösung mit der Anfrage gefunden wird, wird diese Lösung mithilfe der untersuchten Verfahren weiter optimiert. Die Autoren konnten zeigen, dass sich Parallelisierung für Probleme im dynamischen Kontext eignet. Weiter konnten die Autoren feststellen, dass die initiale Lösung nicht besonders relevant zu sein scheint.

Madsen et al. [MRR95] stellen in ihrer Arbeit einen heuristischen Algorithmus für das DARP mit Zeitfenstern vor. Dabei wird aus allen bisher nicht zugeordneten Anfragen eine Anfrage anhand eines Kriteriums, wie die Schwierigkeit, diese Anfrage zu planen, ausgewählt. Für diese Anfrage werden alle Punkte berechnet, an denen diese eingefügt werden kann. Für alle diese potenziellen Lösungen wird das Ergebnis einer Zielfunktion berechnet. Die Lösung, welche die geringste Änderung der Zielfunktion besitzt, wird ausgewählt. Sollte sich herausstellen, dass eine Anfrage nicht eingefügt werden kann, so wird diese Anfrage abgelehnt. Die Autoren zeigten, dass sich ihr Algorithmus aufgrund seiner geringen Berechnungszeit, auch für dynamische Probleme eignet.

Beaudry et al. [BLMN10] stellen einen, in zwei Phasen eingeteilten, Algorithmus für ein dynamisches DARP im Kontext von Patienten-Transporten vor. Die erste Phase nutzt eine einfache Insertion-Heuristik, um eine erste gültige Lösung zu erhalten. Eine Tabu-Search verbessert diese Lösung in einer zweiten Phase. Die Autoren unterscheiden zwei Arten von Tabu-Search Zügen. Inter-Route-Züge verschieben Anfragen aus der aktuellen Route in eine andere Route. Intra-Route-Züge verändern die Bearbeitungsreihenfolge der Anfragen innerhalb einer Route. Die Autoren konnten zeigen, dass sich dieser Algorithmus für dynamische Probleme eignet und dabei gute Ergebnisse liefert.

Puppe [Pup23b] vergleicht in seiner Arbeit eine fahrplanbasierte mit einer anfragebasierten Minibuslinie. Die von Puppe entwickelte anfragebasierte Minibuslinie kann als LiDyDARP aufgefasst werden, wobei sich das Problem auf die dynamische Phase beschränkt. Für dieses LiDyDARP wurde ein Greedyalgorithmus vorgestellt und mithilfe einer eigens entwickelten Simulation evaluiert. Puppe konnte zeigen, dass eine anfragebasierte Minibuslinie, welche seinen Greedyalgorithmus verwendet, in Fällen mit geringer Nachfrage, einer fahrplanbasierten Minibuslinie überlegen ist. Im Greedyalgorithmus von

Puppe wird in jedem Schritt der Simulation überprüft, ob neue Anfragen eingegangen sind. Sollten neue Anfragen planbar sein, so wird jede neue Anfrage der Reihe nach so in bestehende Busrouten integriert, dass die Wartezeit dieser Anfragen minimiert wird. Sollte sich dabei herausstellen, dass eine Anfrage in keine Busroute eingeplant werden kann, so wird diese Anfrage abgelehnt.

4 Tabu-Search

Dieser Abschnitt ist eine kurze Einführung in die Funktionsweise einer Tabu-Search. Die nachfolgende Beschreibung einer Tabu-Search basiert auf der Arbeit von Gendreau [Gen03]. Die Tabu-Search (TS) wurde 1986 von Fred Glover vorgestellt [Gen03]. Sie versucht, lokale Optima einer Local-Search zu überwinden. Um dies zu erreichen, werden in der TS auch Züge erlaubt, die die aktuelle Lösung verschlechtern. Um zu verhindern, dass schon erkundete Lösungen erneut betrachtet werden, wird eine sogenannte Tabu-Liste genutzt. Die Tabu-Liste speichert verbotene Züge, die es ermöglichen würden, zurück zu früheren Lösungen zu gelangen. Ein Zug ist dabei nicht für immer verboten, sondern nur für eine gewisse Zeit.

Die TS benötigt eine initiale Lösung. Diese kann zum Beispiel über eine Heuristik erzeugt werden. Um außerdem Lösungen miteinander vergleichen zu können, wird eine Bewertungsfunktion benötigt, die jeder Lösung des Lösungsraums einen Wert zuweist. Im Folgenden wird angenommen, dass diese Werte minimiert werden sollen.

Zu Beginn der TS wird die initiale Lösung als aktuelle und als aktuell beste Lösung gesetzt. Die Tabu-Liste ist zu Beginn leer. In jedem Schritt der TS wird die Nachbarschaft der aktuellen Lösung erkundet.

Die Nachbarschaft einer Lösung ist eine Teilmenge des Lösungsraums. Die Nachbarschaft enthält all die Lösungen, die durch lokale Transformationen aus der betrachteten Lösung hervorgehen. Diese lokalen Transformationen sind abhängig vom betrachteten Problem.

Aus der Nachbarschaft der aktuellen Lösung wird diejenige Lösung mit dem niedrigsten Wert der Bewertungsfunktion ausgewählt. Diese Lösung wird im Folgenden als bester Nachbar bezeichnet.

Falls der beste Nachbar einen geringeren Wert der Bewertungsfunktion als die aktuell beste Lösung besitzt, so wird der beste Nachbar zur aktuell besten Lösung.

Unbeachtet davon wird der beste Nachbar zur neuen aktuellen Lösung. Die Züge, die aus dem besten Nachbarn die bisherige aktuelle Lösung erzeugen, werden auf die Tabu-Liste gesetzt.

Die Tabu-Suche terminiert, falls das sogenannte Terminierungs-Kriterium erfüllt ist. Das Terminierungs-Kriterium kann etwa eine maximale Anzahl an Iterationen oder eine maximale Berechnungszeit sein.

5 Methodik

5.1 Simulation

Die Simulation ermöglicht es, verschiedene Strategien, im folgenden auch *Transportstrategien* genannt, für das Lösung des 2P-LiDyDARPs, zu testen und miteinander zu vergleichen. Als Ausgangspunkt dient dieser Arbeit das von Leo Puppe entwickelte *Line Based Minibus Simulation* Projekt [Pup23a]. Es wurde für diese Arbeit weiterentwickelt und ergänzt. Der Quellcode der weiterentwickelten Version ist unter [Hei23] zu finden. Änderungen, die durch den Autor dieser Arbeit vorgenommen wurden, werden durch Kommentare im Quellcode hervorgehoben. Nachfolgend wird aufgelistet, welche Anpassungen vorgenommen worden sind:

- Aufteilung in zwei Simulationsphasen
- Testcases
- Entwicklung einer Grafischen-Benutzer-Oberfläche (GUI)
- Ausweitung des Importsystems
- Implementierung eines Exportsystems
- Verbesserung der Codestruktur

Teile der Anpassungen wurden bereits im Rahmen einer wissenschaftlichen Hilfstätigkeit durch den Autor dieser Arbeit vorgenommen. Dazu zählen insbesondere die Aufteilung der Simulationsphasen, die Ausweitung der Import- und Export-Funktion, die Grundlagen der GUI sowie das Refactoring und Testen der Hauptkomponenten der Simulation. Die meisten der Anpassungen wurden im Rahmen dieser Arbeit ausgebaut und verbessert.

Die wichtigste Änderung ist die Aufteilung in zwei Simulationsphasen. Jede Phase besitzt eine eigene Transportstrategie. Dies ermöglicht es, Anfragen in Phase-1 und Phase-2, wie sie in der Problembeschreibung definiert sind, unterschiedlich zu behandeln. Eine Übersicht über den abgeänderten Simulationsablauf ist in Abbildung 5.1 zu sehen. Neben den erwähnten zwei Phasen gibt es zu Beginn noch eine Initialisierungs-Phase, in der die Simulation aufgesetzt werden kann und zum Schluss eine Evaluations-Phase, in der evaluiert werden kann, wie gut die jeweiligen Transportstrategien das Problem gelöst haben.

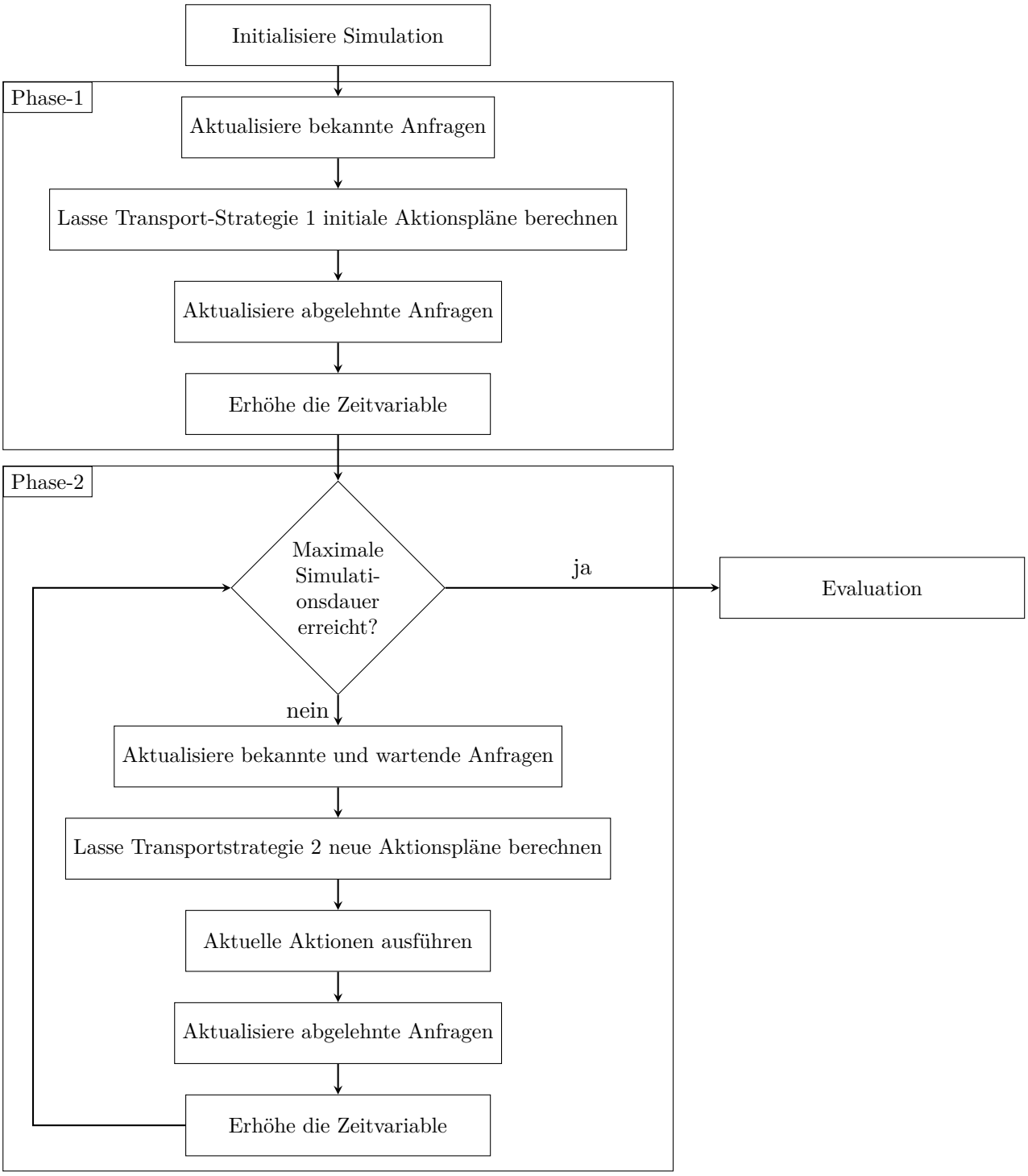


Abb. 5.1: Der Simulationsablauf

Eine weitere wichtige Änderung ist die Einführung von *Testcases*. Ein Testcase ist eine Vorlage für den Aufbau eines Experiments. Jeder Testcase kann seine eigenen Einstellungen definieren. Dies schafft mehr Flexibilität im Aufbau eines Testcases.

Um die Bedienung des Simulationstools zu verbessern, wurde eine *Grafische-Benutzer-Oberfläche* (GUI) entwickelt. Die GUI unterteilt sich in die Reiter:

- Simulation
- RouteGeneration
- RequestGeneration
- Examples
- Visualization

Vergleiche dazu Abbildung 5.2.

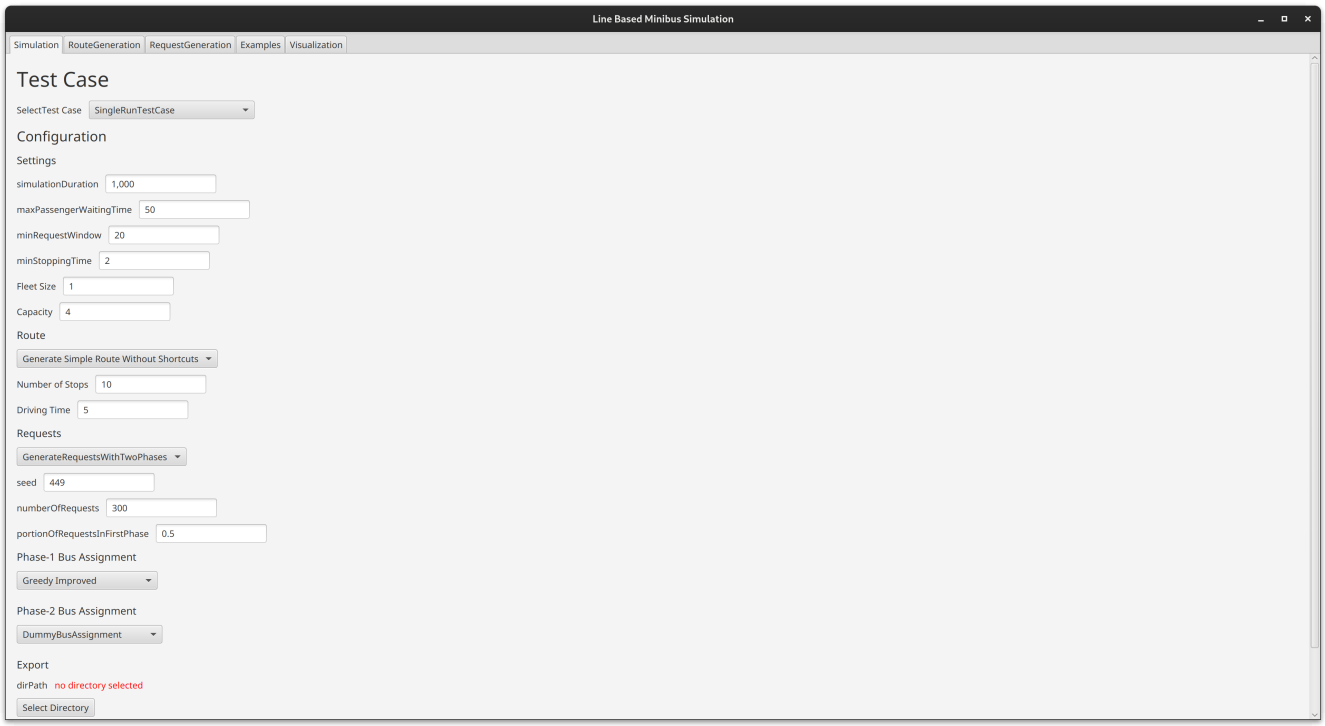


Abb. 5.2: Grafische-Benutzer-Oberfläche des Simulationstools

Die wichtigsten Funktionen der GUI sind unter dem Reiter *Simulation* zu finden. Dort können Testcases ausgewählt, konfiguriert und gestartet werden.

Die Reiter *RouteGeneration* und *RequestGeneration* ermöglichen es, bestehende Anfragen und Streckengeneratoren zu nutzen, um Routen oder Strecken zu generieren und in eine Datei zu speichern. Diese generierten Routen und Anfragen können mithilfe eines

Importmoduls in einen Testcase geladen werden. Das bestehende Importsystem wurde dazu um das Importieren von Routen erweitert. Diese Schnittstellen erlauben es auch, Routen- oder Anfragedaten aus anderen Quellen in das Simulationsmodell zu laden.

Für die Visualisierung kommt das von Puppe [Pup23b] entwickelte Strecke-Zeit-Diagramm zum Einsatz. Ein Beispiel für ein solches Diagramm ist in Abbildung 5.3 zu sehen. Auf der x-Achse werden die Haltestellen und auf der y-Achse die Zeit in Simulationsticks aufgetragen. Die Fahrtstrecke eines Busses wird als farbige Linie dargestellt. Dabei erhält jeder Bus seine eigene Farbe. Die Zahlen neben den Fahrlinien symbolisieren, wie viele Kunden der Bus auf diesem Abschnitt transportiert. Anfragen werden in diesem Diagramm als vertikale Balken auf der vertikalen Linie ihrer Starthaltestelle dargestellt. Die Zielhaltestelle der Anfrage wird mit einem lila Pfeil neben der Anfrage aufgezeichnet. Der vertikale Balken einer Anfrage unterteilt sich in drei Abschnitte. Der lila Abschnitt steht für die Zeit zwischen der Anfragezeit und der gewünschten Abfahrtszeit. Der dunkelblaue Abschnitt markiert die Zeit zwischen der gewünschten Abfahrtszeit und dem spätesten Abfahrtszeitpunkt. Der hellblaue Abschnitt steht für die Zeit nach dem spätesten Abfahrtszeitpunkt und dem spätesten Ankunftszeitpunkt. Zusätzlich sind der gewünschte und der späteste Abfahrtszeitpunkt mit einem gelben Strich markiert.

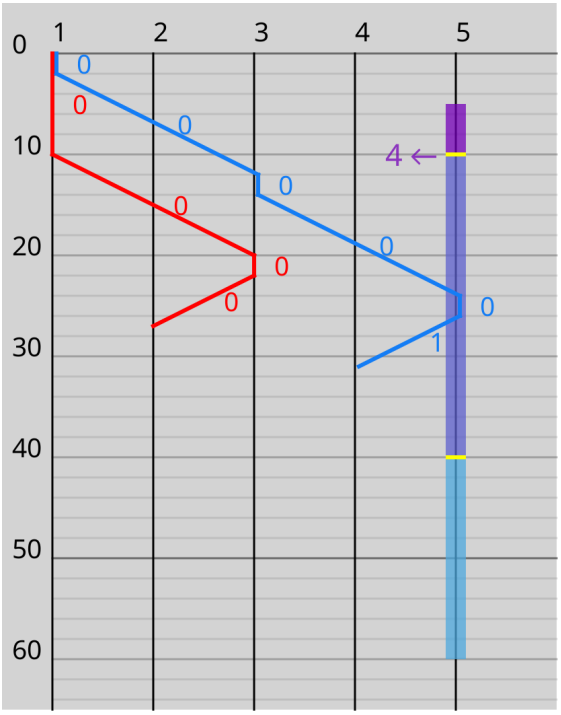


Abb. 5.3: Beispiel für ein Strecke-Zeit-Diagramm

Um die Strecke-Zeit-Diagramme besser nutzen zu können, wurden in der GUI die Reiter *Examples* und *Visualization* hinzugefügt. Im Reiter Examples lassen sich vorgefertigte Beispiele in das Strecke-Zeit-Diagramm laden. Ein Beispiel definiert Anfragen, Busse und die zu den Bussen gehörigen Aktionspläne in Code. Mithilfe dieser Funktion

lassen sich leicht spezielle Simulationszustände visualisieren. Unter dem Reiter Visualization lassen sich hingegen echte Simulationszustände laden. Aktuell können innerhalb der Simulation nur Endzustände exportiert werden, eine Erweiterung auf Zwischenzustände ist jedoch möglich. In beiden Reitern öffnet sich nach dem Laden der Diagramm-Editor (siehe Abbildung 5.4). Der Diagramm-Editor basiert auf der ursprünglichen Implementierung von Puppe. Dieser ursprünglichen Implementierung wurden einige Funktionen ergänzt. Zuerst wurde eine Screenshot-Funktion hinzugefügt, die es ermöglicht, das Zeit-Weg-Diagramm als Bilddatei zu exportieren. Weiter ist es in der überarbeiteten Version möglich, einzelne Busse und Anfragen ein- und auszublenden. Weiter wurde ein Menü eingefügt, in dem einige Optionen für das Diagramm gesetzt werden können. Um einzelne Streckenabschnitte hervorheben zu können, wurde die Möglichkeit eingefügt, mit einem Mausklick einzelne Abschnitte gestrichelt und/oder fett darstellen zu lassen.

