Modellbasierte Parallelisierung von Anwendungen zur Verkehrssimulation - Ein dynamischer und adaptiver Ansatz -

Inaugural-Dissertation

zur

Erlangung des Doktorgrades

der Mathematisch-Naturwissenschaftlichen Fakultät
der Universität zu Köln

Vorgelegt von

Oliver Ullrich
aus Duisburg

Berichterstatter:	Prof. Dr. Ewald Speckenmeyer
(Gutachter)	Prof. Dr. Ulrich Lang
Tag der mündlichen Prüfung:	22. Januar 2014

Danksagungen

Mein besonderer Dank gilt meinem (zukünftigen) Doktorvater Prof. Dr. Ewald Speckenmeyer für die Chance im Rahmen des Projekts CATS über dieses interessante Thema promovieren zu dürfen, die engagierte Betreuung, seine stete Diskussionsbereitschaft und die vielen richtungsweisenden Impulse. Ich habe viel von ihm gelernt.

Mein Dank gilt weiterhin Prof. Dr. Ulrich Lang, der sich bereit erklärt hat, diese Arbeit als Zweitgutachter zu prüfen.

Meiner Kollegin Gabriele Eslamipour und meinen Kollegen Patrick Kuckertz, Dr. Mile Lemaic, Manuel Molina Madrid, Prof. Dr. Stefan Porschen, Prof. Dr. Bert Randerath, Felix Werth und Dr. Andreas Wotzlaw möchte ich danken für die gute Zusammenarbeit und die gemeinsamen Diskussionen, die mir in vielfältiger Weise geholfen haben.

Prof. Dr. Martina Joisten und Felix Werth haben diese Arbeit kritisch durchgesehen, vielen Dank dafür! Die verbliebenen Fehler und Ungenauigkeiten sind vollständig auf meinem Mist gewachsen.

Zuletzt möchte ich noch meinem Kollegen Daniel Lückerath danken für die nie langweilig werdende, gemeinsame Arbeit an Artikeln und Vorträgen, für seine engagierte Hilfsbereitschaft und insbesondere für seinen Beistand bei so mancher ewig erscheinender Debugging-Sitzung. Ich wünsche ihm viel Erfolg bei der Arbeit am Nachfolgeprojekt INSTEP.

Abstract

The project *Computer Aided Tram Scheduling (CATS)* is concerned with the design and implementation of methods and software tools that enable transport providers to generate, simulate, and evaluate tram schedules on their own notebook or desktop computers. The scientific interest lies in the research and development of the required concepts and methods, these are mainly based in the areas of mathematical optimization and discrete simulation. In this context originated the desire for a parallel simulation software, which would be more accurate and also faster than the existing sequential model.

This thesis attends to three main goals:

At first a method for the parallel execution of discrete simulation models will be devised, which should utilize properties of (amongst others) traffic simulation models. To use the resources of the target platforms to the full extend, the approach should include a dynamic and adaptive load balancing scheme. The method will be implemented as a software framework, its dynamic behavior will be examined by conducting experiments on example applications.

Furthermore, the software tools designed and implemented for project CATS will be presented. The thesis concentrates on a simulation module based on the described framework, which can be utilized to examine dynamic properties of tram schedules. To find candidates for this, an optimization module should be applied which generates robust tram schedules that also adhere to given sets of transport planning requirements.

At last, the presented software tools should be applied to the tram networks of the cities of Cologne, Germany, and Montpellier, France. The optimizer should be applied to generate both robust and non-robust schedules for these networks, which will be simulated, evaluated, and compared to the schedules applied by the tram network providers.

The thesis begins with an introduction of context, motivation and aims (chapter 1), followed by some background on methods of the parallel execution of discrete simulation models (chapter 2). After this a method is proposed which is especially suitable for the parallel execution of discrete traffic simulation models. The section begins by outlining the ideas of the method and its realization, followed by some thoughts on its scalability and efficiency (chapter 3). The thesis then continues by showing an implementation of the

method as a software framework for simulation applications (chapter 4), followed by the description of some experiments conducted on a model of token movements on randomly generated graphs, based on the presented framework (chapter 5). This is followed by an application of method and framework in the simulation of tram schedules. Background and architecture of project CATS are described, the applied simulation and optimization models presented. These modules are utilized to generate and evaluate tram schedules for the networks of the cities of Cologne and Montpellier. This is followed by some observations on the run time behavior of the simulation module (chapter 6). The thesis concludes with a short summary and some thoughts on further research (chapter 7).

Kurzzusammenfassung

Im Rahmen des *Projekts Computer Aided Tram Scheduling (CATS)* werden Methoden und Werkzeuge zum Generieren, Simulieren und Bewerten von Stadtbahnfahrplänen entwickelt, die von den Verkehrsplanern vor Ort auf Desktop-PCs oder Notebooks genutzt werden können. Das wissenschaftliche Interesse liegt dabei in der Erforschung und Entwicklung der benötigten Konzepte und Verfahren, diese liegen hauptsächlich in den Bereichen mathematische Optimierung und diskrete Simulation. In diesem Zusammenhang entstand der Wunsch nach einer parallelen Simulationsanwendung, die deutlich genauer und schneller als das bestehende sequentielle Modul arbeitet.

Im Rahmen dieser Arbeit werden Ziele aus drei Bereichen bearbeitet:

Zunächst soll ein Verfahren zur modellbasierten Parallelisierung diskreter Simulationsanwendungen entworfen werden. Dabei sollen eine Reihe von Eigenschaften genutzt werden, die u.a. viele Verkehrsmodelle aufweisen. Um die von der Zielplattform zur Verfügung gestellten Ressourcen auszunutzen, soll der Ansatz ein dynamisches und adaptives Lastausgleichsverfahren beinhalten. Das Verfahren soll als Framework zur Entwicklung von Simulationsanwendungen implementiert werden, sein dynamisches Verhalten soll anhand von Beispielanwendungen untersucht werden.

Weiterhin sollen die im Rahmen des Projekts CATS erstellten Werkzeuge vorgestellt werden. Im Vordergrund soll dabei das auf das beschriebene Framework aufsetzende, parallele Simulationsmodul stehen, mit dessen Hilfe die dynamischen Eigenschaften von Stadtbahnfahrplänen untersucht werden können. Um hierzu Kandidaten zu haben, soll eine Optimierungsanwendung genutzt werden, die Fahrpläne hinsichtlich ihrer Robustheit optimiert und gleichzeitig planerische Ansprüche an einsatztaugliche Fahrpläne berücksichtigt. Um die Nutzung mit verschiedenen Stadtbahnnetzen zu ermöglichen, sollen die Anwendungen einen von vielen Verkehrsunternehmen genutzten Datenstandard verwenden.

Zuletzt sollen die erstellten Werkzeuge noch praktisch erprobt werden. Hierzu werden die Stadtbahnnetze der Städte Köln und Montpellier modelliert, und sowohl robuste als auch bewusst nicht robuste Fahrpläne für sie generiert. Diese Fahrpläne werden gemeinsam mit den von den Verkehrsunternehmen real verwendeten Fahrplänen simuliert und

die dabei resultierenden Verspätungsdaten verglichen.