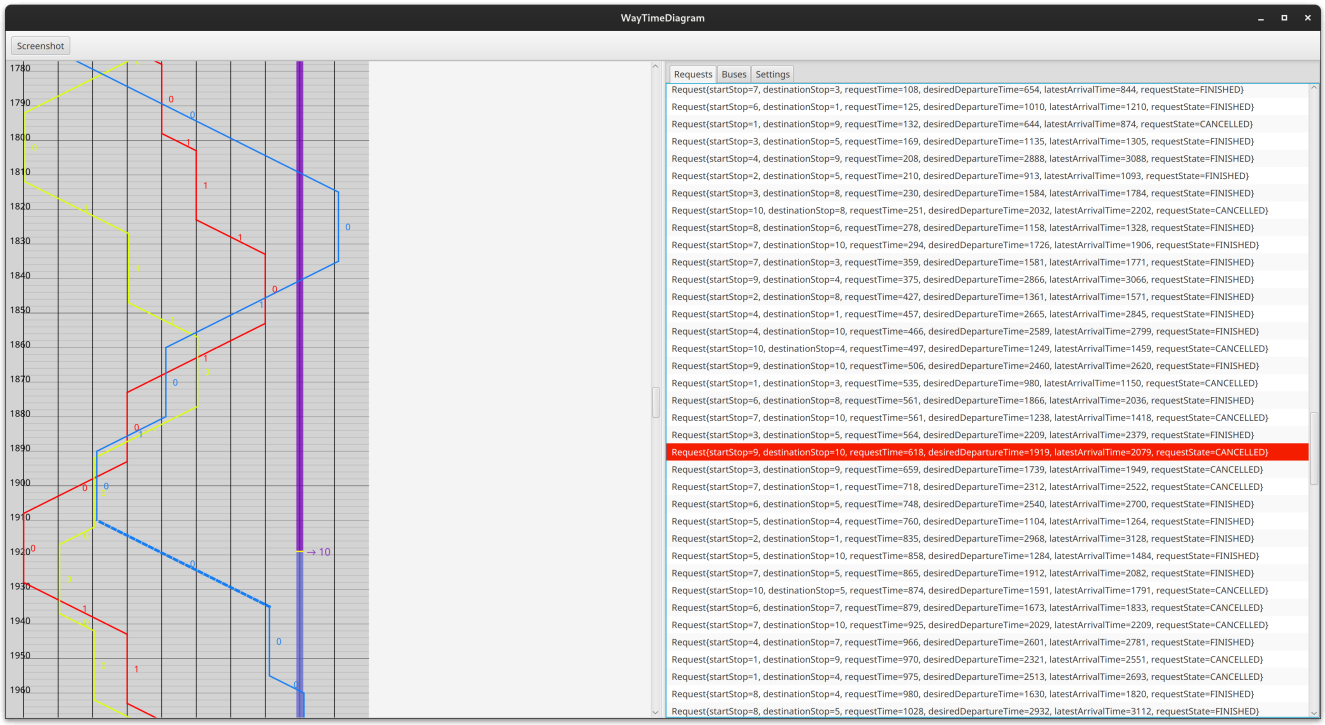


Abb. 5.4: Diagramm-Editor des Simulationstools

Um die Ergebnisse der Simulation festhalten zu können, wurde ein umfangreiche *Exportsystem* entwickelt. Dies ist insbesondere für die Analyse und Dokumentation der Ergebnisse relevant. Für einen Simulationsdurchlauf können folgende Dinge exportiert werden:

- Eine Übersicht aller Busse
- Die Ergebnisse der Evaluation
- Eine Übersicht aller Events

- Eine Übersicht aller Anfragen
- Die verwendete Route
- Eine Übersicht über der verwendeten Einstellungen
- Der Endzustand der Simulation

Der vollständige Export erfolgt nur für den Testcase *SingleRunTestCase*, der genau einen Simulationsdurchlauf simuliert. Dies verhindert eine „Überflutung“ mit Daten. Testcases, die mehr als einen Simulationsdurchlauf durchführen, exportieren nur ihre Einstellungen und die Ergebnisse der Evaluation.

Neben diesen sichtbaren Änderungen wurde auch die eigentliche Codestruktur verändert. Hier soll insbesondere die grundlegende Änderung des Simulationszustands hervorgehoben werden.

In der bisherigen Implementierung waren Simulationslogik und Simulationszustand in einer Klasse vereint. In der neuen Version wurden daraus die beiden Klassen *Simulation* und *SimulationData* gemacht. Dieser Schritt verbessert die Lesbarkeit des Codes und hilft, den Zugriff auf den Simulationszustand besser einzuschränken.

Um zu verhindern, dass eine Transportstrategie eigenständig, ohne vorherige Validierung, den Simulationszustand verändern kann, werden in der neuen Version Proxy-Objekte eingesetzt, die den Zugriff auf den Zustand einschränken.

Die erste Version des Simulationstools übergibt der Transportstrategie ein Simulations-Interface, bei dem keine Methoden für die Veränderung des Zustands offengelegt werden. Dieses System kann jedoch durch geschicktes Umwandeln des Interfaces in die eigentliche Simulations-Klasse umgangen werden. Durch diesen Zugriff auf den Simulationszustand könnte es zu invaliden Simulationsergebnissen kommen. Auch alle weiteren Objekte, die das Simulation-Interface übergeben bekommen, könnten so den Simulationszustand verändern.

In der neuen Version gibt es für jedes Objekt, das Daten des Simulationszustands hält, eine veränderbare und eine unveränderbare Version. Die veränderbare und unveränderbare Version implementieren dabei ein gemeinsames Interface. In diesem Interface werden sowohl Funktionen offengelegt, die den Zustand auslesen können, sowie Funktionen, die den Zustand verändern können. Um zu verhindern, dass in der unveränderlichen Version Daten geändert werden, wirft der Zugriff auf eine Funktion, die Daten verändert, in dieser Version eine Fehlermeldung. Die unveränderliche Version erhält bei ihrer Erstellung eine Referenz auf eine veränderbare Version. Der Aufruf einer Lesefunktion der unveränderbaren Variante wird an die referenzierte veränderbare Version weitergegeben. Die unveränderbare Version enthält also selbst keine Simulationsdaten und fungiert nur als Proxy, der den Zugriff auf die Daten einschränkt. Dieses Vorgehen ähnelt der Implementierung der *Unmodifiable Collections* in Java [jav].

Um außerdem zu verhindern, dass Daten über eine Referenz, welche über eine Lesefunktion erhalten wurde, verändert werden können, wird immer die unveränderbare Version einer Klasse zurückgegeben. Für Collections, wie Listen, werden immer die unveränderbaren Varianten aus der Standardbibliothek verwendet.

In der neuen Version erhält nur noch die Simulation selbst eine Referenz auf die veränderbare Version des Simulationszustands. Alle anderen Klassen erhalten nur noch die unveränderbare Version. Um Daten ändern zu können, muss nun immer über die Simulation gegangen werden, die dabei die nötigen Validierung vornehmen kann.

Dieses System soll nachfolgend am Beispiel der Klassen `BusInterface`, `Bus` und `ImmutableBus` demonstriert werden. Ein Auszug des Source-Codes zu diesen drei Klassen ist in den Listings 5.1 - 5.3 zu sehen. Das `BusInterface` definiert alle Methoden, die ein Bus besitzt. Hier auszugsweise die beiden Methoden `getPreviousActions()` und `setNextActions(...)`.

Die Klassen `Bus` und `ImmutableBus` implementieren dieses `BusInterface`. Wobei die eingentliche Funktionalität der beiden Methoden `getPreviousActions()` und `setNextAction(...)` in der Klasse `Bus` implementiert wird. Eine Instanz der Klasse `ImmutableBus` kann über die Methode `of(...)` erstellt werden. Diese erhält eine Referenz zu einer `Bus`-Instanz.

Ein Aufruf der Methode `getPreviousActions()` der Klasse `ImmutableBus` wird an diese `Bus`-Instanz weitergereicht. Wird hingegen die Methode `setNextAction(...)` der Klasse `ImmutableBus` aufgerufen, wird eine Fehlermeldung geworfen.

```java
public interface BusInterface {

    ...

    List<ImmutableAction> getPreviousActions();

    ...

    void setNextActions(List<Action> nextActions);

    ...
}
```

Listing 5.1: BusInterface Klasse

```java
public class Bus implements BusInterface, ... {

    ...

    private final Route route;

    private final List<Action> nextActions;
    private final List<Action> previousActions;
    private final List<Request> currentlyTransportedRequests;
    private final int capacity;

    public Bus(Route route, ImmutableStop stop, int capacity) {
        this.route = route;
        this.nextActions = new LinkedList<>();
        this.previousActions = new LinkedList<>();
        this.previousActions.add(new ArrivalAction(stop, 0));
        this.currentlyTransportedRequests = new LinkedList<>();
        this.capacity = capacity;
    }

    ...

    @Override
    public List<ImmutableAction> getPreviousActions() {
        return ImmutableAction.ofList(previousActions);
    }

    ...

    @Override
    public void setNextActions(List<Action> nextActions) {
        this.nextActions.clear();
        this.nextActions.addAll(nextActions);
    }

    ...
}
```

Listing 5.2: Bus Klasse

```java
public class ImmutableBus implements BusInterface, ... {

    ...

    private final Bus bus;

    public static ImmutableBus of(Bus bus) {
        return new ImmutableBus(bus);
    }

    public static List<ImmutableBus> ofList(List<Bus> buses) {
        return buses.stream().map(ImmutableBus::of).toList();
    }

    private ImmutableBus(Bus bus) {
        this.bus = bus;
    }

    ...

    @Override
    public List<ImmutableAction> getPreviousActions() {
        return bus.getPreviousActions();
    }

    ...

    @Override
    public void setNextActions(List<Action> nextActions) {
        throw new UnsupportedOperationException(
    ACCESS_NOT_ALLOWED_ERROR_MESSAGE);
    }

    ...
}
```

Listing 5.3: ImmutableBus Klasse

Um Fehler im Code besser finden zu können und um mehr Sicherheit bezüglich der Funktionsweise des Codes zu erhalten, wurden im Rahmen der Weiterentwicklung und Implementierung neuer Transportstrategien mehr als 200 Unit-Tests geschrieben.

5.1.1 Anfragengenerierung

Für diese Arbeit wurde ein eigener *Fahrtanfragengenerator* entwickelt. Dieser generiert Anfragen für Phase-1 und Phase-2. Die Gesamtanzahl der Anfragen m und der Anteil R_{ratio} von m, der in Phase-1 liegt, können konfiguriert werden. Für die Anzahl der Anfragen in beiden Phasen gilt $|R_1| = \lfloor R_{\text{ratio}} \cdot m \rfloor$ und $|R_2| = m - |R_1|$.

Im Folgenden sei $t_{\max}$ die Simulationsdauer. Alle Anfragen werden so generiert, dass sie bis zum Ende der Simulation bearbeitet werden können. Um dies sicherzustellen, werden für die Anfragezeit und die gewünschte Abfahrtszeit Maxima festgelegt, die es ermöglichen, eine Anfrage bis zum Ende der Simulation zu bearbeiten. Versucht eine Transportstrategie trotzdem eine Anfrage so zu planen, dass sie erst nach $t_{\max}$ an ihrer Zielhaltestelle ankommt, so wird die Simulation einen Fehler werfen.

Die Maxima LetzteAbfahrtszeit und LetzterAnfragezeitpunkt sind definiert als:

$$\text{LetzteAbfahrtszeit} \coloneqq t_{\max} - (2 \cdot T_{\{h_1, h_k\}}) \tag{5.1}$$

$$\text{LetzterAnfragezeitpunkt} \coloneqq \text{LetzteAbfahrtszeit} - \text{MinAnfragefenster} \tag{5.2}$$

Eine Anfrage aus $r_1 \in R_1$ wird wie folgt erzeugt. Alle im Folgenden betrachteten Intervalle sind diskret. Zuerst wird eine Starthaltestelle r_1^o und eine Zielhaltestelle r_1^d gewählt. Das Verfahren für die Auswahl der Haltestellen entspricht dem Verfahren aus [Pup23b]. Zuerst wird ein Index l für r_1^o aus dem Intervall $[1, k]$ gleichverteilt gezogen. Ein weiterer Index p für r_1^d wird ebenfalls gleichverteilt aus $[1, k-1]$ gewählt. Sollte $p \geq l$ sein, so wird $p \coloneqq p+1$ gesetzt. Dieses Verfahren gewährleistet, dass für die Start- und Zielhaltestelle nicht die gleiche Haltestelle gewählt wird. Für den Anfragezeitpunkt wird immer $r^a \coloneqq -1$ gewählt. Der Wert für die gewünschte Abfahrtszeit r_1^g wird gleichverteilt aus dem Intervall $[1, \text{LetzteAbfahrtszeit}]$ gezogen.

Für eine Anfrage $r_2 \in R_2$ wird mit dem gleichen Verfahren wie in Phase-1 die Starthaltestelle r_2^o und r_2^d bestimmt. Für die Anfragezeit r_2^a wird ein Wert aus $[1, \text{LetzterAnfragezeitpunkt}]$ gleichverteilt gewählt. Die Anfrage r_2 kann frühstens zum Zeitpunkt r_2^a+MinAnfragefenster abfahren. Dieser Zeitpunkt wird im Folgenden als frühster Abfahrtszeitpunkt FrühsteAbfahrtszeit bezeichnet. Für den gewünschten Abfahrtszeitpunkt r^g wird ein Wert aus

$$[\text{FrühsteAbfahrtszeit}, \text{LetzteAbfahrtszeit}] \tag{5.3}$$

gleichverteilt gewählt. Die Werte für den spätesten Ankunftszeitpunkt werden in beiden Phasen mithilfe der Gleichung 2.1 bestimmt.

Für die Erzeugung der Zufallszahlen kommt der Pseudozufallsgenerator von Java zum Einsatz. Dieser erlaubt es, einen sogenannten *Seed* anzugeben, der für die Erzeugung der Zufallszahlen genutzt wird. Dieser Seed ermöglicht es, die Zahlenfolge des Pseudozufallsgenerators zu reproduzieren. Um für unterschiedliche Testszenarien die gleichen Anfragen generieren zu können, kann ein solcher Seed dem Fahrtanfragengenerator übergeben werden.

5.2 Transportstrategien

Die Transportstrategien sind für die Erzeugung der Aktionspläne verantwortlich. Phase-1 und Phase-2 besitzen jeweils ihre eigene Transportstrategie. Die Transportstrategie aus Phase-1 erhält zu Beginn einmalig die Möglichkeit nach der Aktualisierung aller bekannten Anfragen Aktionspläne zu planen. Vergleiche dazu Abbildung 5.1. Die Transportstrategie der Phase-2 erhält in jedem Simulationszyklus die Möglichkeit, neue Aktionspläne zu planen. Vergleiche auch hierfür Abbildung 5.1. Die Transportstrategie in Phase-2 muss bei der Planung der Aktionspläne gewisse Regeln einhalten, damit bereits ausgeführte Aktionen mit den neuen Aktionsplänen kompatibel bleiben. Die Anforderungen für die Transportstrategien der beiden Phasen unterscheiden sich. Die Berechnungszeit von Transportstrategien in Phase-1 ist nicht so ausschlaggebend, da man diese nur ein einziges Mal ausführen muss und sich das Problem nicht dynamisch während der Berechnung ändern kann. Transportstrategien in Phase-2 hingegen müssen schnell eine Lösung finden, da sich hier das Problem jederzeit ändern kann und weil es dazu kommen kann, dass Busse im laufenden Betrieb umgeplant werden müssen. Transportstrategien aus Phase-2 erhalten zu Beginn eine Liste mit allen Aktionsplänen aus Phase-1 und eine Liste mit allen Anfragen, die in Phase-1 bereits abgelehnt wurden.

Im Folgenden sollen zuerst Operationen beschrieben werden, die Aktionspläne verändern können. Danach werden die eigentlichen Transportstrategien dieser Arbeit beschrieben, die auf den vorher definierten Operationen aufbauen.

5.2.1 Operationen auf Aktionsplänen

Für diese Arbeit werden drei Operationen definiert, die einen Aktionsplan verändern können. Diese Operationen sind die *Insert-Operation*, die *Delete-Operation* und die *Move-Operation*.

Um über die Arbeitsweise der Operationen sprechen zu können, muss kurz das Konzept der *Sublinie* eingeführt werden. Wie bereits in Abschnitt 2 beschrieben, definiert sich eine Lösung des 2P-LiDyDARPs über die Aktionspläne der Busse. Der Aktionsplan eines Busses $b \in B$ setzt sich aus einer Aneinanderreihung von Sublinien zusammen. Eine *Sublinie* besteht immer aus Aktionen, bei denen der Bus b in die gleiche Richtung fährt und nicht die Richtung ändert. Nach jedem Richtungswechsel von b beginnt eine neue Sublinie. Sublinien müssen mit einer Abfahrtsaktion starten und mit einer Ankunftsaktion enden. Eine Sublinie besteht aus allen Aktionen, die zwischen zwei Richtungswechseln liegen.

Eine Operation kann entweder erfolgreich den Aktionsplan verändern oder fehlschlagen. Wann eine Operation fehlschlägt, hängt vom Typ der Operation ab und wird im Folgenden in der detaillierten Beschreibung der Operationen vorgestellt.

Insert-Operation

Die *Insert-Operation* versucht einem Bus $b \in B$ eine neue Anfrage r hinzuzufügen. Die neue Anfrage muss also in den Aktionsplan A für diesen Bus integriert werden.

Für das Einfügen von r in b werden zwei Varianten unterschieden. Zuerst kann eine bestehende Sublinie abgeändert werden, sodass sie die neue Anfrage bearbeitet. Dabei kann es dazu kommen, dass bisherige Aktionen verzögert werden müssen. Zweitens kann eine neue Fahrt geplant werden, welche die Anfrage bearbeitet. Eine Fahrt besteht dabei immer aus mindestens einer neuen Sublinie oder verlängert zwei bestehende Sublinien.

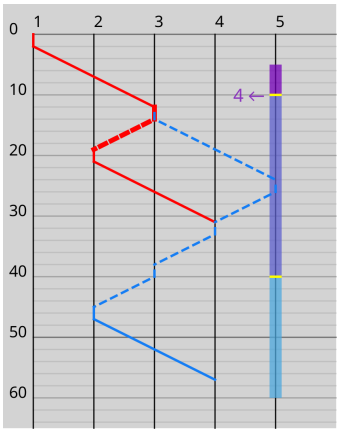
Nachfolgend wird zuerst beschrieben, wie diese beiden Varianten funktionieren. Danach wird erklärt, wie diese Varianten genutzt werden, um eine Anfrage in einen Bus einzufügen.

Die *erste Haltestelle* einer Sublinie ist definiert als die Haltestelle der ersten Abfahrtsaktion der Sublinie. Die *letzte Haltestelle* einer Sublinie ist hingegen definiert als die Haltestelle der letzten Ankunftsaktion in der Sublinie. Eine Haltestelle befindet sich vor einer Sublinie, wenn sie der Fahrtrichtung der Sublinie folgend vor der ersten Haltestelle der Sublinie in der Buslinie liegt. Ähnlich gilt eine Haltestelle innerhalb der Sublinie, wenn sich die Haltestelle zwischen der ersten und letzten Haltestelle der Sublinie befindet oder selbst die Start- oder Zielhaltestelle ist. Weiter befindet sich eine Haltestelle nach einer Sublinie, wenn sie sich der Fahrtrichtung folgend nach der letzten Haltestelle der Sublinie befindet.

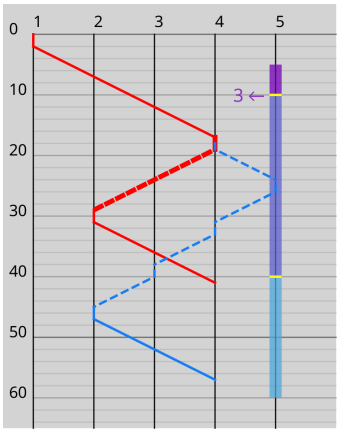
Beim Einfügen einer Anfrage kann für die Start- und Zielhaltestelle dieser Anfrage unterschieden werden, wie diese bezüglich einer bestehenden Sublinie liegen. Dabei werden folgende Fälle unterschieden:

1. Die Start- und Zielhaltestelle befinden sich vor der Sublinie.
2. Die Starthaltestelle befindet sich vor und die Zielhaltestelle innerhalb der Sublinie.
3. Die Starthaltestelle befindet sich vor und die Zielhaltestelle nach der Sublinie.
4. Die Start- und Zielhaltestelle befinden sich innerhalb der Sublinie.
5. Die Starthaltestelle befindet sich innerhalb und die Zielhaltestelle nach der Sublinie.
6. Die Start- und die Zielhaltestelle befinden sich nach der Sublinie.

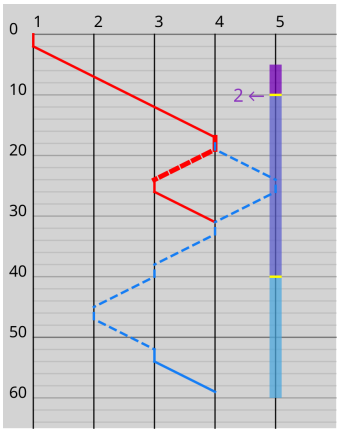
Für jeden dieser Fälle wird in Abbildung 5.5 ein Beispiel gezeigt. Dabei zeigt die rote Linie den Aktionsplan vor dem Einfügen. Die rote gestrichelte Linie zeigt hierbei die Sublinie, in die eingefügt werden soll. Die Änderung des Aktionsplans wird durch die blaue Linie dargestellt. Dabei ist der gestrichelte Teil der blauen Linie der Teil des Aktionsplans, der aus der Sublinie hervorgeht. Folgen nach der Sublinie Aktionen, so stellt der durchgezogene Teil der blauen Linie diese dar.