Die Arbeit beginnt mit einer Einführung in Kontext, Motivation und Ziele (Kapitel 1), gefolgt von einigen Hintergründen zu Verfahren der parallelen Berechnung von diskreten Simulationsmodellen. Dabei werden eine Reihe von allgemeinen und anwendungsbezogenen Parallelisierungsverfahren betrachtet (Kapitel 2). Danach wird ein Verfahren entwickelt, das insbesondere für die parallele Simulation von diskreten Verkehrsmodellen geeignet ist. Zuerst wird die Idee des Verfahrens beschrieben und auf die Umsetzung eingegangen, dann werden einige Überlegungen zu Skalierbarkeit und Effizienz des Verfahrens angestellt (Kapitel 3). Darauf folgt die Beschreibung einer Implementierung des Verfahrens als Framework für Simulationsanwendungen (Kapitel 4). Die Überlegungen zum Verhalten des Verfahrens werden nun ergänzt um eine Reihe von Beobachtungen, die bei Experimenten mit der Simulation von Tokenbewegungen auf zufällig erzeugten Graphen gemacht werden (Kapitel 5). Danach folgt die eigentliche Anwendung des Verfahrens: Hintergründe und Architektur des Projekts CATS werden erläutert, dazu die eingesetzten Optimierungs- und Simulationsmodelle beschrieben. Das Optimierungsmodul und die auf das vorgestellte Framework aufsetzende Simulationsanwendung werden genutzt, um Fahrpläne für die Stadtbahnnetze von Köln und Montpellier zu generieren und auszuwerten. Darauf folgen einige Beobachtungen zum Laufzeitverhalten der Simulationsanwendung (Kapitel 6). Die Arbeit endet mit einer kurzen Zusammenfassung des Erreichten und einem Ausblick auf weitere Forschungsmöglichkeiten (Kapitel 7).

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen und Symbole

α	Gewicht oder Glättungsfaktor
a_i	Last eines Prozessors p_i vor dem Lastausgleich
$a_{opt}(p)$	Optimale Last eines Prozessors p
$a_t(p)$	Last eines Prozessors p zum Zeitpunkt t
$\bar{a}$	Durchschnittslast aller Prozessoren
b_i	Last eines Prozessors p_i nach dem Lastausgleich
B_p	(Teil-)Baum mit Wurzel p
β_i, β_{min}, β_{max}	Schwellwert β_i für die zulässige Synchronisierungszeit, der dynamisch zwischen β_{min} und β_{max} schwanken kann
$CATS$	Computer Aided Tram Scheduling
d	Trödelfaktor
δ	Anzahl zu verlagernder Knoten
$e \in E$	Kante zwischen zwei Knoten im Modellgraphen $G_M(V, E)$
F_i	Last einer Spalte i
$f \in F$	Fahrt $f = (l, t_0)$ aus der Menge der Fahrten F
$f_p(i)$	Freie Kapazität eines Prozessors p zum Simulationsschritt i
FEL	Future Event List
Δt	Zeitabschnitt
ΔT	Fortschritt der Simulationszeit zwischen zwei Messungen
γ	Maximale Veränderung des Schwellwerts β_i
$g \in G$	Streckenabschnitt, gerichtete Verbindung zwischen zwei Knoten eines Stadtbahnnetzes
$G_R(P, Q)$	Graph mit der Knotenmenge P der Prozessoren und der Kantenmenge Q der Netzwerkverbindungen zwischen diesen Prozessoren
$G_T(P, U)$	Graph mit der Knotenmenge P der den Prozessoren eineindeutig zugeordneten Teilmodellen m_i und der Kantenmenge U der Kommunikationskanäle zwischen diesen Teilmodellen

$G_M(V, E)$	Graph mit der Knotenmenge V der Modellknoten und der Kantenmenge E der Kommunikationsverbindungen zwischen diesen Knoten
GVT_t	Globale virtuelle Zeit *(Global Virtual Time)* zum Uhrzeitpunkt t
$h \in H$	Haltepunkt $h \in H$
$h' \in H'$	Haltepunkttyp $h' \in H'$
IVT_i	Integrated Virtual Time, Messung i
k	Anzahl $k = \|P\|$der Prozessoren in P
KVB	Kölner Verkehrsbetriebe AG
λ	Fahrplan, entweder als Menge aller Umläufe U (aus Sicht des Anbieters) oder als Menge der Abfahrtszeiten der Linien L (aus Sicht der Fahrgäste)
$l \in L$	Eine Linie l aus der Menge der Linien L
$l_p(i)$	Grad der Auslastung eines Prozessors p zum Simulationsschritt i
L_i	Lookahead eines Prozessors p_i
LAN	Local Area Network
$L(v)$	Durch einen Modellknoten v erzeugte Rechenlast
m_i	Teilmodell des Simulationsmodell, zugeordnet einem Prozessor $p_i \in P$
N	Nachricht oder Simulationsereignis mit Zeitstempel t
N_A	Anti-Nachricht
N_v	Als Nachricht kodierter Modellknoten $v \in V$
$p \in P$	Prozessor p aus der Menge aller Prozessoren $P = \{p_1, \ldots, p_k\}$
$p(v)$	Prozessor, der den Modellknoten $v \in V$ verwaltet
p_d	Trödelwahrscheinlichkeit
p_f	Gemäß des Simulationsfortschritts langsamster Prozessor
p_s	Gemäß des Simulationsfortschritts schnellster Prozessor
PEL	Processed Event List
φ_p	Anteil der von einem Prozessor p verwalteten Modellknoten, die migriert werden sollen
q	Maximal zulässige Lastungleichheit
$r \in R$	Planerische Bedingung r aus der Menge aller planerischen Bedingungen R
$s \in S$	Station $s = \{h_{1,}..., h_n\}$ als Menge von Haltepunkten aus der Menge der Stationen S
s_i	Geglättete Synchronisierungszeit im Simulationsschritt i
t	Zeitpunkt

t_0	Startzeitpunkt
t_{busy}	Gesamtberechnungszeit aller Prozessoren in einem Simulationsschritt
$t_c(p,i)$	Kommunikationszeit eines Prozessors p in Schritt i
t_{cycle}	Zykluszeit einer Lichtsignalanlage, $t_{cycle} = t_{red} + t_{green}$
$t_g(i)$	Gesamtdauer eines Simulationsschritts i
t_{green}	Dauer der Grünphase einer Lichtsignalanlage, $t_{green} = t_{cycle} - t_{red}$
t_{idle}	Gesamtsynchronisierungszeit aller Prozessoren in einem Simulationsschritt
t_{init}	Zeitpunkt des ersten Phasenwechsels von rot auf grün einer Lichtsignalanlage
t_L	Aktuelle Simulationszeit plus globalem Lookahead
$t_m(p,i)$	Berechnungszeit eines Prozessors p in Schritt i
t_M	Für die Migration von Modellbestandteilen geschätzte Zeit
t_{red}	Dauer der Rotphase einer Lichtsignalanlage, $t_{red} = t_{cycle} - t_{green}$
$t_s(p,i)$	Synchronisierungszeit eines Prozessors p in Schritt i
$u \in U$	Umlauf $u = \{f_1,...,f_n\}$ aus der Menge aller Umläufe U
$v \in V$	Modellknoten v aus der Menge aller Modellknoten V im Modellgraphen $G_M(V,E)$
VDV	Verband Deutscher Verkehrsunternehmen e.V.
V_p	Teilmenge $V_p \subseteq V$ der Modellknoten, die einem Prozessor p zugeordnet werden
VPT_i	Virtual Time Progress, Messung i
$w \in W$	Weiche w aus der Menge der Weichen W, verbindet drei Streckenabschnitte
WAN	Wide Area Network
$w(p)$, $w(P)$	Leistungskoeffizient eines Prozessors p oder einer Prozessormenge P
x_i	Optimal zu verschiebende Last von Prozessor i
y	Simulationsereignis mit Zeitstempel *time(y)*
z	Speedup