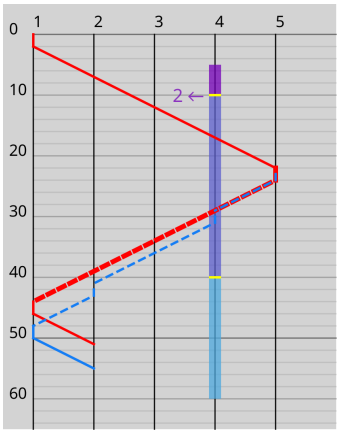
(a) Die Start- und Zielhaltestelle befindet sich vor der Sublinie.



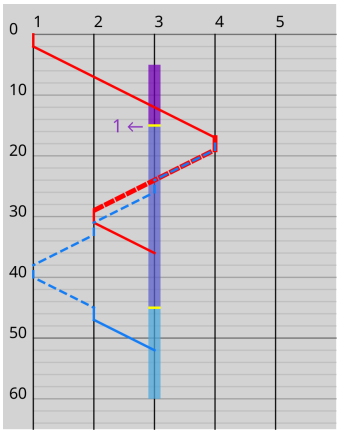
(b) Die Starthaltestelle befindet sich vor und die Zielhaltestelle innerhalb der Sublinie.



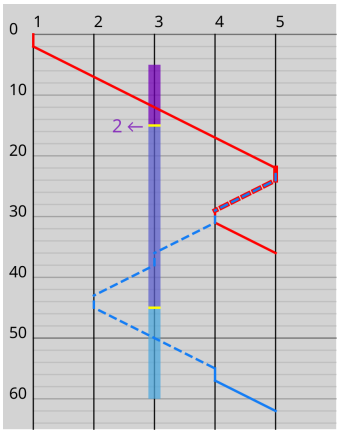
(c) Die Starthaltestelle befindet sich vor und die Zielhaltestelle innerhalb der Sublinie.



(d) Die Start- und Zielhaltestelle befindet sich innerhalb der Sublinie.



(e) Die Starthaltestelle befindet sich vor und Zielhaltestelle befindet sich innerhalb der Sublinie.



(f) Die Start- und Zielhaltestelle befindet sich nach der Sublinie.

Abb. 5.5: Beispiele für die verschiedenen Fälle, die beim Einfügen in eine bestehende Sublinie auftreten können.

Beim Einfügen einer Anfrage in eine Sublinie wird die Sublinie so abgeändert, dass sie die Start- und Zielhaltestelle der Anfrage anfährt und an diesen hält, um den Kunden dieser Anfrage einsteigen und aussteigen zu lassen. Der beim Einfügen entstehende abgeänderte Teil des Aktionsplans muss selbst keine Sublinie nach obiger Definition darstellen. Das trifft zum Beispiel auf die Änderungen (in Blau dargestellt) aus Abbildung 5.5a zu. Die Änderungen des Aktionsplans bestehen aus einer Sublinie, die bei Haltestelle 3 startet und bei Haltestelle 5 endet und aus einer Sublinie, die bei Haltestelle 5

startet und bei Haltestelle 2 endet.

Das Einfügen der Start- und Zielhaltestellen kann außerdem dazu führen, dass der Teil des Aktionsplans, der nach der Sublinie folgt, verzögert werden muss.

Im Folgenden wird für jeden Fall erklärt, wie der Aktionsplan A von $b \in B$ verändert wird, um eine Anfrage r in eine Sublinie *sub* dieses Aktionsplans einzufügen. Dabei werden für jeden Fall neue Aktionen $a_1, a_2, \ldots$ definiert und erklärt, wie diese in A integriert werden. Für den Fall, dass eine der Haltestellen r^o oder r^d innerhalb von *sub* liegt, muss überprüft werden, ob b bereits innerhalb von *sub* dort hält, oder ob ein neuer Halt eingeplant werden muss.

Fall 1: Im ersten Fall befinden sich die Starthaltestelle r^o und die Zielhaltestelle r^d vor *sub*. Alle Aktionen müssen in diesem Fall direkt vor *sub* in A eingefügt werden. Das Vorgehen in diesem Fall sieht immer wie in Abbildung 5.5a aus. Die erste Aktion a_1 ist eine Abfahrtsaktion an der ersten Haltestelle von *sub*. Die Abfahrtszeit a_1^t entspricht der Ankunftszeit der letzten Aktion vor der Sublinie plus die minimale Standzeit t_s. Die zweite Aktion a_2 ist eine Ankunftsaktion an r^o. Für die Ankunftszeit wird $a_2^t \coloneqq a_1^t + T_{\{a_1^h, a_2^h\}}$ gesetzt. Sie wird also genau so gewählt, dass sie die Fahrzeit zwischen der ersten Haltestelle von *sub* zur Haltestelle r^o abbildet. Die dritte Aktion ist eine Abfahrtsaktion a_3 mit $a_3^r \coloneqq \{r\}$. Für den Abfahrtszeitpunkt a_3^t wird $\max\{a_2^t + t_s, r^g\}$ gewählt. Der Bus wartet also an r^o solange, bis r abfahrbereit ist. Die vierte Aktion a_4 ist eine Ankunftsaktion an r^d. Für den Ankunftszeitpunkt a_4^t wird $a_4^t \coloneqq a_3^t + T_{\{r^o, r^d\}}$ gewählt. Die fünfte Aktion a_5 ist eine Abfahrtsaktion von r^d mit $a_5^r \coloneqq \emptyset$. Die Abfahrtszeit a_5^t wird auf $a_4^t + t_s$ gesetzt. Die sechste Aktion a_6 ist eine Ankunftsaktion an der ersten Haltestelle von *sub*. Die Ankunftszeit von a_6 ist die Zeit a_5^t plus die Fahrzeit zwischen r^d und der ersten Haltestelle von *sub*. Alle Aktionen nach a_6 müssen unter Umständen verzögert werden. Dafür wird der erste Nachfolger a_* betrachtet. Falls $a_*^t < a_6^t + t_s$ so wird a_* um die Zeit $a_6^t + t_s - a_*^t$ verzögert. Wenn a_* verzögert wurde, so werden auch alle Nachfolger um die Zeit $a_6^t + t_s - a_*^t$ verzögert.

Fall 2: Im zweiten Fall befindet sich die Starthaltestelle r^o vor und die Zielhaltestelle r^d innerhalb der Sublinie *sub*. Für die Starthaltestelle muss eine neue Fahrt eingeplant werden, die r^o besucht und zurück zur ersten Haltestelle von *sub* fährt. Bei der Zielhaltestelle muss hingegen überprüft werden, ob der Bus bereits an r^d oder ob er nur an r^d vorbeifährt.

Für die Fahrt zur Starthaltestelle r^o müssen vier neue Aktionen $a_1, \ldots, a_4$ vor die bisherige Sublinie *sub* in den Aktionsplan A eingefügt werden. Die erste Aktion a_1 ist eine Abfahrtsaktion an der ersten Haltestelle von *sub* mit $a_1^t \coloneqq a_0^t + t_s$, wobei a_0 die unmittelbare Vorgängeraktion bezeichnet. Die zweite Aktion a_2 ist eine Ankunftsaktion an r^o mit $a_2^t \coloneqq a_1^t + T_{\{a_1^h, r^o\}}$. Die dritte Aktion a_3 ist eine Abfahrtsaktion von r^o mit $a_3^r \coloneqq \{r\}$. Für diese Aktion wird, analog zum ersten Fall, der frühestmögliche Abfahrtszeitpunkt gesucht und a_3^t zugewiesen.

Im Fall, dass r^o bereits Teil von *sub* ist, müssen keine weiteren neuen Aktionen in den Aktionsplan eingefügt werden. Alle bisherigen Aktionen von *sub* und alle Aktionen, die nach *sub* folgen, müssen analog zum Verzögerungsverfahren aus dem ersten Fall verzögert werden.

Für den Fall, dass an r^d bereits gehalten wird und r_o nicht die erste Haltestelle der Sublinie *sub* ist, müssen zwei neue Aktionen a_5 und a_6 geplant werden. Um diese Aktionen einfügen zu können, wird zuerst die letzte Abfahrtsaktion a_* gesucht, bevor der Bus an r^d vorbeifährt. Nach dieser Aktion werden die Aktionen a_5 und a_6 in den Aktionsplan eingefügt. Die Aktion a_5 ist eine Ankunftsaktion an r^d zum Zeitpunkt $a_*^t + T_{\{a_*^h, r^d\}}$. Weiter ist a_6 die dazugehörige Abfahrtsaktion von r^d. Dabei wird $a_6^t \coloneqq a_5^t + t_s$ und $a_6^r \coloneqq \emptyset$ gesetzt. Die Ankunftszeit der Ankunftsaktion, welche in der bisherigen Sublinie *sub* auf die a_* folgt, muss in jedem Fall um die zusätzliche Haltezeit an r^d verzögert werden. Alle weiteren Aktionen werden im Bedarfsfall mit dem Verfahren aus Fall 1 verzögert.

Fall 3: Im dritten Fall befinden sich die Starthaltestelle r^o vor und die Zielhaltestelle r^d nach der Sublinie *sub*. Für diesen Fall muss zuerst eine neue Fahrt zu r^o geplant werden. Diese Fahrt besteht aus den gleichen vier Aktionen $a_1, \ldots, a_4$ wie die Fahrt zur Starthaltestelle aus dem zweiten Fall. Diese Aktionen werden ebenfalls vor der bestehenden Sublinie *sub* in den Aktionsplan eingefügt und alle nachfolgenden Aktionen werden nach Bedarf verzögert. Für die Fahrt zur Zielhaltestelle werden zwei Aktionen a_5 und a_6 nach den Aktionen von *sub* eingefügt. Bei a_5 handelt es sich um eine Abfahrtsaktion, welche von der letzten Haltestelle der Sublinie *sub* abfährt. Es gilt $a_5^r \coloneqq \emptyset$ und als Abfahrtszeitpunkt a_5^t wird die Ankunftszeit der letzten Aktion innerhalb der Sublinie plus die minimale Wartezeit t_s gewählt. Die Aktion a_6 ist eine Ankunftsaktion an r^d. Die Ankunft erfolgt zum Zeitpunkt $a_6^t \coloneqq a_5^t + T_{\{a_5^h, r^d\}}$. Sollten sich weitere Aktionen nach *sub* im Aktionsplan befinden, müssen zwei weiteren Aktionen a_7 und a_8 eingefügt werden. Diese beiden Aktionen verbinden die Fahrt zur Zielhaltestelle mit der auf die Sublinie *sub* folgende Sublinie. Alle weiteren Aktionen müssen mit dem Verfahren aus dem ersten Fall bei Bedarf verzögert werden.

Fall 4: Im vierten Fall befinden sich sowohl die Starthaltestelle r^o als auch die Zielhaltestelle r^d innerhalb der Sublinie *sub*. Für den Fall, dass der Bus bereits an r^o hält, muss die Abfahrtsaktion dieses Halts verändert und der Menge a^r die Anfrage r hinzugefügt werden. Im Fall, dass der Bus bisher nicht an r^o hält, müssen für den Halt zwei neue Aktionen a_1 und a_2 eingefügt werden. Dabei muss zuerst die letzte Abfahrtsaktion a_* vor r^o gefunden werden. Nach dieser wird die Ankunftsaktion a_1 an r^o eingefügt. Der Ankunftszeitpunkt wird dabei auf $a_1^t \coloneqq a_*^t + T_{\{a_*^h, r^o\}}$ gesetzt. Die zweite Aktion a_2 ist eine Abfahrtsaktion von r^o. Es gilt $a_2^r \coloneqq \{r\}$ und für den Abfahrtszeitpunkt wird, wie bereits bei den vorherigen Fällen, der frühestmögliche Zeitpunkt berechnet. Für die Zielhaltestelle r^d wird analog zur Zielhaltestelle im zweiten Fall verfahren. Für jeden zusätzlichen Halt muss der erste Nachfolger und alle weiteren Aktionen bei Bedarf mithilfe des Verfahrens aus dem ersten Fall verzögert werden.

Fall 5: Im fünften Fall befindet sich die Starthaltestelle r^o innerhalb und die Zielhaltestelle r^d nach der Sublinie *sub*. In diesem Fall muss wie im vierten Fall überprüft werden, ob der Bus bereits an r^o hält. Die eventuell notwendigen Aktionen werden, analog zum vierten Fall, eingefügt. Sollte b bereits an r^o halten und die letzte Haltestelle von *sub* sein, muss an dieser Stelle eine Abfahrtsaktion a_0 eingefügt werden, da in diesem Fall noch keine Abfahrt von r^o existiert. Dabei wird für den Abfahrtszeitpunkt die Zeit der letzten Aktion der Sublinie *sub* plus die minimale Haltezeit t_s gewählt. In diesem Fall

wird danach eine Ankunftsaktion a_1 an r^d geplant. Dabei gilt $a_1^t \coloneqq T_{\{r^o,r^d\}}$. Im Fall, dass r^o nicht die letzte Haltestelle in *sub* ist, müssen hingegen zwei neue Aktionen a_2 und a_3 geplant werden. Bei a_2 handelt es sich um eine Abfahrtsaktion von der letzten Haltestelle von *sub*. Der Abfahrtszeitpunkt ist dabei die Ankunftszeit der letzten Aktion in *sub* plus die minimale Standzeit t_s. Diese Anfrage transportiert keine Anfragen, es gilt also $a^r \coloneqq \emptyset$. Die Aktion a_3 ist eine Ankunftsaktion an r^d mit $a_3^t \coloneqq t_2^t + T_{\{a_2^h,r^d\}}$. Sollten bisher Aktionen nach der Sublinie *sub* folgen, muss, genau wie im dritten Fall, weitere Aktionen eingefügt werden, die den Halt an r^d mit der nächsten Sublinie verbinden.

Fall 6: Im sechsten Fall befindet sich die Starthaltestelle r^o und die Zielhaltestelle r^d nach der Sublinie *sub*. Für den Halt an r^o und an r^d werden insgesamt vier neue Aktionen $a_1, \ldots, a_4$ benötigt, welche nach den Aktionen von *sub* in den Aktionsplan eingefügt werden. Die erste Aktion a_1 ist eine Abfahrtsaktion von der letzten Haltestelle der Sublinie *sub* mit $a_1^r \coloneqq \emptyset$. Der Abfahrtszeitpunkt wird auf die Zeit der letzten Aktion innerhalb von *sub* plus die minimale Standzeit t_s gesetzt. Die zweite Aktion a_2 ist eine Ankunftsaktion an r^o zum Zeitpunkt $a_2^t \coloneqq a_1^t + T_{\{a_1^h,r^o\}}$. Bei der dritten Aktion a_3 handelt es sich um eine Abfahrtsaktion von r^o mit $a_3^r \coloneqq \{r\}$. Für die Abfahrtszeit wird, wie bereits in den vorherigen Fällen, der frühestmögliche Abfahrtszeit gewählt. Die vierte Aktion a_4 ist eine Ankunftsaktion an r^d mit $a_4^t \coloneqq a_3^t + T_{\{r^o,r^d\}}$. Sollte es vorher Aktionen nach *sub* gegeben haben, muss nun wieder, wie im dritten Fall, der Halt an r^d mit der nächsten Sublinie verbunden werden.

Neben diesen Optionen eine Anfrage r in eine bestehende Sublinie *sub* einzufügen, kann auch eine ganz neue Fahrt, welche die Anfrage r bearbeitet, geplant werden. Hierfür wird die von Puppe [Pup23b] entwickelte *IntegriereAnfrageBeimWarten*-Funktion verwendet. Diese läuft durch den bestehenden Aktionsplan und sucht ab dem aktuellen Simulationszeitpunkt den frühestmöglichen Zeitpunkt, zu dem der Bus länger als die minimale Standzeit t_s wartet und dabei aktuell keine Kunden befördert. An dieser Stelle im Aktionsplan wird eine neue Fahrt, welche über r_o nach r_d und gegebenenfalls wieder zurück zur Ausgangshaltestelle fährt, eingefügt.

Die Insert-Operation versucht in jede Sublinie, die noch unausgeführte Aktionen besitzt, die Anfrage r einzufügen. Zusätzlich wird mithilfe der Funktion *IntegriereAnfrageBeimWarten* eine weitere Variante berechnet. Alle diese möglichen Varianten werden auf Gültigkeit überprüft. Aus der Menge aller gültigen Varianten wird diejenige ausgewählt, die bezüglich der Bewertungsfunktion den niedrigsten Wert besitzt. Hierbei wird immer die Bewertungsfunktion der Transportstrategie genutzt, die die Insert-Operation ausführt. Sollte es keine gültige Variante geben, schlägt diese Insert-Operation fehl.

Delete-Operation

Die *Delete-Operation* löscht eine Anfrage r aus dem Aktionsplan A eines Busses $b \in B$. Der Aktionsplan wird dabei so angepasst, dass alle Aktionen, die rein für die Bearbeitung von r benötigt werden, entfernt werden. Die Anfrage r kann nur aus A gelöscht werden, wenn der Bus b die Bearbeitung der Anfrage r plant und die Bearbeitung bisher nicht begonnen hat. Sollte r diese Bedingungen verletzen, so schlägt die Delete-Operation fehl. Im ersten Schritt wird die Abfahrtsaktion in A gesucht, welche r bearbeitet. Diese

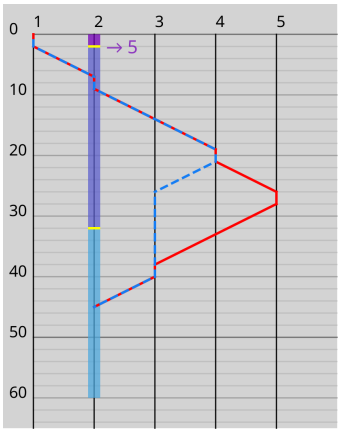
Aktion wird durch eine Kopie ersetzt, in der aus der Menge der bearbeiteten Anfragen, r gelöscht wurde. Im nächsten Schritt muss ermittelt werden, ob das Halten an der Start- und Zielhaltestelle weiterhin benötigt wird. Das Halten an einer Haltestelle besteht in der Regel aus einer Ankunfts- und einer Abfahrtsaktion. Ist der Halt der letzte Halt des gesamten Aktionsplans, kann er auch nur aus einer Ankunftsaktion bestehen.

Ein Halt wird nicht mehr benötigt, wenn er folgende Bedingungen erfüllt:

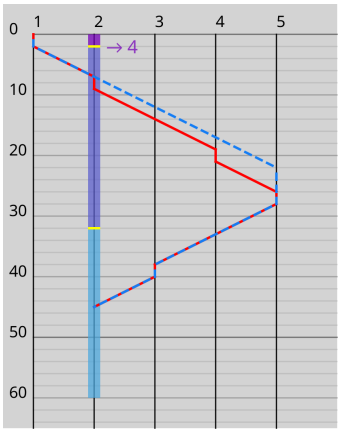
1. An diesem Halt werden keine Anfragen dem Bus hinzugefügt.
2. Die Haltestelle dieses Halts ist keine Zielhaltestelle einer Anfrage, die aktuell von diesem Bus bearbeitet wird.
3. Keine Aktion des Halts wurde bereits ausgeführt.

Für die Start- und Zielhaltestelle von r ergeben sich vier Fälle, je nachdem, welche der beiden Haltestellen noch benötigt wird:

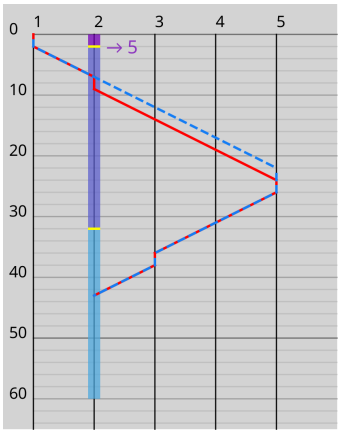
1. Der Halt an der Starthaltestelle wird weiterhin benötigt, der Halt an der Zielhaltestelle nicht.
2. Der Halt an der Start- und an der Zielhaltestelle wird nicht mehr benötigt.
3. Der Halt an der Starthaltestelle wird nicht mehr benötigt, aber der Halt an der Zielhaltestelle wird weiterhin benötigt.
4. Der Halt an der Start- und Zielhaltestelle wird weiterhin benötigt.



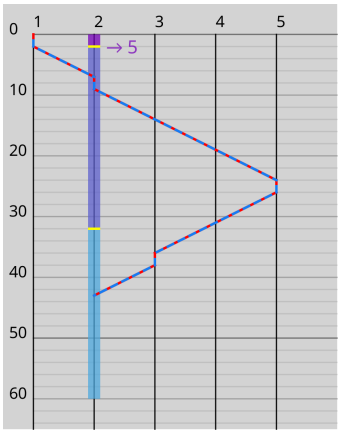
(a) Der Halt an der Haltestelle wird noch benötigt, der Halt an der Zielhaltestelle nicht.



(b) Der Halt an der Start- und an der Zielhaltestelle wird nicht mehr benötigt.