1. Einführung

1.1. Kontext und Motivation

Die vorliegende Dissertation bildet den abschließenden Baustein des am Lehrstuhl von Prof. Dr. Ewald Speckenmeyer am Institut für Informatik der Universität zu Köln beheimateten Projekts *Computer Aided Tram Scheduling (CATS,* für eine Zusammenfassung siehe Abschnitt 6.1 oder Ullrich, et al. in [74]). Im Rahmen dieses Projekts werden Methoden und Werkzeuge zum Generieren, Simulieren und Bewerten von Stadtbahnfahrplänen entworfen und entwickelt, die Verkehrsplanern beim Erstellen robuster Fahrpläne helfen sollen. Als Robustheit wird in diesem Kontext der Grad bezeichnet, zu dem auftretende kleinere Verspätungen auf einzelne Fahrzeuge beschränkt bleiben und sich nicht über nachfolgende Fahrzeuge durch das Netz ausbreiten. Die Werkzeuge sollen insbesondere im Rahmen des Stadtbahnnetzes der Kölner Verkehrsbetriebe AG (KVB) angewendet werden.

Die zum Projekt gehörende Software umfasst eine Reihe von Modulen:

- Eine Datenbank, die Abbildungen von Stadtbahnnetzen und Fahrplänen enthält.
- Ein Optimierungsmodul, das hinsichtlich Robustheit optimiert und zugleich die Vorgaben der Verkehrsplaner an einen validen Fahrplan berücksichtigt.
- Ein Simulationsmodul, um generierte und real verwendete Fahrpläne hinsichtlich ihrer dynamischen Eigenschaften (wie die Bewegung der Fahrzeuge während des Betriebstags, die entstehenden Verspätungen an Haltepunkten und das Verhalten bei kleineren Störungen) zu bewerten und zu vergleichen.
- Ein Visualisierungswerkzeug zum Auswerten der Simulations- und Optimierungsergebnisse.

Zielplattformen für diese Software sind handelsübliche Desktop-PCs oder Notebooks, so dass die Werkzeuge von Projektmitarbeitern und Planern vor Ort und ohne Zugang zu Großrechnern oder anderen speziellen Maschinen genutzt werden können.

Das wissenschaftliche Interesse liegt dabei in der Erforschung und Entwicklung der benötigten Konzepte und Methoden, diese liegen hauptsächlich in den Bereichen mathematische Optimierung und diskrete Simulation. Neben einer ganzen Reihe von Diplomarbeiten entstanden im Rahmen des Projekts bislang zwei Dissertationen (siehe Genç in [20] und Lückemeyer in [38]), einzelne Forschungsergebnisse wurden bereits in einer Reihe von Artikeln und Vorträgen vorgestellt (siehe Lückemeyer und Speckenmeyer in [39], Lückerath, Ullrich und Speckenmeyer in [41], Speckenmeyer, et al. in [62], Ullrich, et al. in [74] und [78], und Ullrich, Lückerath und Speckenmeyer in [75], [76] und [77]).

Die Architektur des Projekts, die Datenbank und das Optimierungsmodul werden in den Abschnitten 6.1 und 6.2 zusammenfassend vorgestellt, das Hauptaugenmerk dieser Arbeit liegt auf dem Bereich der Simulation. Hier soll das bisherige, sequentielle Simulationsmodul durch ein deutlich genaueres und auch schnelleres Modul ersetzt werden. Die Architektur des Moduls soll so ausgelegt sein, dass es später um ein Modell des mit dem Stadtbahnfahrplan synchronisierten Busverkehrs erweitert werden kann. Zugleich soll das Modul auf den Übergang zur Online-Simulation vorbereitet sein: Dabei wird zu einem bestimmten Stichzeitpunkt ein Schnappschuss des realen Systemzustands in eine Datenbank übernommen und von diesem aus der Betriebstag weiter simuliert, um so z.B. bei konkreten größeren Störungen mögliche Lösungsstrategien vor ihrem Einsatz im realen Stadtbahnnetz zu prüfen. Insbesondere ungeeignete Strategien können mitunter durch einen einzelnen Lauf ausgeschlossen werden.

Moderne Arbeitsplatz- und Mobilrechner verfügen über mehrere Prozessoren oder Prozessorkerne. Um eine möglichst hohe Ablaufgeschwindigkeit zu erreichen, soll das Simulationsmodul diese zur Verfügung stehenden Ressourcen ausnutzen, also parallel ablaufen. Dazu soll eine Parallelisierungsmethode entwickelt und als Framework implementiert werden, die einige bekannte Eigenschaften von Modellen zur Simulation von Stadtbahnfahrplänen ausnutzt. Die Methode soll sich jedoch nicht ausschließlich auf die Simulation von Stadtbahnfahrplänen beschränken, sondern auch für Modelle mit ähnlichen Eigenschaften nutzbar sein.

Vom wissenschaftlichen Standpunkt gesehen ist der Bereich der parallelen Simulation sehr interessant, da in vielen Modellen zwar ein großer Anteil von Parallelismus angelegt ist, dieser sich jedoch wegen der bestehenden Abhängigkeiten zwischen den Modellkomponenten oft nur schwierig ausnutzen lässt (siehe hierzu Fujimoto in [16], S. 19f.).

Zum Erreichen dieser Ziele sollen grundlegende Methoden aus dem Bereich der modellbasierten Parallelisierung (beschrieben in Abschnitt 2.2) verwendet werden: Diese skalieren gut, sie sind bekannt und erprobt (siehe hierzu und zu den folgenden Absätzen Fujimoto in [17], S. 4ff.). Allgemeine modellbasierte Methoden, die für alle Arten von

diskreten Modellen propagiert werden, können so erweitert und adaptiert werden, dass sie sich besonders gut für die Verwendung mit Klassen von Modellen mit bestimmten Eigenschaften eignen.

Für die Verwendung mit parallelen Simulationen sind spezielle Lastausgleichsverfahren nötig, da hier nicht nur eine möglichst hohe Auslastung der Prozessoren erreicht werden soll, sondern zugleich auf ein möglichst gleichmäßiges Voranschreiten in der Simulationszeit geachtet werden muss. Dies ist nötig, da wegen der typischen Abhängigkeiten im Modell der Gesamtfortschritt der Simulation von dem Prozessor oder Rechner abhängig ist, der am langsamsten in der Simulationszeit fortschreitet (siehe hierzu auch Abschnitt 2.2).

Um die verfügbaren Ressourcen effizient zu nutzen, sollte das verwendete Lastausgleichsverfahren dynamisch und adaptiv sein: Im Gegensatz zu einem rein statisch arbeitenden Lastausgleich, also einer vor dem Simulationslauf ausgeführten Partitionierung der Last, beachtet ein dynamisches Verfahren die Lastimbalancen, die sich aus der im Laufe der Simulation verändernden Aktivität in den einzelnen Modellbereichen ergeben. Das Verfahren misst und bewertet die Last der einzelnen Prozessoren und gleicht sie ggf. durch die Verlagerung einzelner Modellbestandteile aus. Die durch die Veränderung der Modellaktivität entstehenden Imbalancen werden auch als innere Störungen bezeichnet (siehe Schlagenhaft in [57]).

Ein adaptives Verfahren beachtet während des Laufs Schwankungen der durch die einzelnen Prozessoren zur Verfügung gestellten Rechenleistung. Diese entstehen, da sich die Simulation die vorhandenen Ressourcen typischerweise mit dynamisch entstehenden, externen Prozessen teilen muss. Läuft die Simulation in einem inhomogenen Rechnernetz ab, passt sich ein adaptives Verfahren zudem durch Lastverlagerung den unterschiedlich schnellen Einzelprozessoren an. Die Veränderungen der zur Verfügung stehenden Rechenleistung durch nicht im Modell begründete Umstände werden auch als äußere Störungen bezeichnet (siehe nochmals Schlagenhaft in [57]).