(c) Der Halt an der Starthaltestelle wird nicht mehr benötigt, aber der Halt an der Zielhaltestelle schon.



(d) Der Halt an der Start- und Zielhaltestelle wird weiterhin benötigt.

Abb. 5.6: Beispiele für die verschiedenen Fälle der Delete-Operation

Nachfolgend wird für jeden dieser vier Fälle beschrieben, wie der Aktionsplan A abgeändert werden muss. Abbildung 5.6 zeigt für jeden Fall ein Beispiel, wie der Aktionsplan verändert wird. Die rote Linie zeigt dabei den bisherigen Aktionsplan und die blaugestrichelte Linien den Aktionsplan, nach der Delete-Operation.

Die Menge A^{between} beschreibt alle Aktionen, die sich zwischen dem Halt an r^o und dem Halt r^d befinden. Weiter beschreiben die Mengen A_{before} und A_{after} die Aktionen in A vor dem Halt an r^o oder nach dem Halt an r^d.

Fall 1: Im ersten Fall muss der Halt an der Zielhaltestelle r^d entfernt werden. Hierfür

werden die Aktionen dieses Halts aus A entfernt. Danach werden drei Teilfälle (a), (b) und (c) unterschieden:

(a) In diesem Fall gilt $A_{\text{between}} \neq \emptyset$ und $A_{\text{after}} \neq \emptyset$. Es wird überprüft, ob die letzte Aktion $a_{\text{between}} \in A_{\text{between}}$ an der gleichen Haltestelle hält wie die erste Aktion $a_{\text{after}} \in A_{\text{after}}$. Ist dies der Fall, so müssen die Aktionen a_{between} und a_{after} ebenfalls aus A entfernt werden, da diese keinen Zweck mehr erfüllen. Ist dies hingegen nicht der Fall, so muss die Ankunftszeit von a_{after} angepasst werden. Diese wird auf $a^t_{\text{after}} \coloneqq a^t_{\text{between}} + T_{\{a^h_{\text{between}}, a^h_{\text{after}}\}}$ gesetzt.

(b) In diesem Fall gilt $A_{\text{between}} \neq \emptyset$ und $A_{\text{after}} = \emptyset$. Die letzte Aktion $a_{\text{between}} \in A_{\text{between}}$ wird gelöscht, da diese eine Abfahrtsaktion ist und im aktualisierten Aktionsplan nach a_{between} keine weiteren Aktionen folgen.

(c) In diesem Fall gilt $A_{\text{between}} = \emptyset$ und $A_{\text{after}} = \emptyset$. Es wird die Abfahrtsaktion des Halts an r^o gelöscht, da im aktualisierten Aktionsplan keine weiteren Aktionen folgen.

Fall 2: Im zweiten Fall muss sowohl der Halt an der Starthaltestelle r^o als auch an der Zielhaltestelle r^d entfernt werden. Dafür werden die Aktionen der beiden Halts aus A entfernt. Danach werden vier Unterfälle (a)-(d) unterschieden.

(a) In diesem Fall gilt $A_{\text{between}} = \emptyset$ und $A_{\text{after}} = \emptyset$. Gilt außerdem $|A_{\text{before}}| > 1$, so muss die letzte Aktion aus A_{before} gelöscht werden.

(b) In diesem Fall gilt $A_{\text{between}} \neq \emptyset$ und $A_{\text{after}} = \emptyset$. Hier muss zusätzlich die letzte Aktion aus A_{between} gelöscht werden. Weiter muss die Ankunftszeit der ersten Aktion $a_{\text{between}} \in A_{\text{between}}$ aktualisiert werden. Sie wird auf $a_{\text{between}} \coloneqq a^t_{\text{before}} + T_{\{a^h_{\text{before}}, a^h_{\text{between}}\}}$. Falls $|A_{\text{before}}| > 1$, dann ist a_{before} die letzte Aktion aus A_{before} sonst die von b zuletzt ausgeführte Aktion.

(c) In diesem Fall gilt $A_{\text{after}} \neq \emptyset$, $A_{\text{before}} \neq \emptyset$ und die letzte Aktion $a_{\text{before}} \in A_{\text{before}}$ hält an der gleichen Haltestelle wie die erste Aktion $a_{\text{after}} \in A_{\text{after}}$. Hier müssen die Aktionen a_{before} und a_{after} gelöscht werden.

(d) Dieser Fall wird gewählt, wenn keiner der anderen Fälle zutrifft. Der Aktionsplan wird in diesem Fall iterativ verändert. Falls $A_{\text{before}} \neq \emptyset$ und $A_{\text{between}} \neq \emptyset$, muss überprüft werden, ob die letzte Aktion $a_{\text{before}} \in A_{\text{before}}$ die gleiche Haltestelle besitzt wie die erste Aktion $a_{\text{between}} \in A_{\text{between}}$. Ist dies der Fall, müssen die Aktionen a_{before} und a_{between} gelöscht werden und im Folgenden gilt $A_{\text{before}} \coloneqq A_{\text{before}} \backslash \{a_{\text{before}}\}$ und $A_{\text{between}} \coloneqq A_{\text{between}} \backslash \{a_{\text{between}}\}$. Falls $A_{\text{between}} \neq \emptyset$ und $A_{\text{after}} \neq \emptyset$ gilt, muss überprüft werden, ob die aktuell letzte Aktion aus $a_{\text{between}} \in A_{\textit{between}}$ die gleiche Haltestelle besitzt wie die aktuell erste Aktion $a_{\text{after}} \in A_{\text{after}}$. Trifft dies zu, so müssen a_{between} und a_{after} gelöscht werden und im Folgenden gilt, dass $A_{\text{between}} = A_{\text{between}} \backslash \{a_{\text{between}}\}$ und $A_{\text{after}} = A_{\text{after}} \backslash \{a_{\text{after}}\}$. Danach wird überprüft, ob $A_{\text{after}} \neq \emptyset$ und $a^h_{\text{before}} \neq a^h_{\text{after}}$ gilt. Dabei ist a_{after} die aktuell letzte

Aktion in A_{after} und a_{before} entweder die letzte Aktion in A_{before} ist oder wenn $A_{\text{before}} = \emptyset$, gleich die von b zuletzt ausgeführte Aktion ist. Trifft dies zu, so wird $a^t_{\text{after}} = a^t_{\text{before}} + T_{\{a^h_{\text{before}}, a^h_{\text{after}}\}}$ gesetzt. Zum Schluss muss überprüft werden, ob aktuell gilt, dass $A_{\text{after}} \neq \emptyset$ und $a^h_{\text{after}} = a^h_{\text{before}}$. Dabei ist a_{after} die aktuell letzte Aktion aus A_{after} und a_{before} entweder die letzte Aktion aus $A_{\text{before}} \cup A_{\text{between}}$ oder falls $A_{\text{before}} \cup A_{\text{between}} = \emptyset$ die Aktion, die b zuletzt ausgeführt hat. Trifft dies zu, so wird $a^t_{\text{after}} \coloneqq a^t_{\text{before}} + T_{\{a^h_{\text{before}}, a^h_{\text{after}}\}}$ gesetzt.

Fall 3: Im dritten Fall muss der Halt an der Starthaltestelle r^o entfernt werden. Dafür werden die jeweiligen Aktionen aus A gelöscht. Als Nächstes muss überprüft werden, ob $a^h_{\text{before}} = a^h_{\text{between}}$. Dabei ist a_{between} die erste Aktion aus A_{between} und a_{before} ist entweder die letzte Aktion aus A_{before} falls $|A_{\text{before}}| > 1$ oder die von b zuletzt ausgeführte Aktion. Trifft dies zu, so müssen zusätzlich a_{before} und a_{between} gelöscht werden, da sie eine verbotene Fahrt zwischen der gleichen Haltestelle bilden. Sollten die Haltestellen hingegen nicht übereinstimmen, so muss die Ankunftszeit von a_{between} auf $a^t_{\text{before}} + T_{\{a^h_{\text{before}}, a^h_{\text{between}}\}}$ gesetzt werden.

Fall 4: Im vierten Fall müssen keine weiteren Aktionen gelöscht oder verändert werden.

Move-Operation

Die *Move-Operation* verschiebt eine Anfrage r aus dem Aktionsplan eines Busses b_a in den Aktionsplan eines anderen Busses b_b. Diese Operation kombiniert dabei die Delete- und Insert-Operation. Im ersten Schritt löscht die Delete-Operation r aus dem Aktionsplan von b_a. Ein Fehlschlagen der Delete-Operation führt dazu, dass auch die Move-Operation fehlschlägt. Sollte die Delete-Operation jedoch eine gültige Lösung liefern, so wird in diese Lösung mithilfe der Insert-Operation r in den Aktionsplan von b_b eingefügt. Auch diese Operation kann fehlschlagen. Sollte dies der Fall sein, wird auch die Move-Operation fehlschlagen. Alternativ wird die nun erhaltene Lösung zurückgegeben. Trifft ein solcher Fall ein, so ist auch das Ergebnis der Move-Operation eine ungültige Lösung.

5.2.2 Bewertungsfunktion

Um Lösungen miteinander vergleichen zu können, wird eine Bewertungsfunktion benötigt. Gleichung 5.4 zeigt die $Bewertungsfunktion$ für eine Lösung s. Ein geringer Wert gilt hierbei als besser.

$$\begin{aligned} Bewertungsfunktion(s) = &\alpha \cdot \text{DurchschnWartezeit} + \\ &\beta \cdot \text{DurchschnTransportzeit} + \\ &\gamma \cdot \text{GesamteFahrzeit} \end{aligned} \tag{5.4}$$

Die $Bewertungsfunktion$ setzt sich aus folgenden Evaluationskriterien zusammen: DurchschnWartezeit, DurchschnTransportzeit und AnzahlGenutzerBusse. Die Definitionen für diese Evaluationskriterien finden sich in Abschnitt 5.3. Mithilfe der Parameter α, β und γ lassen sich die verschiedenen Evaluationskriterien gewichten. Bei der Auswahl der Evaluationskriterien wurde der Fokus auf Evaluationskriterien gelegt, für die erwartet wird, dass sie den größten Einfluss auf die endgültigen Lösungen haben.

5.2.3 Greedy Transportstrategie

Die Greedy Transportstrategie ist die erste der zwei Transportstrategien, die für diese Arbeit entwickelt wurden. Sie nutzt die Insert-Operation, um neue Anfragen in die Aktionspläne der Busse einzufügen. Zuerst werden alle Anfragen, die bisher nicht geplant oder abgelehnt wurden, gesucht und in der Menge Z gespeichert.

Sollte $Z = \emptyset$ sein, so werden keine Änderungen an den Aktionsplänen der Busse vorgenommen. Sollte hingegen $Z \neq \emptyset$ gelten, wird versucht, jede Anfrage $r \in Z$ iterativ in die bestehende Lösung einzufügen. Das Einfügen einer Anfrage $r \in Z$ läuft wie folgt ab:

Die Anfrage r wird zuerst mithilfe der Insert-Operation in jeden Bus $b \in B$ eingefügt. Schlägt das Einfügen für einen Bus fehl, so wird dieser Bus für r nicht mehr betrachtet. Ist das Einfügen hingegen erfolgreich, so wird die entstandene Lösung einer Menge X hinzugefügt. Sollte jede Insert-Operation fehlschlagen ($X = \emptyset$), konnte die Anfrage r in keinen Bus integriert werden. In diesem Fall wird r abgelehnt. Für die nächste Iteration wird die bestehende Lösung weitergenutzt. Konnte r hingegen in mindestens einen Bus eingefügt werden ($X \neq \emptyset$), so wird die Lösung ausgewählt, für welche, die $Bewertungsfunktion$ den niedrigsten Wert liefert. Für die $Bewertungsfunktion$ werden immer die Parameter $\alpha = 100$, $\beta = 0$ und $\gamma = 0$ gewählt.

Diese Lösung dient als Ausgangspunkt für das Einfügen der nächsten Anfrage. Außerdem wird die Anfrage r als geplant markiert.

Die durch diesen Prozess zuletzt erhaltene Lösung wird zurückgegeben.

5.2.4 Tabu-Search Transportstrategie

Die Tabu-Search Transportstrategie dieser Arbeit erweitert die Greedy Transportstrategie mit einer Tabu-Search. Der Ablauf dieser Strategie ist in Abbildung 5.7 dargestellt. Im ersten Schritt wird die Greedy Strategie genutzt, um neue Anfragen entweder in

die bestehende Lösung zu integrieren oder abzulehnen. Um Lösungen der Greedy Strategie weiter zu verbessern, wird alle fünf Minuten (in der Simulation 50 Ticks) eine Tabu-Search ausgeführt. In den Fällen, in denen noch keine fünf Minuten seit der letzten Tabu-Search vergangen sind, wird an dieser Stelle die Lösung der Greedy Strategie zurückgegeben. Sollte die Greedy Strategie keine Änderung an den Aktionsplänen vorgenommen haben, so wird in diesem Fall auch von der Tabu-Search Transportstrategie keine Änderung vorgenommen.

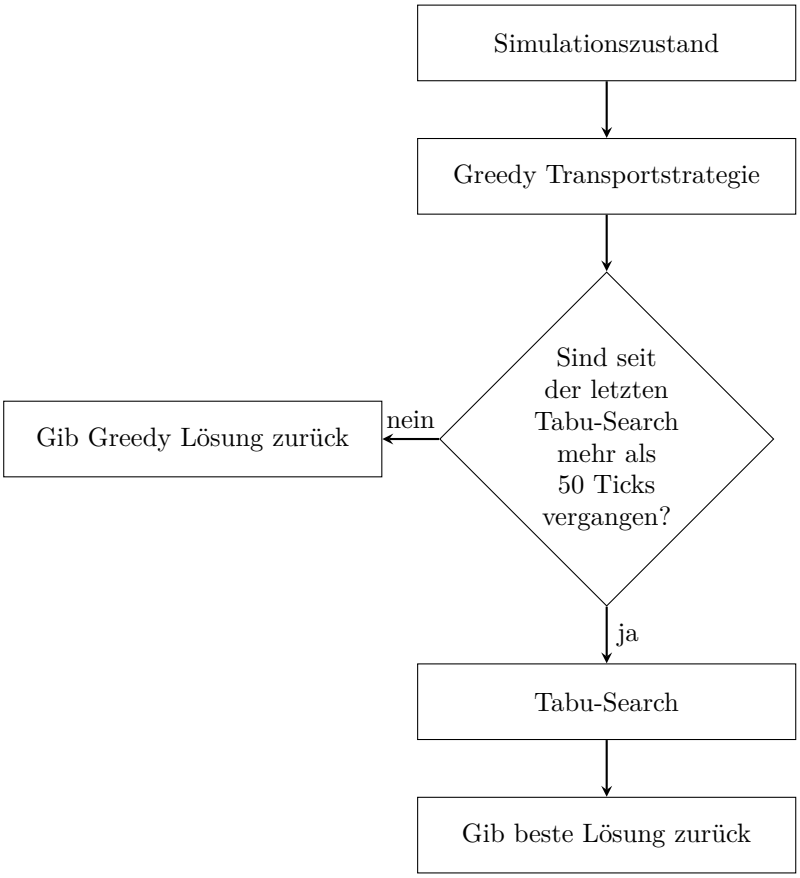


Abb. 5.7: Ablauf der Tabu-Search Transportstrategie

Vor dem Ausführen der Tabu-Search wird ermittelt, welche Anfragen aktuell verändert werden können. Diese werden in der Menge R_{v} gespeichert. Eine Anfrage kann aktuell verändert werden, wenn sie bisher nicht abgelehnt wurde oder bisher nicht bearbeitet wird. Um die Rechenzeit zu optimieren, kann zusätzlich ein Simulationsparameter Zeithorizont angeben werden. Durch diesen Parameter werden in der Menge R_{v} nur Anfragen betrachtet, deren gewünschte Abfahrtszeit kleiner der aktuellen Zeit plus dem Zeithorizont sind.

Algorithmus 1: TabuSearch(s, R_v)

Eingabe: initiale Lösung s, Menge veränderbarer Anfragen R_v
Ausgabe: beste gefundene Lösung

```
1  besteLösung = s
2  besterWert = Bewertungsfunktion(s)
3  aktuelleLösung = s
4  letzteÄnderung = 0
5  i = 0
6  while i < TabuIterationen und (i − letzteÄnderung) < MaximalKeineÄnderung
   do
7      AktualisiereTabuZeiten()
8      N(aktuelleLösung) = BerechneNachbarschaft(aktuelleLösung)
9      if N(aktuelleLösung) = ∅ then
10         break
11     besterNachbar = FindeBestenNachbarn(N(aktuelleLösung))
12     besterNachbarWert = Bewertungsfunktion(besterNachbar)
13     SetzeDiesenZugAufDieTabuListe(aktuelleLösung, besterNachbar)
14     aktuelleLösung = besterNachbar
15     if besterNachbarWert < besterWert then
16         besteLösung = besterNachbar
17         besterWert = besterNachbarWert
18         letzteÄnderung = i
19     i = i + 1
20 return besteLösung
```

Algorithmus 1 zeigt die eigentliche Tabu-Search. Als Eingabe erhält sie die aktuelle Lösung s und die Menge der aktuell veränderbaren Anfragen R_v. Die Parameter TabuIterationen, MaximalKeineÄnderung, TabuZeit und Zeithorizont können vor Beginn der Simulation festgelegt werden. Der Parameter TabuIterationen gibt an, wie viele Iterationen die Tabu-Search maximal nach einer besseren Lösung suchen soll. Der Parameter MaximalKeineÄnderung gibt an, wie viele Iterationen weitergesucht werden soll, wenn sich der Wert der besten Lösung nicht weiter verändert. Dadurch kann Rechenzeit eingespart werden. Der Parameter TabuZeit gibt an, wie viele Iterationen ein genutzter Zug auf der Tabu-Liste bleibt. Die initiale Lösung s wird zu Beginn der Tabu-Search als beste und aktuelle Lösung gesetzt (Zeile 1-3).

Die Zeilen 4 und 5 initialisieren die Variablen letzteÄnderung und den Zähler i, der für das Mitzählen der Iterationen genutzt wird. Die Variable letzteÄnderung wird dafür genutzt, um sich zu merken, in welcher Iteration zuletzt eine neue beste Lösung gefunden wurde.

Jede Iteration der Tabu-Search beginnt damit, dass alle Tabu-Zeiten aktualisiert werden (Zeile 7). Für jeden Zug der Tabu-Liste, der tabu ist, wird initial ein Zähler auf

die TabuZeit gesetzt. Ein Zug verbietet es, eine Anfrage in einen bestimmten Bus zu verschieben.

Die Methode *AktualisiereTabuZeiten()* reduziert diese Zähler in jeder Iteration um 1. Erreicht ein Zähler 0, so wird der dazugehörige Zug nicht weiter tabuisiert. Im nächsten Schritt wird die Nachbarschaft der aktuell besten Lösung berechnet (Zeile 8).

Die *Nachbarschaft* $N(s)$ einer Lösung s ist definiert als die Menge aller gültigen Lösungen, bei denen genau eine Anfrage aus R_v einem anderen Bus zugeordnet worden ist. Für die Berechnung von $N(s)$ wird die Move-Operation eingesetzt. Dabei wird Busweise vorgegangen und jeder Bus aus B der Reihe nach betrachtet. Für einen Bus $b \in B$ werden alle Anfragen gesucht, die in der aktuellen Lösung von b bearbeitet werden. Aus diesen Anfragen werden nur diejenigen betrachtet, die sich auch in R_v befinden. Diese Auswahl wird in der Menge R_b gespeichert. Im nächsten Schritt wird jede Anfrage $r \in R_b$ mithilfe der Move-Operation in jeden anderen Bus aus B von b verschoben. Sollte es aktuell tabu sein, die Anfrage r in einen bestimmten Bus zu verschieben, so wird diese spezielle Move-Operation übersprungen. Wenn es außerdem dazu kommt, dass die Move-Operation fehlschlägt, dann wird dieser Zug ebenfalls nicht weiter betrachtet. Die Lösung jeder sonst erfolgreichen Move-Operation wird der Menge $N(s)$ hinzugefügt.

Sollte $N(\text{aktuelleLösung}) = \emptyset$ gelten, so wird die Tabu-Search an dieser Stelle beendet (Zeile 9-10).

Sollte $N(\text{aktuelleLösung}) \neq \emptyset$ gelten, so wird im nächsten Schritt der Tabu-Search die beste Lösung der Nachbarschaft gesucht (Zeile 11-12). Hierfür iteriert die Methode *FindeBestenNachbarn* über alle Lösungen in $N(s)$ und gibt diejenige Lösung zurück, die den kleinsten Wert der Bewertungsfunktion liefert.

Die Lösungen besterNachbar und aktuelleLösung unterscheiden sich durch genau eine Anfrage, die in einen anderen Bus verschoben wurde. Für diese Anfrage setzt die Methode SetzeDiesenZugAufDieTabuListe(...) einen neuen Tabu-Zug, der verbietet, diese Anfrage in den ursprünglichen Bus zu verschieben (Zeile 13).

Danach wird die Lösung besterNachbar die neue aktuelleLösung. Für den besten Nachbarn wird zusätzlich geschaut, ob er besser als die aktuell beste Lösung ist (Zeile 15).

Sollte er besser sein, so wird er die neue besteLösung (Zeile 16-17)

Danach wird die Variable letzteÄnderung auf die aktuelle Iteration gesetzt (Zeile 18).

Am Ende jeder Iteration wird der Zähler i um 1 erhöht. Die Tabu-Search endet, wenn in einem Durchlauf keine Nachbarn gefunden wurden, oder wenn TabuIterationen viele Iterationen durchgeführt wurden, oder MaximalKeineÄnderung viele Iterationen keine neue beste Lösung gefunden wurde. Beim Beenden der Tabu-Search wird die Lösung besteLösung zurückgegeben (Zeile 20).

Die Lösung, die durch die Tabu-Search gefunden wurde, wird von der Transportstrategie als neue Lösung zurückgegeben.

5.3 Evaluationskriterien

Am Ende der Simulation werden Werte für den Vergleich und die Bewertung der verwendeten Transportstrategien berechnet. Diese Werte werden im Rahmen dieser Arbeit als Evaluationskriterien bezeichnet. Dabei orientiert sich die Auswahl der Evaluationskriterien in Teilen an der Arbeit von Puppe [Pup23b]. Die Notation hingegen orientiert sich an [LLMS21]

Für die Berechnung der Evaluationskriterien werden die Anfragen in zwei disjunkte Mengen aufgeteilt. Die Menge $R_\mathrm{a} \subseteq R$ enthält alle akzeptierten Anfragen und die Menge $R_\mathrm{c} \subseteq R$ alle abgelehnten Anfragen. Die DurchschnWartezeit, die DurchschnTransportzeit und die DurchschnFahrzeit werden nur mit Anfragen aus R_a berechnet. Dies verhindert, dass abgelehnte Anfragen den Wert dieser Evaluationskriterien beeinflussen. Unterschiedliche Transportstrategien lassen sich dadurch besser vergleichen. Für die Berechnung dieser drei Kriterien werden die Abfahrts- und Ankunftsaktionen, die eine Anfrage aus R_a bearbeiten, benötigt. Für eine Anfrage $r \in R_a$ sei a_{r^o}, die Abfahrtsaktion, die r von der Starthaltestelle abholt und a_{r^d} die Ankunftsaktion, die r an der Zielhaltestelle absetzt.

Durchschnittliche Wartezeit

Die Wartezeit(r) einer Anfrage $r \in R_a$ ist die Zeit, die der Kunde von r an der Starthaltestelle auf die Abholung durch einen Bus wartet. Sie berechnet sich aus der Differenz der Abholzeit $a^t_{r^o}$ und der gewünschten Abfahrtszeit r^g.

Die DurchschnWartezeit ist definiert als:

$$\text{DurchschnWartezeit} \coloneqq \frac{\sum_{r \in R_a} \text{Wartezeit(r)}}{|R_a|} \tag{5.5}$$

Die DurchschnWartezeit wird in Simulationsticks gemessen. Eine kurze DurchschnWartezeit wird bevorzugt.

Durchschnittliche Transportzeit

Die Transportzeit(r) einer Anfrage $r \in R_a$ ist die Zeit, die der Kunde von r im Bus verbringt. Sie berechnet sich aus der Differenz zwischen dem Ankunftszeitpunkt $a^t_{r^d}$ und dem Abfahrtszeitpunkt $a^t_{r^o}$.

Die DurchschnTransportzeit ist definiert als:

$$\text{DurchschnTransportzeit} \coloneqq \frac{\sum_{r \in R_a} \text{Transportzeit(r)}}{|R_a|} \tag{5.6}$$

Die DurchschnTransportzeit wird in Simulationsticks gemessen und es werden kleine Werte bevorzugt.

Durchschnittliche Fahrzeit

Die Fahrzeit(r) einer Anfrage $r \in R_a$ ist die Zeit, die der Kunde von r im Bus sitzt und dabei fährt. Sie entspricht der Transportzeit(r), bei der das Warten an Haltestellen herausgerechnet wurden.

Die DurchschnFahrzeit ist wie folgt definiert:

$$\text{DurchschnFahrzeit} \coloneqq \frac{\sum_{r \in R_a} \text{Fahrzeit(r)}}{|R_a|} \tag{5.7}$$

Die DurchschnFahrzeit sollte möglichst kurz sein und wird in Simulationsticks gemessen.

Anzahl genutzter Busse

Sei B_{genutzt} die Menge aller genutzten Busse aus B. Ein Bus gilt als genutzt, wenn er am Ende der Simulation mehr als eine Aktion seines Aktionsplans ausgeführt hat. Da jeder Bus zu Beginn eine Ankunftsaktion an seiner Starthaltestelle ausführt, gilt ein Bus mit nur einer ausgeführten Aktion nicht als genutzt.

Die AnzahlGenutzerBusse ist definiert als:

$$\text{AnzahlGenutzerBusse} \coloneqq |B_{\text{genutzt}}| \tag{5.8}$$

Die AnzahlGenutzerBusse sollte möglichst gering sein.

Gesamte Fahrzeit

Die Fahrzeit(b) eines Busses $b \in B$ ist die Zeit, die b fährt. Sie wird ermittelt, indem durch den Aktionsplan von b gelaufen und die Fahrzeit zwischen dem Halt an zwei Haltestellen aufsummiert wird. Die Fahrzeit wird berechnet, indem von der Ankunftszeit an der zweiten Haltestelle die Abfahrtszeit der ersten Haltestelle abgezogen wird. Die Fahrzeit(b) enthält also keine Wartezeiten an Haltestellen.

Die GesamteFahrzeit ist wie folgt definiert:

$$\text{GesamteFahrzeit} \coloneqq \sum_{b \in B} \text{Fahrzeit(b)} \tag{5.9}$$

Die GesamteFahrzeit ist proportional zur gefahrenen Strecke der Busse. Sie sollte möglichst klein sein, da so Kosten für einen Betreiber eines solchen Bussystems eingespart werden können.

Akzeptanzquote

Die Akzeptanzquote ist definiert als:

$$\text{Akzeptanzquote} \coloneqq \frac{|R_a|}{|R|} \tag{5.10}$$

Stellt ein Kunde eine Anfrage, erwartet er, dass diese auch bearbeitet wird. Daher sollte die Akzeptanzquote möglichst hoch sein.

Durchschnittliche Busauslastung

Die Auslastung(b) eines Busses $b \in B$ ist folgendermaßen definiert:

$$\text{Auslastung(b)} \coloneqq \frac{\sum_{r \in b^r} \text{Fahrzeit(r)}}{C \cdot \text{Fahrzeit(b)}} \tag{5.11}$$

Die DurchschnBusauslastung sollte möglichst hoch sein und ist definiert als:

$$\text{DurchschnBusauslastung} \coloneqq \frac{\sum_{b \in B_{\text{genutzt}}} \text{Auslastung(b)}}{|B_{\text{genutzt}}|} \tag{5.12}$$

Durchschnittliche Berechnungszeit

Die Berechnungszeit gibt an, wie lange die Transportstrategie der Phase-2 im Durchschnitt für die Planung der neuen Aktionspläne benötigt hat. Dafür wird für jeden Planungsdurchgang die Zeit gestoppt und in eine Menge T gespeichert. Hierbei ist anzumerken, dass nur Planungsdurchgänge betrachtet werden, die auch tatsächlich eine Veränderung der Aktionspläne bewirken. Die Zeit wird dabei in Millisekunden gemessen. Die Berechnungszeit ist definiert als der Durchschnitt aller Werte in T:

$$\text{Berechnungszeit} \coloneqq \frac{\sum_{t \in T} t}{|T|} \tag{5.13}$$

Die Berechnungszeit muss möglichst gering sein, damit Anfragen auch im dynamischen Kontext bearbeitet werden können.

6 Ergebnisse

Die Greedy und Tabu-Search Strategie, die für diese Arbeit entwickelt wurden, sollen im Folgenden evaluiert werden. Dazu werden die Greedy und Tabu-Search Transportstrategie mit der Greedy Strategie von Puppe [Pup23b] verglichen. Im Folgenden wird die Greedy Strategie dieser Arbeit als *Greedy*, die Tabu-Search Strategie als *Tabu-Search* und die Greedy Strategie von Puppe als *Puppe* abgekürzt.

Für den Vergleich der Transportstrategien wird die weiterentwickelte Version des Simulationstools [Hei23] genutzt. Dabei wurde untersucht, wie sich die Evaluationskriterien aus Abschnitt 5.3 in Abhängigkeit der Anzahl von Anfragen für die drei Transportstrategien verändern. Die Anzahl der Anfragen wurde dabei zwischen 25 und 400 mit einer Schrittweite von 25 betrachtet. Die Simulationsdauer betrug drei Stunden (1800 Simulations-Ticks, wobei 1 Tick sechs Sekunden entspricht). Es wurde eine Flotte mit 5 Minibussen und einer Kapazität von $C = 6$ eingesetzt. Die Buslinie besteht aus 10 Haltestellen, wobei die Fahrzeit zwischen den Haltestellen auf 2 Minuten (20 Simulationsticks) festgelegt wurde. Die minimale Haltezeit t_s wurde auf 30 Sekunden (5 Simulationsticks) festgelegt.

Für Anfragen wurde eine maximale Wartezeit $W_{\max}$ von 15 Minuten (150 Simulationsticks) und ein MinAnfragefenster von 30 Minuten (300 Simulationsticks) gewählt.

Die folgenden Parameter wurden in der Bewertungsfunktion der Tabu-Search festgelegt: $\alpha = 100$, $\beta = 100$ und $\gamma = 1$. Für γ wurde deshalb 1 gewählt, da die Werte für die GesamteFahrzeit meist höher sind als die Werte für die DurchschnWartezeit und DurchschnTransportzeit. Würde man für γ einen ähnlichen Wert wie für α und β wählen, würde das dazu führen, dass zu stark auf die GesamteFahrzeit optimiert werden würde. Für den Anteil an Anfragen R_{ratio}, der in Phase-1 liegt, wurde 0.5 gewählt. Es werden also genau 50% aller Anfragen in Phase-1 und 50% aller Anfragen in Phase-2 gestellt.

Für Phase-1 kommt immer die Strategie Greedy zum Einsatz. Dies ermöglicht es, die Strategien Greedy, Tabu-Search und Puppe in Phase-2 unabhängig von der gewählten Strategie in Phase-1 zu evaluieren.

Für die Tabu-Search wurden die Parameter wie folgt gewählt: MaximalKeineÄnderung = 20, TabuIterationen = 100, TabuZeit = 10. Es wurde weiter kein Zeithorizont verwendet.

Um die Ergebnisse unabhängiger von einer konkreten „Anfragen-Instanz“ zu machen, wurde jeder Schritt der Experimente zehnmal durchgeführt. Diese 10 Instanzen basieren auf den Seeds 888 bis 897 (inklusive).

Das Testsystem war ein Apple Macbook Air 2020 mit einem M1 Prozessor und 8 GB Arbeitsspeicher.

Die Rohdaten aller Messungen können unter [Hei24] abgerufen werden.

Nachfolgend werden die Ergebnisse der einzelnen Evaluationskriterien beschrieben.

6.1 Akzeptanzquote

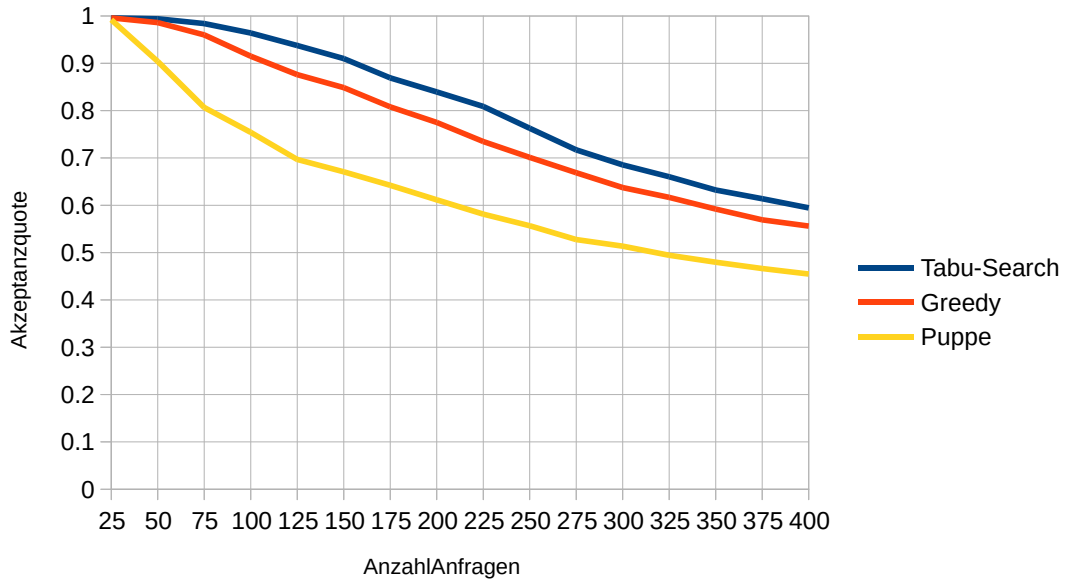


Abb. 6.1: Akzeptanzquote in Abhängigkeit der Anzahl der Anfragen

Abbildung 6.1 zeigt die Akzeptanzquote in Abhängigkeit der Anzahl von Anfragen. Für alle drei Strategien gilt, dass die Akzeptanzquote bei mehr Anfragen geringer ausfällt. Für jede Anzahl von Anfragen gilt, dass Tabu-Search die höchsten Werte für Akzeptanzquote, die Puppe die niedrigsten Werte und Greedy Werte zwischen Puppe und der Tabu-Search erreicht.

Für 25 Anfragen erreichen alle drei Strategien eine Akzeptanzquote von nahezu 1. Die Akzeptanzquote von Puppe fällt danach aber bereits für 75 Anfragen auf einen Wert von circa 0.8. Greedy erreicht diesen Wert erst bei 175 und die Tabu-Search erst bei 225 Anfragen.

Greedy und die Tabu-Search können für 50 Anfragen ebenfalls einen Wert nahe 1 erreichen. Ab 75 Anfragen fallen auch die Akzeptanzquote Werte von Greedy und Tabu-Search. Der Wert von Greedy für 75 Anfragen entspricht dabei dem Wert der Tabu-Search für 100 Anfragen. Danach entspricht der Wert von Greedy ungefähr dem Wert der Tabu-Search 50 Anfragen später. Dies gilt für jede Anzahl von Anfragen zwischen 100 und 400.

Etwas Ähnliches gilt für die Werte von Puppe und Greedy zwischen 75 und 400.

Dabei gilt jedoch, dass der Abstand, nach dem Greedy die Werte von Puppe erreicht, nicht konstant ist, sondern zwischen 100 und 125 variiert.

Für 400 Anfragen erreicht die Tabu-Search eine Akzeptanzquote von um die 0.6, die Greedy Strategie um die 0.55 und die Strategie von Puppe um die 0.45.

Für eine gegebene Anzahl an Anfragen kann der Wert von Akzeptanzquote nur höher

sein, wenn mehr Anfragen bearbeitet und somit nicht abgelehnt werden.

Daher kann festgehalten werden, dass die Tabu-Search für die gleiche Anzahl an Anfragen die meisten Anfragen, die Greedy die zweitmeisten und die Strategie von Puppe die wenigsten Anfragen bearbeitet. Weiter kann man festhalten, dass alle drei Strategien mit steigender Anzahl an Anfragen, einen geringeren Anteil aller Anfragen bearbeiten können.

6.2 Gesamte Fahrzeit

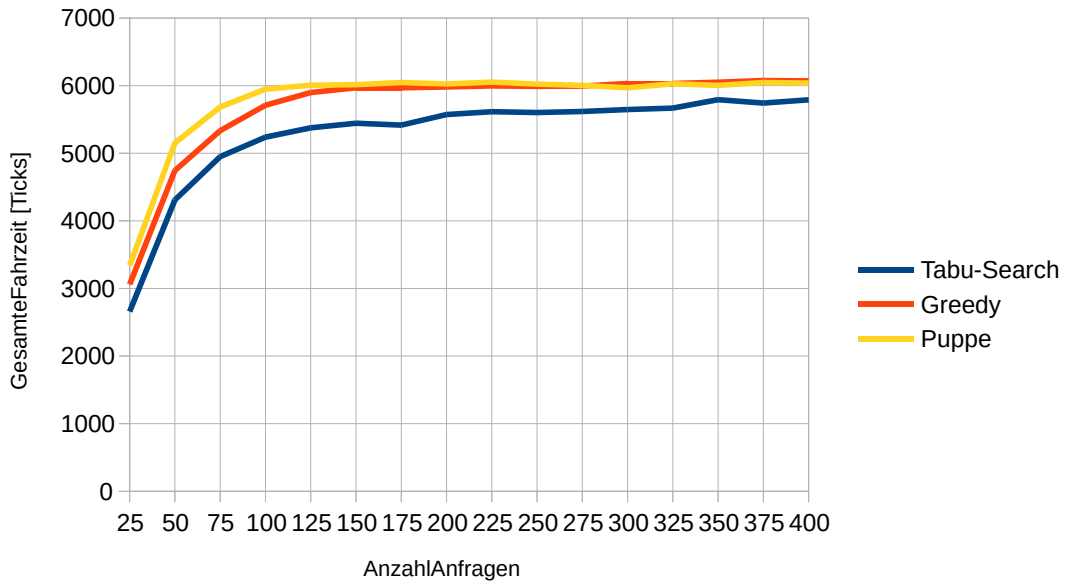


Abb. 6.2: Gesamte Fahrzeit in Abhängigkeit der Anzahl der Anfragen

Abbildung 6.2 zeigt die GesamteFahrzeit im Verhältnis zur Anzahl an Anfragen. Für alle drei Strategien gilt, dass die Werte für die GesamteFahrzeit im Bereich zwischen 25 und 125 Anfragen stark ansteigen und ab 125 nur noch leicht anwachsen, genauer gesagt (fast) konstant bleiben.

Dabei gilt für alle Strategien, dass die Steigung zwischen 25 und 50 höher ist als zwischen 50 und 75. Wobei die Steigungen zwischen 75 und 100 und danach zwischen 100 und 125 weiter abnimmt.

Über jede Anzahl an Anfragen hinweg, kann die Tabu-Search die niedrigsten Werte für die GesamteFahrzeit erzielen. Es kann jedoch beobachtet werden, dass sich die Werte der Tabu-Search mit der Zeit den Werten von Greedy und Puppe annähern. Die Werte von Puppe und Greedy sind ab 150 fast identisch. Davor, also zwischen 25 und 150 sind die Werte von Greedy etwas niedriger.

Kombiniert man die Ergebnisse der Akzeptanzquote und die Ergebnisse für die GesamteFahrzeit sieht man, dass die Tabu-Search sowohl die meisten Anfragen bearbeitet als auch die ge-

ringsten Werte für die GesamteFahrzeit besitzt. Das heißt also, dass es die Tabu-Search schafft, weniger zu fahren als Greedy und Puppe und trotzdem mehr Anfragen als die anderen beiden Strategien zu bearbeiten. Außerdem sieht man, dass der Greedy bei ähnlichen Werten für die GesamteFahrzeit mehr Anfragen als Puppe bearbeitet.

6.3 Anzahl genutzter Busse

Über jede untersuchte Anzahl an Anfragen hinweg, lag die AnzahlGenutzerBusse bei allen drei Transportstrategien bei 5. Das heißt, dass alle drei Transportstrategien bereits bei nur 25 Anfragen in drei Stunden 5 Busse eingesetzt haben.

6.4 Durchschnittliche Busauslastung

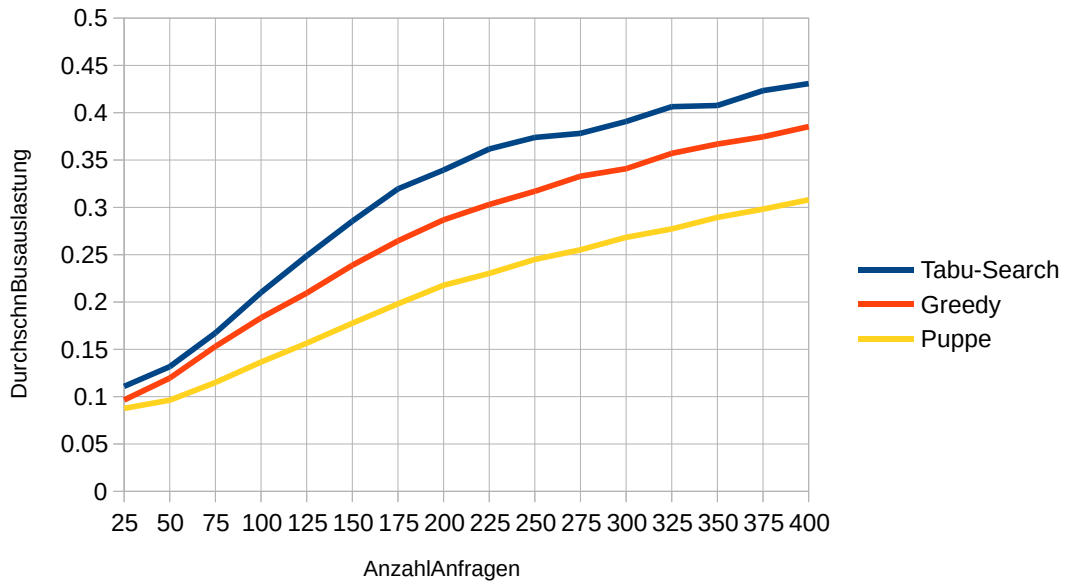


Abb. 6.3: Durchschnittliche Busauslastung in Abhängigkeit der Anzahl der Anfragen

Abbildung 6.3 zeigt die DurchschnBusauslastung im Verhältnis zur Anzahl an Anfragen. Für alle drei Strategien gilt, dass je höher die Anzahl von Anfragen, desto höher ist auch die DurchschnBusauslastung. Dabei gilt für jede Anzahl an Anfragen, dass die Tabu-Search die höchsten Werte für die höchsten und Puppe die niedrigsten Werte für die DurchschnBusauslastung erreicht. Die Werte von Greedy befinden sich zwischen den Werten der Tabu-Search und Puppe.