Entscheidend für die Effektivität eines dynamischen und adaptiven Verfahrens sind die Messung und die Bewertung der Last: Gute Verfahren müssen hier ein Lastmaß verwenden, das sowohl die zur Verfügung stehende Rechenleistung als auch den Fortschritt der Simulation einbezieht.

Das Vorhaben lässt sich wie folgt zusammenfassen: Diese Arbeit beschäftigt sich mit dem Entwurf und der Entwicklung einer Methode zur modellbasierten Parallelisierung der Simulation von Stadtbahnmodellen, der Beschreibung der für das Projekt CATS entwickelten Projekt-Software, der Ergänzung und Erweiterung der Datenbasis zum Kölner KVB-Stadtbahnnetz und zuletzt mit der Auswertungen dieser Daten.

Mit der Fertigstellung dieser Dissertation sind die grundlegenden Arbeiten am Projekt CATS abgeschlossen, die Werkzeuge können in den Einsatz überführt werden. Das Nachfolgeprojekt INSTEP beschäftigt sich mit der Erweiterung der bestehenden Werkzeuge in Richtung der Online-Simulation multi-modaler Verkehrssysteme (für erste Berichte hierzu siehe Lückerath, Ullrich und Speckenmeyer in [42] und Ullrich, et al. in [78]).

1.2. Ziele der Arbeit

Die im Rahmen des letzten Abschnitts vermittelten Ziele lassen sich in drei Bereiche aufgliedern:

1. Zunächst soll ein Verfahren zur modellbasierten Parallelisierung von Simulationsanwendungen entworfen werden. Ziel ist dabei nicht die Entwicklung eines allgemeinen Ansatzes, der für alle denkbaren Modelle gleich gut geeignet ist, sondern ein Verfahren, das auf spezifischen Eigenschaften einer Untergruppe von Modellen aufbaut. Insbesondere soll es die Eigenschaften von Modellen zur Simulation von Stadtbahnfahrplänen ausnutzen. Wie beschrieben soll das Verfahren ausgelegt sein für die Ausführung auf einem Notebook oder Desktop-PC, dabei sollen die verfügbaren Ressourcen zur Parallelverarbeitung genutzt werden. Um diese Rechenressourcen für die Simulation effektiv zu nutzen, soll der Ansatz ein dynamisches und adaptives Lastausgleichsverfahren beinhalten.
   Das Verfahren soll als Framework zur Entwicklung für Simulationsanwendungen implementiert werden. Die Technik des Lastausgleichs und des Managements der Parallelität soll dabei gekapselt werden und aus Sicht der Anwendung transparent bleiben. So ist eine Änderung der zugrunde liegenden Methoden möglich, ohne die Weiterverwendbarkeit der auf dem Framework aufsetzenden Anwendungen zu gefährden. Das dynamische Verhalten des Frameworks soll anhand einer Beispielanwendung untersucht werden.

2. Die im Rahmen des Projekt CATS erstellten Software-Werkzeuge zur Optimierung und Simulation von Stadtbahnfahrplänen sollen beschrieben werden. Im Vordergrund steht dabei eine auf das zu entwickelnde Framework aufsetzende, parallele Simulationsanwendung. Mit Hilfe dieser Simulationsanwendung sollen dynamische Eigenschaften von Fahrplänen untersucht werden können. Um hierzu Kandidaten zu haben, soll eine Optimierungsanwendung genutzt werden, die Fahrpläne hinsichtlich ihrer Robustheit optimiert und dabei gleichzeitig planerische Ansprüche an einsatztaugliche Fahrpläne berücksichtigt. Um die Nutzung mit verschiedenen

Stadtbahnnetzen zu ermöglichen, sollen die Anwendungen einen von vielen Verkehrsunternehmen genutzten Datenstandard verwenden.