Für 25 Anfragen besitzen alle drei Strategien eine DurchschnBusauslastung von circa 0.1. Für die Anzahl an Anfragen zwischen 25 und 75 liegen die Werte der DurchschnBusauslastung für die Tabu-Search und den Greedy nahe beieinander. Zwischen 75 und ca. 175 steigen

die Werte der Tabu-Search schneller als die Werte von Greedy. Danach wachsen beide mit einer ähnlichen Steigung weiter. Die Werte für die Strategie von Puppe steigen generell etwas langsamer als die von Greedy und Tabu-Search.

Für 400 Anfragen erreicht die Tabu-Search einen DurchschnBusauslastung von um die 0.43, Greedy von um die 0.39 und Puppe von um die 0.31. Für die Tabu-Search und den Greedy sitzen also im Schnitt 2 Anfragen und für Puppe etwas unter 2 Anfragen im Bus.

Es lässt sich festhalten, dass Greedy und Tabu-Search eine um einiges höhere DurchschnBusauslastung besonders bei einer höheren Anzahl an Anfragen als Puppe erreichen.

6.5 Durchschnittliche Wartezeit

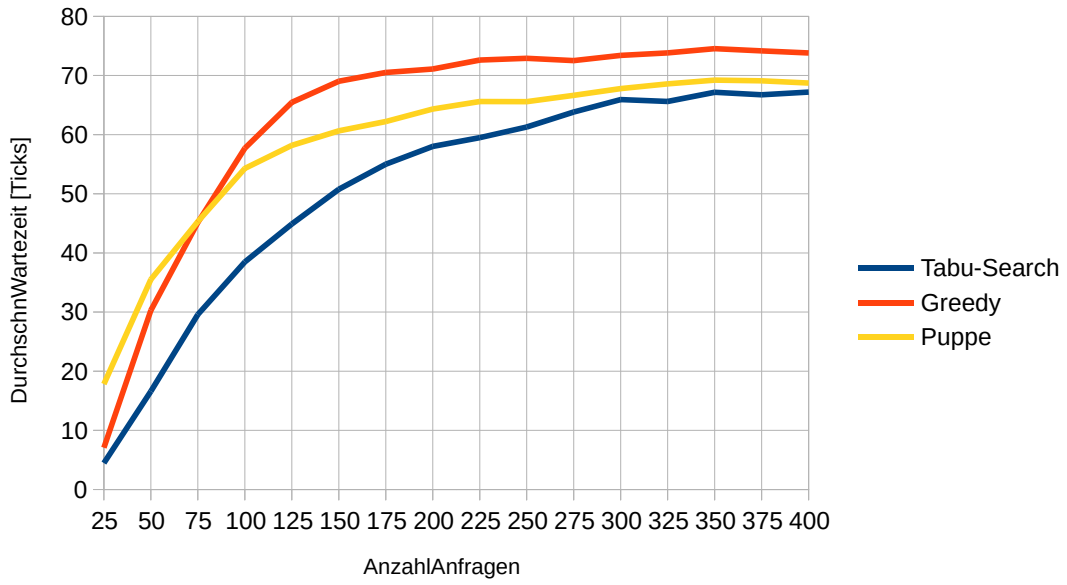


Abb. 6.4: Durchschnittliche Wartezeit in Abhängigkeit der Anzahl der Anfragen

Abbildung 6.4 zeigt die durchschnittliche Wartezeit in Abhängigkeit der Anzahl der Anfragen. Für alle drei Strategien gilt, dass je mehr Anfragen gestellt werden, desto höher ist die durchschnittliche Wartezeit. Bei allen Strategien lässt sich beobachten, dass die Werte anfangs stark ansteigen. Mit zunehmender Anzahl an Anfragen flacht dieser Anstieg bei allen drei Strategien ab.

Über jede Anzahl an Anfragen hinweg, sind die Werte der Tabu-Search unter denen der anderen beiden Strategien. Die Tabu-Search startet mit einer DurchschnWartezeit von um die 5 bei 25 Anfragen und steigt bis auf eine DurchschnWartezeit von um die 67 bei 400 Anfragen.

Die Greedy Strategie startet mit einer DurchschnWartezeit von um die 7 bei 25 und steigt bis auf eine DurchschnWartezeit von um die 74 bei 400 Anfragen.

Puppe startet mit einer DurchschnWartezeit von um die 18 bei 25 Anfragen und steigt auf eine DurchschnWartezeit von um die 69.

Dabei liegen die Werte von Greedy für das Intervall 25 bis 75 Anfragen unter den von Puppe. Für das restliche Intervall (75 bis 400 Anfragen) liegen die Werte von Greedy hingegen über denen von Puppe.

Es lässt sich also festhalten, dass die Tabu-Search die niedrigste DurchschnWartezeit liefert und Greedy anfangs niedrigere Werte als Puppe liefert, aber bei steigender Anzahl an Anfragen Puppe unterlegen ist.

6.6 Durchschnittliche Fahrzeit

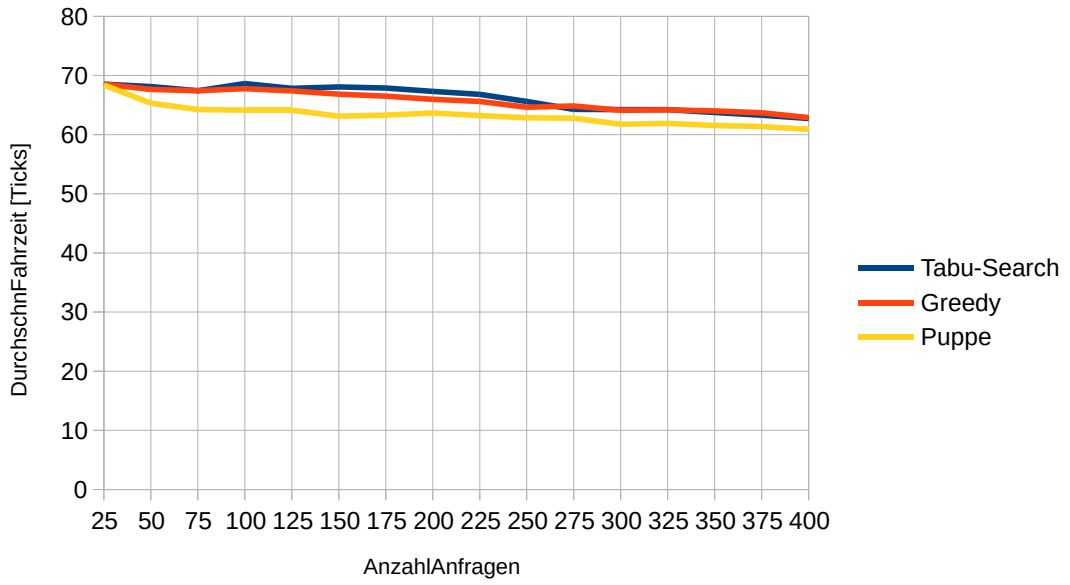


Abb. 6.5: Durchschnittliche Fahrzeit in Abhängigkeit der Anzahl der Anfragen

Abbildung 6.5 zeigt die DurchschnFahrzeit in Abhängigkeit der Anzahl an Anfragen. Für alle Strategien lässt sich erkennen, dass mehr Anfragen zu einer geringeren DurchschnFahrzeit führen. Puppe erzielt dabei für jede Anzahl an Anfragen die geringsten Werte für die DurchschnFahrzeit.

Es fällt auf, dass die Werte der Tabu-Search und des Greedys für die Mehrzahl der Anzahl an Anfragen über denen von Puppe liegen. Einzige Ausnahme sind die Werte für 25 Anfragen. Hier erzielen alle drei Strategien einen Wert von um die 68 Ticks. Zwischen 100 und 250 Anfragen liegen die Werte der Tabu-Search leicht über denen des Greedys.

6.7 Durchschnittliche Transportzeit

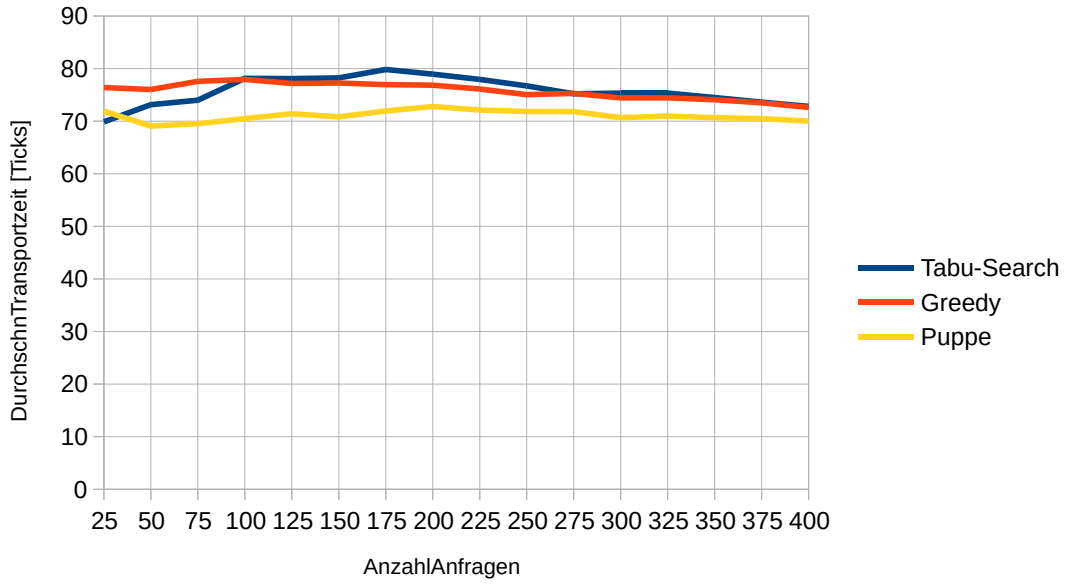


Abb. 6.6: Durchschnittliche Transportzeit in Abhängigkeit der Anzahl der Anfragen

Die durchschnittliche Transportzeit in Abhängigkeit der Anzahl an Anfragen ist in Abbildung 6.6 abgebildet. Puppe erzielt, bis auf für 25 Anfragen, die geringsten durchschnittlichen Transportzeiten aller Strategien. Für 25 Anfragen erzielt die Tabu-Search einen um etwa 2 Ticks niedrigeren Wert. Die Werte von Puppe schwanken zwischen um 69 und 73, wobei kein klarer Trend erkennbar ist. Für die Tabu-Search steigen die Werte für DurchschnTransportzeit zwischen 25 und 175 Anfragen von einem Wert um die 70 auf einen Wert von um die 80. Zwischen 175 und 400 fallen die Werte der Tabu-Search hingegen von 80 auf um die 72 Ticks.

Für die Werte des Greedys lässt sich eine leicher Abwärtstrend bei steigender Anzahl an Anfragen erkennen. Die Werte des Greedy liegen für jede Anzahl an Anfragen über denen von Puppe. Für das Intervall von 25 bis 100 Anfragen, liegen die Werte des Greedys über denen der Tabu-Search. Für das Intervall von 100 bis 275 liegen die Werte der Tabu-Search leicht über denen des Greedys. Für das Intervall 275 bis 400 besitzen die Tabu-Search und der Greedy fast identische Werte, wobei die Werte der Tabu-Search ganz leicht über denen des Greedys liegen.

6.8 Durchschnittliche Berechnungszeit

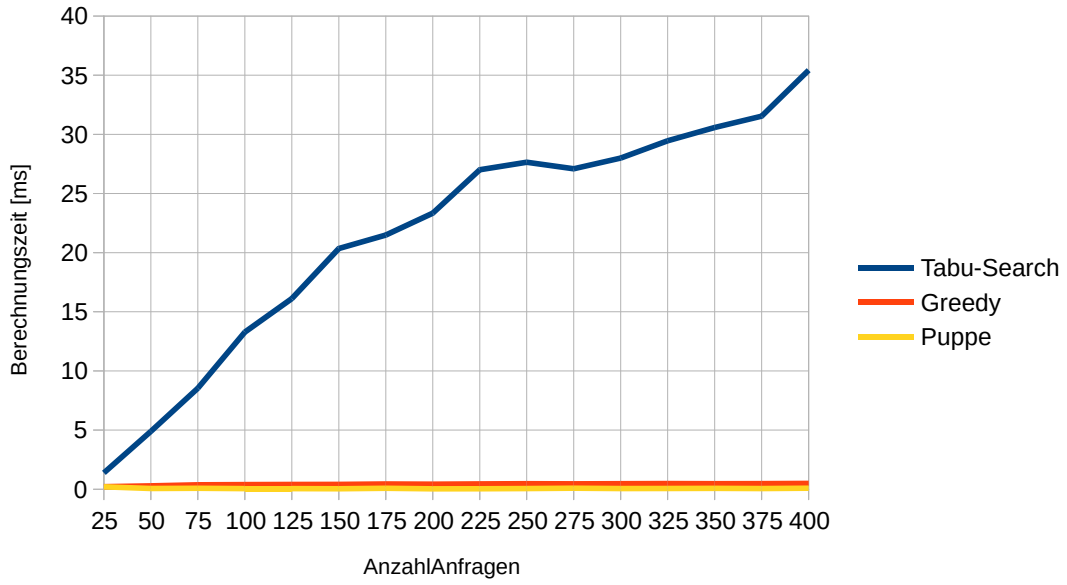


Abb. 6.7: Berechnungszeit in Abhängigkeit der Anzahl der Anfragen

Abbildung 6.7 zeigt die Berechnungszeit in Abhängigkeit der Anzahl an Anfragen. Die Werte für die Berechnungszeit von Puppe und Greedy liegen beide für alle je Anzahl an Anfragen nahe 0 Millisekunden. Für die Tabu-Search lässt sich hingegen ein klarer Aufwärtstrend erkennen. Dabei startet die Tabu-Search bei einem Wert von um die 1 Millisekunden bei 25 Anfragen und steigt auf einen Wert von um die 35 Millisekunden bei 400 Anfragen.

Um einen genaueren Einblick zu bekommen, wie lange die Tabu-Search Transportstrategie für die eigentliche Verbesserung durch die Tabu-Search benötigt, wurde für die Tabu-Search Transportstrategie zusätzlich die sogenannte TabuBerechnungszeit gemessen. Sie entspricht der Zeit in Millisekunden, die die Tabu-Search im Durchschnitt für die Verbesserung der Greedy Lösung gebraucht hat.

Abbildung 6.8 zeigt die TabuBerechnungszeit in Abhängigkeit der Anzahl von Anfragen. Hier kann erkannt werden, dass der Verlauf der TabuBerechnungszeit dem Verlauf der Berechnungszeit der Tabu-Search stark ähnelt.

Für die Instanzen mit 25 Anfragen benötigt die Tabu-Search im Schnitt 30 Millisekunden. Für Instanzen mit 400 Anfragen hingegen 700 Millisekunden.

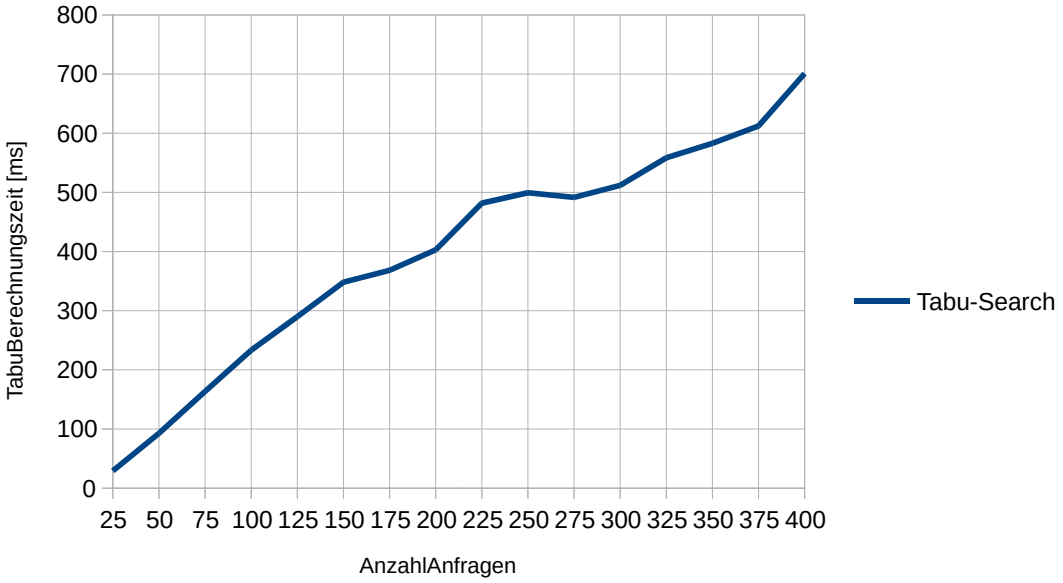


Abb. 6.8: Durchschnittliche Berechnungszeit der Tabu-Search

7 Diskussion

In diesem Abschnitt werden die Ergebnisse aus Abschnitt 6 diskutiert.

7.1 Akzeptanzquote

Dass die Akzeptanzquote mit steigender Anzahl von Anfragen sinkt, lässt sich damit erklären, dass mehr Anfragen in der gleichen Zeiteinheit transportiert werden müssen.

Als Nächstes soll untersucht werden, warum die Greedy Transportstrategie eine höhere Akzeptanzquote im Vergleich zur Strategie von Puppe besitzt. Wie Puppe bereits in seiner Arbeit [Pup23b] beschreibt, besitzt seine Strategie ein Problem im Entwurf, das dafür sorgt, dass Anfragen, die eigentlich bearbeitet werden könnten, abgelehnt werden. Das Problem tritt auf, wenn entschieden werden soll, ob eine Anfrage in einen bestehenden Aktionsplan eines Busses integriert werden kann. Die Entscheidung, ob ein Bus $b \in B$ eine Anfrage $r \in R$ integrieren kann, wird zum Zeitpunkt r^a entschieden. Zu diesem Zeitpunkt wird geschaut, ob b zum Zeitpunkt r^g in die gleiche Richtung wie die Anfrage r weiterfahren und dabei ohne Wenden an der Starthaltestelle r^o vorbeikommen wird [Pup23b]. Wie Puppe anmerkt, sorgt das dafür, dass insbesondere Anfragen, die von Haltestelle h_1 oder h_k abfahren, abgelehnt werden.

Abbildung 7.1 zeigt das Beispiel, das Puppe in seiner Arbeit [Pup23b] für die Veranschaulichung dieses Fehlers nutzt. In dieser Darstellung des Strecke-Zeit-Diagramms steht der horizontale, rot gestrichelte Strich für die Markierung der aktuellen Simulationszeit. In diesem Beispiel wird versucht, die dargestellte Anfrage in den Aktionsplan des roten Busses zu integrieren. Da der rote Bus zum gewünschten Abfahrtszeitpunkt (gelber Strich im Diagramm) der Anfrage in die falsche Richtung fährt, wird diese Anfrage abgelehnt. Wie man aber aus dem Diagramm entnehmen kann, könnte der rote Bus die Anfrage nach seinem Halt an Haltestelle 2 mitnehmen und zu ihrer Zielhaltestelle bringen.

Die Greedy Strategie dieser Arbeit würde hier zu einem anderen Ergebnis kommen. Zuerst würde sie den sichtbaren Teil der Fahrtroute des roten Busses als zwei Sublinien auffassen. Die erste Sublinie ist der Teil vor dem Halt an Haltestelle 2 und die zweite Sublinie ist der Teil nach dem Halt an Haltestelle 2. Die erste Sublinie kommt für das Einfügen der gezeigten Anfrage nicht infrage, da der Bus hier in die falsche Richtung fährt. Nun wird aber auch die zweite Sublinie betrachtet. Diese erfüllt die Anforderung, dass auf ihr der rote Bus in die richtige Richtung fährt. Danach wird ermittelt, dass sich die Start- und Zielhaltestelle der Anfrage innerhalb der Sublinie befindet (Fall 4 der Insert-Operation). Für dieses Beispiel wird dann der zusätzliche Halt an der Zielhaltestelle eingeplant und der Abfahrtsaktion an Haltestelle die Anfrage hinzugefügt.

Somit wurde die Anfrage erfolgreich in den bestehenden Aktionsplan des roten Busses integriert.

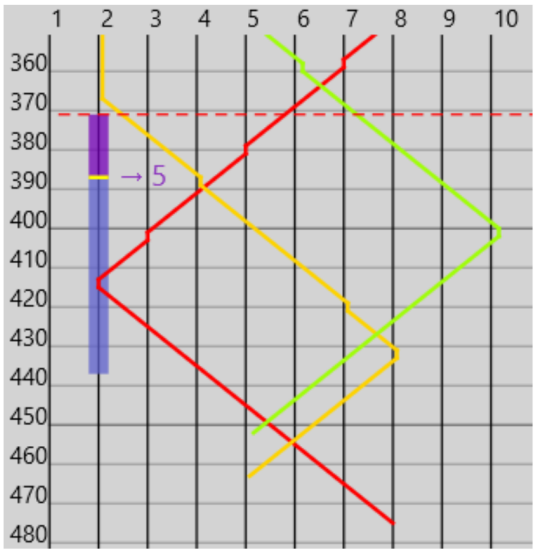


Abb. 7.1: Beispiel für den Fehler im Entwurf der Strategie von Puppe (Quelle: [Pup23b], S. 62)

Aufgrund dessen, dass die Strategie von Puppe und die Greedy Transportstrategie ähnliche Werte für die GesamteFahrzeit besitzen, kann ausgeschlossen werden, dass die Anfragen durch mehr Fahrten bearbeitet werden.

Daher ist höchstwahrscheinlich das bessere Einfügen von Anfragen in bestehende Lösungen dafür verantwortlich, dass der Greedy eine bessere Akzeptanzquote als Puppe aufweist.

Nun soll untersucht werden, warum es die Tabu-Search Transportstrategie schafft, höhere Werte für die Akzeptanzquote zu erhalten, obwohl sie intern für die Ablehnung und Akzeptierung von Anfragen auf die Greedy Strategie zurückgreift.

Die Tabu-Search kann, anders als die reine Greedy Strategie oder Puppe, bereits geplante Anfragen umplanen. Für Routing-Entscheidungen, die sich erst nach einer gewissen Zeit als schlecht herausstellen, kann die Tabu-Search so womöglich eine bessere Route finden. Es wird vermutet, dass diese besseren Routen auch dafür sorgen, dass spätere Anfragen besser eingefügt werden können.

7.2 Gesamte Fahrzeit

In diesem Abschnitt sollen die Ergebnisse für die GesamteFahrzeit untersucht werden. Zuerst soll herausgefunden werden, warum die GesamteFahrzeit anfangs für alle drei Strategien stark ansteigt und danach stagniert. Anfangs wird für jede Anfrage eine komplett eigene Fahrt geplant. Dies führt dazu, dass die leere Fahrt zur Starthaltestelle zusätzliche Fahrzeit verbraucht. Mit steigender Zahl an Anfragen kommt es immer häu-

figer vor, dass Anfragen in bereits existierende Routen integriert werden können und keine neuen Fahrten mehr geplant werden müssen. Ab einem gewissen Punkt kommt es sogar dazu, dass keine neuen Fahrten mehr geplant werden können, da für die Busse für diese Zeiträume bereits existierende Fahren bearbeiten. Hier können Anfragen dann nur noch in diese bestehenden Routen integriert werden. Daher kommt es ab einem gewissen Punkt zu einer Stagnation der Fahrzeit. Zu dieser Einschätzung kommt auch Puppe in seiner Arbeit [Pup23b].

Nun soll erklärt werden, warum die Greedy Strategie anfangs etwas weniger Fahrzeit benötigt als Puppe. Dies hängt höchstwahrscheinlich damit zusammen, dass die Greedy Strategie (wie bereits im Abschnitt über die Akzeptanzquote erklärt) Anfragen besser in bestehende Aktionspläne integrieren kann. Es wird vermutet, dass dies dazu führt, dass die Greedy Strategie bereits früher als Puppe anfängt, Anfragen in bestehende Routen zu integrieren. Dies würde dazu führen, dass der Greedy extra Fahrzeit zu den Starthaltestellen einsparen könnte.

Im nächsten Schritt soll erklärt werden, warum die Tabu-Search für jede Anzahl an Anfragen die niedrigsten Werte für die GesamteFahrzeit besitzt. Anders als die Strategie von Puppe oder der Greedy optimiert die Tabu-Search Transportstrategie gezielt auf den Parameter GesamteFahrzeit, vergleiche dazu die Bewertungsfunktion der Tabu-Search Transportstrategie. Die Tabu-Search schafft es damit, trotz höchster Akzeptanzquote aller drei Strategien, die geringste GesamteFahrzeit zu besitzen. Es scheint aber so, dass dieser Vorteil mit der Zeit abnimmt und sich die Werte der Tabu-Search den Werten von Puppe und Greedy annähern.

7.3 Busauslastung

Zuerst soll untersucht werden, warum, mit steigender Anzahl an Anfragen, die DurchschnBusauslastung für alle drei Strategien steigt. Das hängt damit zusammen, dass Busse ab einer gewissen Anzahl an Anfragen bereits Aktionspläne für den gesamten Zeitraum haben. Werden nun mehr Anfragen gestellt, so kommt es öfter dazu, dass eine neue Anfrage zufälligerweise in einen bestehenden Aktionsplan eingefügt werden kann. Je mehr Anfragen schon in die Aktionspläne eingefügt wurden, desto schwieriger wird es, neue Anfragen zu finden, die noch eingefügt werden können. Daher steigt zwar die Busauslastung, aber gleichzeitig sinkt auch die Akzeptanzquote.

Um diese Theorie weiter zu untersuchen, wurde das Experiment einer Anzahl an Anfragen von 500 bis 10000 wiederholt.

Abbildung 7.2 zeigt die Ergebnisse für die DurchschnBusauslastung. Man sieht, dass mit steigender Anzahl an Anfragen auch die Akzeptanzquote bis zu einem Wert von 0.9 ansteigt. Es fällt auf, dass die DurchschnBusauslastung der Tabu-Search und des Greedys schneller ansteigt als die von Puppe. Für 10000 Anfragen erreichen aber alle drei Strategien einen Wert um die 0.9.

Es wird vermutet, dass die Werte der DurchschnBusauslastung nach 10000 weiter langsam ansteigen würden, bis sie eine DurchschnBusauslastung von 1.0 erreichen würden. Dieser Anstieg würde aber höchstwahrscheinlich sehr langsam sein.

Abbildung 7.3 zeigt die dazugehörige Akzeptanzquote. Es fällt auf, dass, wie vermutet, die Akzeptanzquote mit steigender Anzahl an Anfragen stark abfällt. Dies unterstützt zusätzlich die oben entwickelte Theorie.

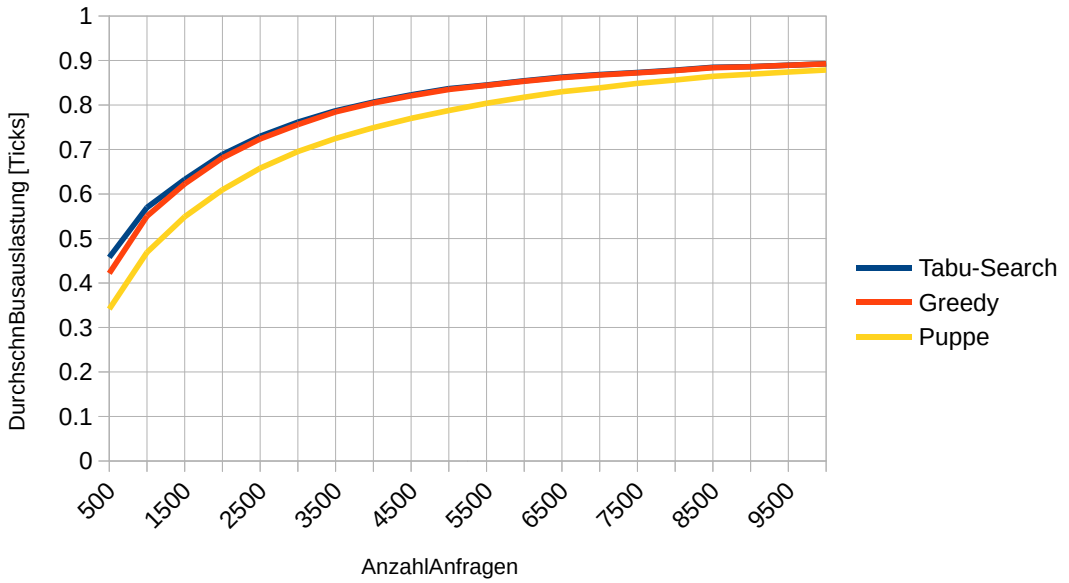


Abb. 7.2: Busauslastung in Abhängigkeit der Anzahl an Anfragen

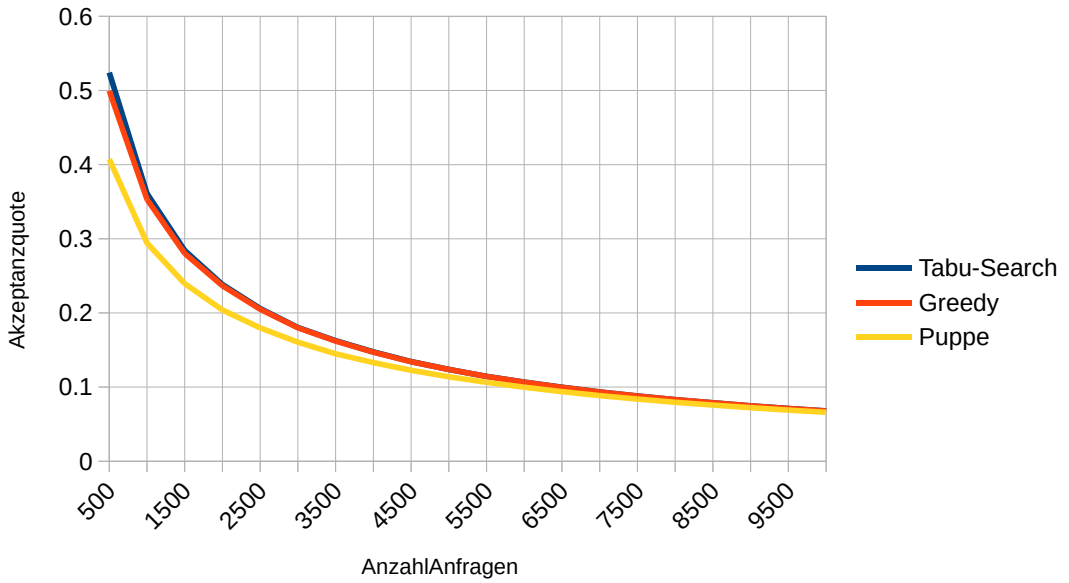


Abb. 7.3: Akzeptanzquote in Abhängigkeit der Anzahl an Anfragen

Nun soll erklärt werden, warum die Greedy Strategie eine höhere DurchschnBusauslastung erreicht als die Strategie von Puppe. Wie bereits gesehen, besitzen der Greedy und die Strategie von Puppe eine ähnliche GesamteFahrzeit. Der Greedy schafft es aber, mit einer ähnlichen GesamteFahrzeit mehr Anfragen zu transportieren. Dadurch steigt die DurchschnBusauslastung, weil, um mehr Anfragen bei gleicher Fahrzeit transportieren zu können, müssen sich im Durchschnitt mehr Anfragen in einem Bus befinden. Was nichts anderes als die DurchschnBusauslastung ist. Die Tabu-Search schafft es, noch mehr Anfragen als der Greedy zu transportieren und gleichzeitig weniger zu fahren. Daraus ergibt sich, dass die DurchschnBusauslastung der Tabu-Search höher ist als die von Greedy. Gut erkennt man das für 75 Anfragen. Ab hier beginnt die Tabu-Search mehr Anfragen als die Greedy Strategie mitzunehmen bei gleichzeitig geringerer Fahrzeit. Dies erklärt auch, warum ab dieser Anzahl an Anfragen die Werte für die Tabu-Search deutlich stärker wachsen als die des Greedys.

7.4 Anzahl genutzter Busse

In diesem Abschnitt soll geklärt werden, warum die AnzahlGenutzerBusse über jede Anzahl an Anfragen und alle Strategien hinweg bei 5 liegt. Es wird vermutet, dass es sich bereits ab 25 Anfragen für alle drei Strategien lohnt, alle verfügbaren Busse einzusetzen.

Um zu untersuchen, was für weniger als 25 Anfragen passiert, wurde das Experiment für 1 bis 25 Anfragen wiederholt. Abbildung 7.4 zeigt die Anzahl genutzter Busse für die Anzahl von Anfragen zwischen 1 und 25. Es zeigt sich, dass alle Strategien zu Beginn weniger als fünf Busse nutzen. Dies ist auch nicht verwunderlich, da, wenn zum Beispiel nur eine einzige Anfrage bearbeitet werden muss, auch nur genau ein Bus eingesetzt werden kann. Interessanterweise zeigt sich aber auch, dass die Tabu-Search bereits früher als die anderen beiden Strategien mehr Busse einsetzt. Wenn man sich für dieses Experiment die Werte für die GesamteFahrzeit anschaut (siehe Abbildung 7.5) fällt weiter auf, dass die Tabu-Search zwar mehr Busse einsetzt, aber dafür gleich viel oder etwas weniger fährt als die anderen beiden Strategien.

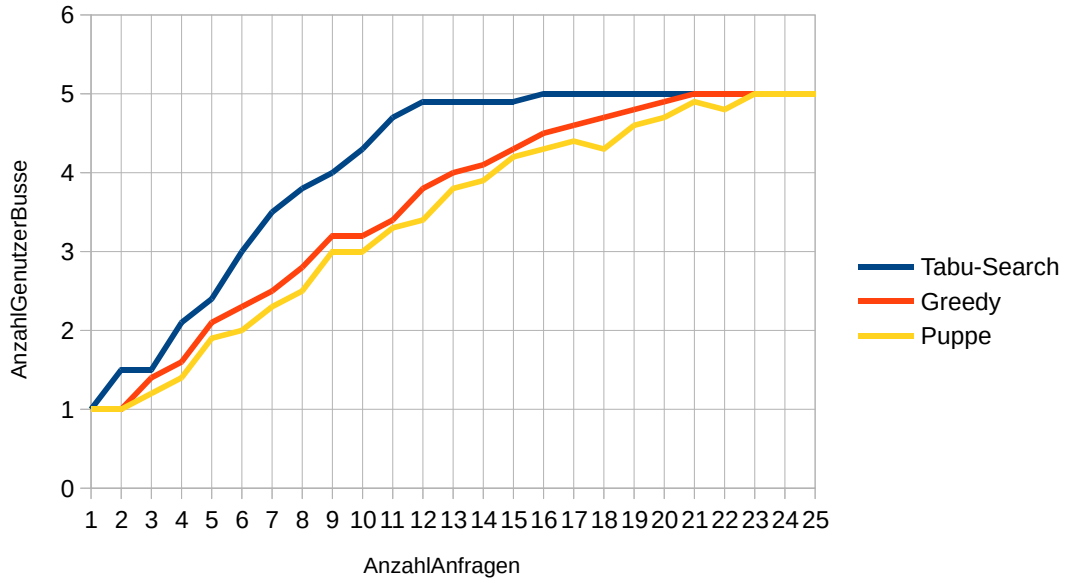


Abb. 7.4: Anzahl genutzter Busse für eine Anzahl an Anfragen zwischen 1 und 25

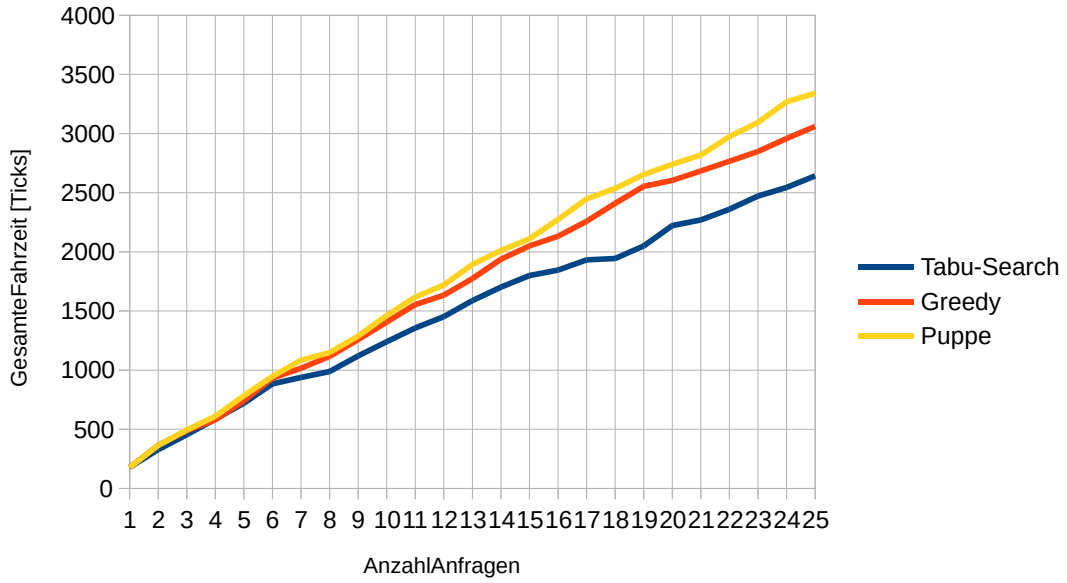


Abb. 7.5: Gesamte Fahrzeit für eine Anzahl an Anfragen zwischen 1 und 25

Dies funktioniert deshalb, da Busse am Ende der Simulation bei ihrer letzten Haltestelle stehen bleiben und nicht wieder zurück zur ersten Haltestelle zurückfahren. Die benötigte Fahrzeit, um zurück zur ersten Haltestelle zu fahren, wird also unterschlagen.

Wenn am Ende der Simulation eine Anfrage nahe der Haltestelle h_1 bearbeitet werden muss, fällt es weniger ins Gewicht, einen noch unbenutzten Bus von h_1 starten zu lassen und nur diese Anfrage zu bearbeiten. Hierbei spart man sich, dass einer der anderen Busse, der unter Umständen am anderen Ende der Buslinie steht, zurückfahren muss, um diese Anfrage zu bearbeiten. Da die Tabu-Search die einzige der drei Strategien ist, die versucht, die GesamteFahrzeit zu optimieren, scheint dieses Problem nur für die Tabu-Search aufzutreten.

7.5 Wartezeit

Zuerst soll geklärt werden, warum die DurchschnWartezeit besonders am Anfang ansteigt und danach eher stagniert. Dieser Effekt konnte bereits von Puppe [Pup23b] beobachtet werden. Puppe argumentiert, dass der Anstieg damit zusammenhängt, dass, wenn mehr Anfragen gestellt werden, auch mehr Anfragen in bestehende Routen integriert werden müssen. Wenn eine Anfrage in eine bestehende Route integriert wird, kann die DurchschnWartezeit seltener optimiert werden. Wird hingegen eine neue Fahrt geplant, so kann die Wartezeit besser optimiert werden [Pup23b]. Diese Argumentation würde erklären, warum die DurchschnWartezeit mit steigender Anzahl von Anfragen steigt.

Nun soll untersucht werden, warum der Greedy ab einer Anzahl von 75 Anfragen eine höhere DurchschnWartezeit als Puppe besitzt. Dies hängt höchstwahrscheinlich damit zusammen, dass der Greedy ab dieser Anzahl an Anfragen eine wesentlich höhere Akzeptanzquote besitzt. Mehr Anfragen führen wieder dazu, dass es häufiger vorkommt, dass Anfragen in bestehende Routen integriert werden müssen, wodurch die DurchschnWartezeit steigt.

Dass der Greedy für weniger als 75 Anfragen eine sehr geringe DurchschnWartezeit besitzt, hängt damit zusammen, dass der Greedy äußerst stark auf diesen Parameter optimiert. In Abschnitt 7.7 wird erklärt, warum dieses Verhalten nicht unbedingt vorteilhaft ist.

Als Letztes soll noch untersucht werden, warum es die Tabu-Search schafft, für jede Anzahl an Anfragen die niedrigsten DurchschnWartezeit zu erreichen. Diese Beobachtung lässt sich höchstwahrscheinlich, wie schon bei der GesamteFahrzeit, auf die bessere Optimierungsfähigkeit der Tabu-Search zurückführen.

7.6 Fahrzeit

In diesem Abschnitt sollen die Ergebnisse für DurchschnFahrzeit untersucht werden. Wie Puppe bereits in seiner Arbeit [Pup23b] feststellt, ist die Fahrzeit einer Anfrage proportional zur Streckenlänge, die diese Anfrage transportiert werden möchte. Aufgrund dessen, dass für diese Arbeit Abkürzungen in der Problemdefinition ausgeschlossen wurden, entspricht die Fahrzeit einer Anfrage immer genau der Zeit, die zum Fahren zwischen der Start- und Zielhaltestelle einer Anfrage benötigt wird. Aus der DurchschnFahrzeit lassen sich daher Rückschlüsse ziehen, wie weit die Start- und Zielhaltestellen der akzeptierten Haltestellen im Durchschnitt auseinander liegen. Sinkt etwa die DurchschnFahrzeit,

so kann man daraus schließen, dass nun eher Anfragen bearbeitet werden, die einen geringeren Abstand zwischen der Start- und Zielhaltestelle besitzen.

Wie bereits im Abschnitt über die Auslastung beschrieben, können die Greedy und Tabu-Search Strategie besser mit Anfragen umgehen, die an den „Rändern“ starten. Daher erklärt sich auch, warum die DurchschnFahrzeit dieser beiden Strategien höher ist als die von Puppe.

Diese Argumentation allein, kann aber bisher nicht erklären, warum es auch bei der Greedy und Tabu-Search Strategie zu einem Absinken der DurchschnFahrzeit mit steigender Anzahl an Anfragen kommt. Um eine Anfrage r von Haltestelle h_1 zur letzten Haltestelle h_k zu transportieren, wird für die gesamte Strecke ein freier Sitzplatz benötigt. Da aber mit steigender Anzahl an Anfragen die Wahrscheinlichkeit steigt, dass für einen Streckenabschnitt der Bus bereits voll ausgelastet ist, kann eine Anfrage, die sehr weit fahren muss, nicht mehr bearbeitet werden. Dies erklärt, warum auch für die Greedy und Tabu-Search zu einem Absinken der DurchschnFahrzeit kommt.

7.7 Transportzeit

In diesem Abschnitt sollen die Ergebnisse für die DurchschnTransportzeit untersucht werden.

Wie auch in der Arbeit von Puppe [Pup23b] soll für die Analyse der DurchschnTransportzeit die DurchschnStandzeit eingeführt werden. Die DurchschnStandzeit ist die Zeit, die eine Anfrage durchschnittlich warten muss, während sie in einem Bus sitzt. Sie berechnet sich aus der Differenz der DurchschnTransportzeit und der DurchschnFahrzeit.

Abbildung 7.6 zeigt die Werte für die DurchschnStandzeit in Abhängigkeit der Anzahl an Anfragen in der Einheit Simulationsticks.

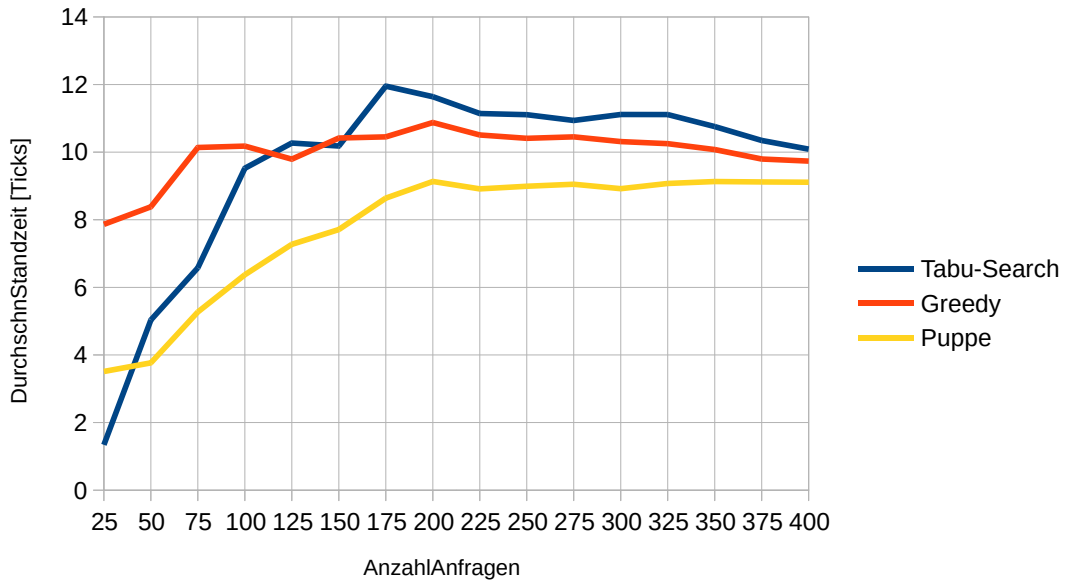


Abb. 7.6: Durchschnittliche Standzeit in Abhängigkeit der Anzahl an Anfragen

Zuerst soll anhand der DurchschnStandzeit erklärt werden, warum die DurchschnTransportzeit der Tabu-Search zwischen 25 und 175 Anfragen ansteigt. Wie man in Abbildung 7.6 sehen kann, steigt genau im gleichen Intervall die DurchschnStandzeit stark an. Da die DurchschnFahrzeit erst nach 175 abfällt, sorgt der Anstieg der DurchschnStandzeit für den Anstieg der DurchschnTransportzeit. Dass die Standzeit bei einer höheren Anzahl an Anfragen steigt, ist nicht verwunderlich. Wie bereits Puppe [Pup23b] feststellt, führt eine Erhöhung der Anfragen dazu, dass mehr Zwischenhalte für die neuen Anfragen eingeplant werden müssen. Dies sorgt dafür, dass Anfragen, die sich bereits im Bus befanden, durch einen Zwischenhalt länger im Bus warten müssen. Dass die DurchschnStandzeit ab 175 Anfragen fällt, lässt sich damit erklären, dass ab einem gewissen Punkt keine zusätzlichen Zwischenhalte benötigt werden, da der jeweilige Bus bereits an dieser Haltestelle hält. Da aber gleichzeitig die Menge akzeptierter Anfragen steigt, sinkt die DurchschnStandzeit. Dieses Absinken ist ein Grund, warum auch die DurchschnTransportzeit der Tabu-Search ab 175 Anfragen absinkt. Ein weiterer Grund ist, dass die DurchschnFahrzeit aller Strategien mit der Zeit sinkt. Dies sorgt auch bei den anderen Strategien dafür, dass die Werte absinken.

Nun soll erklärt werden, warum der Greedy eine höhere DurchschnTransportzeit besitzt als Puppe. Auch hier lohnt sich ein Blick auf die DurchschnStandzeit. Es fällt auf, dass der Greedy für jede Anzahl an Anfragen eine höhere DurchschnStandzeit besitzt. Dies scheint auch die Ursache dafür zu sein, dass die DurchschnTransportzeit beim Greedy höher ist als bei Puppe. Dass der Greedy allgemein eine höhere DurchschnStandzeit besitzt, lässt sich höchstwahrscheinlich damit erklären, dass der Greedy mehr Anfragen als Puppe transportiert. Für 25 Anfragen kann das aber nicht zutreffen, da hier Puppe und Greedy fast identische Werte für die Akzeptanzquote und DurchschnBusauslastung

besitzen. Interessant ist also, woher die höhere Standzeit bei 25 Anfragen kommt. Wie Puppe in seiner Arbeit [Pup23b] beschreibt, unterliegt seine Strategie einem Fehler, der dafür sorgt, dass die DurchschnStandzeit mit steigender Zeit dazu tendiert, besonders lange auf Anfragen zu warten. Dieses Verhalten lässt sich auch in Abbildung 7.6 beobachten. Der Greedy besitzt ebenfalls ein ähnliches Problem, das dafür sorgt, dass die DurchschnStandzeit insbesondere bei wenigen Anfragen stark erhöht ist.

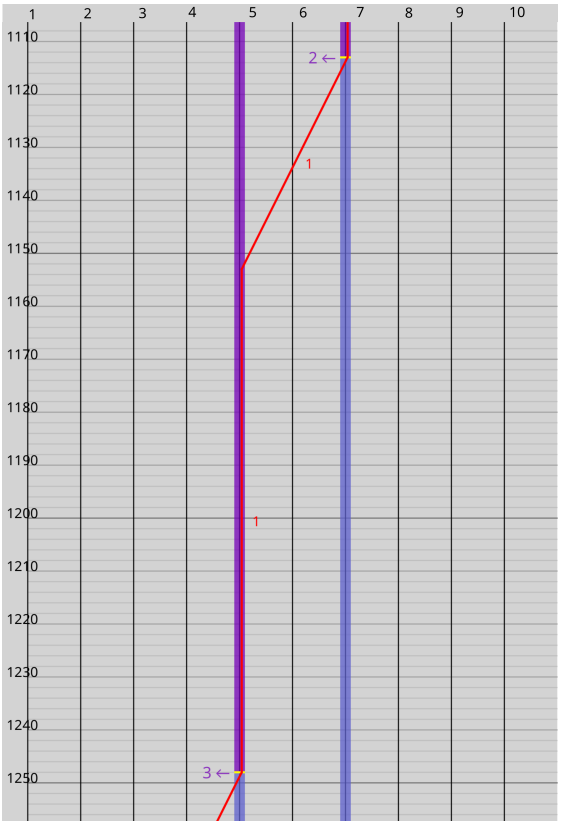


Abb. 7.7: Beispiel für das Problem der Greedy Strategie

Abbildung 7.7 zeigt ein Beispiel für dieses Problem. Man sieht in diesem Beispiel, wie die Greedy Strategie einen Bus geplant hat. Der Bus holt zuerst eine Abfrage an h_7 ab und fährt danach mit dieser Anfrage an Bord zu h_5. Der Abfahrtszeitpunkt von h_5 entspricht dabei genau dem gewünschten Abfahrtszeitpunkt dieser Anfrage. Dort wartet der Bus etwa 95 Simulationsticks, was 9.5 Minuten entspricht. Danach lädt er eine weitere Anfrage ein, die zu diesem Zeitpunkt ihren gewünschten Abfahrtszeitpunkt besitzt. Dies sorgt unnötigerweise dafür, dass die erste Anfrage sehr lange im Bus warten muss und erst spät an ihrer Zielhaltestelle ankommt. Insbesondere hätte der Bus, statt zu warten, die erste Anfrage zu ihrer Zielhaltestelle und danach zu h_5 fahren können, um die zweite Anfrage abzuholen. Dies hätte zwar dazu geführt, dass die zweite Anfrage etwas später abgeholt worden wäre, das wäre aber für die Kundenzufriedenheit aller Kunden die bessere Lösung gewesen. Dieses Problem entsteht, da der Greedy

nur auf die DurchschnWartezeit optimiert. In diesem Fall ist es dem Greedy gelungen, beide Anfragen mit einer Wartezeit von 0 abzuholen, da beide Anfragen genau zu ihrem gewünschten Abfahrtszeitpunkt mitgenommen werden. Die Optimierung der DurchschnWartezeit vernachlässigt dabei aber, dass die DurchschStandzeit unnötig stark ansteigt und infolgedessen auch die DurchschnTransportzeit. Da die Tabu-Search neben der DurchschnWartezeit auch die DurchschnTransportzeit optimiert, kommt es bei ihr nicht zu diesem Problem. Eine mögliche Lösung für den Greedy könnte daher sein, eine Bewertungsfunktion zu nutzen, die aus mehr als einem Faktor besteht.

7.8 Berechnungszeit

Es ist nicht verwunderlich, dass die Tabu-Search über alle Anfragen hinweg die höchste Berechnungszeit aufweist, da die Tabu-Search im Vergleich zu Puppe und Greedy mehr Berechnungen durchführt.

Dass die Berechnungszeit bei einer Erhöhung der Anfragen steigt, lässt sich damit erklären, dass die Größe der Nachbarschaft steigt und somit in jeder Iteration mehr potenzielle Lösungen betrachtet werden müssen.

Interessant ist jedoch, dass auch für 400 Anfragen die TabuBerechnungszeit unter einer Sekunde liegt. Dies zeigt, dass sich die vorgestellte Tabu-Search für den Einsatz in einem dynamischen Kontext eignet.

8 Fazit

Diese Arbeit sollte untersuchen, ob sich eine Tabu-Search für das 2P-LiDyDARP eignet. Dafür wurden zuerst Methoden entwickelt, die es ermöglichen, Anfragen in bestehende Lösung einzufügen, aus bestehenden Lösungen zu löschen und zwischen Bussen einer bestehenden Lösung zu tauschen. Diese Methoden wurde zuerst genutzt, um eine verbesserte Greedy Strategie zu entwickeln, die es schafft, im Vergleich zu einer existierenden Greedy Strategie von Puppe [Pup23b] mehr Anfragen bei gleicher Fahrzeit der Busse zu bearbeiten. In einem zweiten Schritt wurde eine Tabu-Search entwickelt, die Lösungen des von dieser Arbeit entwickelten Greedy-Algorithmus nimmt und weiter verbessert. Es konnte gezeigt werden, dass durch die Tabu-Search ein noch höherer Anteil an Anfragen bearbeitet werden kann und gleichzeitig die Fahrzeit der Busse sinkt. Weiter konnte gezeigt werden, dass die Tabu-Search eine so geringe Berechnungszeit besitzt, dass sie auch für einen dynamischen Kontext eingesetzt werden kann.

Diese Arbeit hat die entwickelten Strategien auf Instanzen mit räumlich und zeitlich gleich verteilten Anfragen getestet. Weiter war der Abstand zwischen Haltestellen auf eine einheitliche Länge festgelegt. Diese Bedingungen sind in der echten Welt nur selten bis überhaupt nicht anzutreffen. Daher sollten die entwickelten Strategien auch mit Daten getestet werden, die so entworfen wurden, dass sie ein realistisches Szenario abbilden oder die direkt in der echten Welt erhoben wurden. Weiter wäre es interessant, zu untersuchen, welchen Einfluss die einzelnen Parameter der Tabu-Search auf die Ergebnisse hätten. Gleiches gilt dafür, welchen Einfluss die Bewertungsfunktion auf das Ergebnis hat. Es könnte auch untersucht werden, welchen Einfluss die Transportstrategie aus Phase-1 hat und welchen Unterschied es macht, wie hoch der Anteil an Anfragen in Phase-1 ist. Als Letztes könnte man noch untersuchen, welchen Effekt es hat, wenn Busse auch Abkürzungen zwischen Haltestellen nutzen können.

Literaturverzeichnis

[AAk18] ADAC e.V., Ressort Verkehr, ADAC S.E., Markt- und Meinungsforschung und komma Forschungs- und Beratungsgesellschaft mbH: Mobilität auf dem Land: Nachholbedarf beim ÖV. https://www.adac.de/verkehr/standpunkte-studien/mobilitaets-trends/mobilitaet-land/, November 2018.

[ACGL04] Andrea Attanasio, Jean François Cordeau, Gianpaolo Ghiani und Gilbert Laporte: Parallel Tabu Search Heuristics for the Dynamic Multi-Vehicle Dial-a-Ride Problem. *Parallel Computing*, 30(3):377–387, März 2004, 10.1016/j.parco.2003.12.001, ISSN 0167-8191.

[BLMN10] Alexandre Beaudry, Gilbert Laporte, Teresa Melo und Stefan Nickel: Dynamic Transportation of Patients in Hospitals. *OR Spectrum*, 32(1):77–107, Januar 2010, 10.1007/s00291-008-0135-6, ISSN 1436-6304.

[CMNG05] Teodor Gabriel Crainic, Federico Malucelli, Maddalena Nonato und François Guertin: Meta-Heuristics for a Class of Demand-Responsive Transit Systems. *INFORMS Journal on Computing*, 17(1):10–24, Februar 2005, 10.1287/ijoc.1030.0051, ISSN 1091-9856, 1526-5528.

[DS14] Karl F. Doerner und Juan José Salazar-González: Chapter 7: Pickup-and-Delivery Problems for People Transportation. In: Paolo Toth und Daniele Vigo (Herausgeber): *Vehicle Routing*, Seiten 193–212. Society for Industrial and Applied Mathematics, Philadelphia, PA, November 2014, ISBN 978-1-61197-358-7 978-1-61197-359-4, 10.1137/1.9781611973594.ch7.

[Gen03] Michel Gendreau: An Introduction to Tabu Search. In: Fred Glover und Gary A. Kochenberger (Herausgeber): *Handbook of Metaheuristics*, Band 57, Seiten 37–54. Kluwer Academic Publishers, Boston, 2003, ISBN 978-1-4020-7263-5, 10.1007/0-306-48056-5_2.

[GKP+23] Daniela Gaul, Kathrin Klamroth, Christian Pfeiffer, Arne Schulz und Michael Stiglmayr: A Tight Formulation for the Dial-a-Ride Problem. 2023, 10.48550/ARXIV.2308.11285.

[GKS21] Daniela Gaul, Kathrin Klamroth und Michael Stiglmayr: Solving the Dynamic Dial-a-Ride Problem Using a Rolling-Horizon Event-Based Graph. In: *21st Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2021)*, September 2021.

[Hei23] Lucas Hein: Line Based Minibus Simulation V2 · GitLab. https://gitlab2.informatik.uni-wuerzburg.de/s383860/line-based-minibus-simulation-v-2, Dezember 2023.

[Hei24] Lucas Hein: Tabu-Search-Für-Das-Linienbasierte-Dynamische-Darp · GitLab. https://gitlab2.informatik.uni-wuerzburg.de/s383860/tabu-search-fuer-das-linienbasierte-dynamische-darp, April 2024.

[jav] Java Development Kit Version 22 API Specification. https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/util/Collection.html#unmodifiable.

[LLMS21] Christian Liebchen, Martin Lehnert, Christian Mehlert und Martin Schiefelbusch: Betriebliche Effizienzgrößen für Ridepooling-Systeme. In: Heike Proff (Herausgeber): *Making Connected Mobility Work*, Seiten 135–150. Springer Fachmedien Wiesbaden, Wiesbaden, 2021, ISBN 978-3-658-32265-6 978-3-658-32266-3, 10.1007/978-3-658-32266-3$_7$.

[MBC17] Yves Molenbruch, Kris Braekers und An Caris: Typology and Literature Review for Dial-a-Ride Problems. *Annals of Operations Research*, 259(1):295–325, Dezember 2017, 10.1007/s10479-017-2525-0, ISSN 1572-9338.

[MRR95] Oli B. G. Madsen, Hans F. Ravn und Jens Moberg Rygaard: A Heuristic Algorithm for a Dial-a-Ride Problem with Time Windows, Multiple Capacities, and Multiple Objectives. *Annals of Operations Research*, 60(1):193–208, Dezember 1995, 10.1007/BF02031946, ISSN 1572-9338.

[Pir96] Marc Pirlot: General Local Search Methods. *European Journal of Operational Research*, 92(3):493–511, August 1996, 10.1016/0377-2217(96)00007-0, ISSN 03772217.

[Pup23a] Leo Puppe: Line Based Minibus Simulation · GitLab. https://gitlab2.informatik.uni-wuerzburg.de/s411716/line-based-minibus-simulation, November 2023.

[Pup23b] Leo Puppe: Vergleich von Einer Anfragebasierten Und Fahrplanbasierten Minibuslinie Durch Simulation, August 2023.

[VMA+22] Pieter Vansteenwegen, Lissa Melis, Dilay Aktaş, Bryan David Galarza Montenegro, Fábio Sartori Vieira und Kenneth Sörensen: A Survey on Demand-Responsive Public Bus Systems. *Transportation Research Part C: Emerging Technologies*, 137:103573, April 2022, 10.1016/j.trc.2022.103573, ISSN 0968090X.

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Versicherung zur selbständigen Leistungserbringung

Titel der Bachelorarbeit:

TABU SEARCH FÜR DAS LINIENBASIERTE DYNAMISCHE DARP

Thema bereitgestellt von (Titel, Vorname, Nachname, Lehrstuhl):

Prof. Dr. Marie Schmidt, Lehrstuhl für Informatik I

Eingereicht durch (Vorname, Nachname, Matrikel):

Lucas Hein, 2448646

Ich versichere, dass ich die vorstehende schriftliche Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die benutzte Literatur sowie sonstige Hilfsquellen sind vollständig angegeben. Wörtlich oder dem Sinne nach dem Schrifttum oder dem Internet entnommene Stellen sind unter Angabe der Quelle kenntlich gemacht.

Weitere Personen waren an der geistigen Leistung der vorliegenden Arbeit nicht beteiligt. Insbesondere habe ich nicht die Hilfe eines Ghostwriters oder einer Ghostwriting-Agentur in Anspruch genommen. Dritte haben von mir weder unmittelbar noch mittelbar Geld oder geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Arbeit stehen.

☐ Mit dem Prüfungsleiter bzw. der Prüfungsleiterin wurde abgestimmt, dass für die Erstellung der vorgelegten schriftlichen Arbeit Chatbots (insbesondere ChatGPT) bzw. allgemein solche Programme, die anstelle meiner Person die Aufgabenstellung der Prüfung bzw. Teile derselben bearbeiten könnten, entsprechend den Vorgaben der Prüfungsleiterin bzw. des Prüfungsleiters eingesetzt wurden. Die mittels Chatbots erstellten Passagen sind als solche gekennzeichnet.

Der Durchführung einer elektronischen Plagiatsprüfung stimme ich hiermit zu. Die eingereichte elektronische Fassung der Arbeit ist vollständig. Mir ist bewusst, dass nachträgliche Ergänzungen ausgeschlossen sind.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht. Ich bin mir bewusst, dass eine unwahre Erklärung zur Versicherung der selbstständigen Leistungserbringung rechtliche Folgen haben kann.

Würzburg, den 01.05.24 Lucas Hein

Ort, Datum, Unterschrift