3. Abschließend sollen die vorliegenden Daten zum Kölner Stadtbahnnetz ausgewertet, und außerdem ein vom Kölner Netz möglichst verschiedenes, zweites Stadtbahnnetz betrachtet werden. Zur Untersuchung des Kölner Netzes müssen bereits vorhandene Daten zum historischen Stadtbahnnetz des Jahres 2001 überprüft und ergänzt werden. Daneben soll auch das zum Zeitpunkt der Niederschrift aktuelle Netz betrachtet werden, dazu der im Bau befindliche Nord-Süd-Tunnel unter der Kölner Innenstadt. Hierzu müssen Repräsentationen dieser Versionen des Kölner Netzes in der Datenbank angelegt werden, das zweite Netz muss ebenfalls erfasst werden. Um für diese Netze eine Reihe von geeigneten Fahrplänen zu erstellen, wird die beschriebene Optimierungsanwendung eingesetzt. Optimale und zufällig generierte Fahrpläne sollen dann mit Hilfe der Simulationsanwendung gegenüber gestellt werden, u.a. um die Eignung des Kriteriums Robustheit zur Reduzierung von Verspätungen zu zeigen. Soweit möglich sollen dann noch einige Aussagen zu den Auswirkungen der bereits durchgeführten und noch anstehenden Veränderungen im Kölner Netz gemacht werden. Neben dem Gewinnen von Erkenntnissen über die untersuchten Stadtbahnnetze und die Eigenschaften von Fahrplänen wird hiermit zusätzlich die Anwendbarkeit der beschriebenen Werkzeuge demonstriert.

1.3. Vorgehen

Nach dieser Einführung wird die Arbeit in Kapitel 2 mit der Beschreibung verschiedener, aus der Literatur bekannter Verfahren zur Parallelisierung von diskreten Simulationsanwendungen fortgesetzt. Hierbei wird insbesondere unterschieden zwischen allgemeinen Verfahren, die für alle diskreten Modelle geeignet sind und deren Erweiterungen und Anpassungen, die auf einzelne Modelle zugeschnitten sind.

In Kapitel 3 wird ein Ansatz zur modellbasierten Parallelisierung von Simulationsanwendungen entwickelt, der bestimmte Eigenschaften von Modellen von Stadtbahnnetzen nutzt. Die dem Verfahren zugrunde liegenden Ideen werden anhand eines Beispiels präsentiert, dabei wird aus Gründen der Übersichtlichkeit auf eine einfache Simulation von Token-Bewegungen in randomisierten Graphen zurück gegriffen. Dazu werden einige Überlegungen zur Skalierbarkeit und Effizienz des Verfahrens angestellt.

Im darauf folgenden Kapitel 4 wird ein auf dem Message Passing Interface (MPI) aufsetzendes, als C++-Bibliothek realisiertes Framework vorgestellt, das den entworfenen Ansatz implementiert. Dabei wird zuerst auf das vom Framework vorgenommene Ma-

nagement der parallelen Simulationsmodelle eingegangen, und dann die Schnittstellen beschrieben, die der Simulationsanwendung zur Verfügung gestellt werden.

Um die dynamischen Eigenschaften des Verfahrens zu untersuchen, wird in Kapitel 5 eine einfache Testanwendung zur bereits beschriebenen Simulation von Tokenbewegungen in zufällig erzeugten Graphen entwickelt und eingesetzt. Anhand dieser Anwendung werden eine Reihe von Experimenten zu Laufzeitverhalten und Skalierbarkeit durchgeführt.

In Kapitel 6 wird auf die Anwendung des Frameworks im Rahmen des Projekts CATS eingegangen. Dazu wird nach der Erläuterung einiger Hintergründe zu Stadtbahnnetzen und robusten Stadtbahnfahrplänen eine Anwendung zur exakten Optimierung von getakteten Stadtbahnfahrplänen beschrieben. Darauf folgt dann eine Beschreibung des Entwurfs und der Implementierung der auf das Framework aufsetzenden Simulationsanwendung, mit deren Hilfe verschiedene (sowohl algorithmisch erzeugte als auch real verwendete) Fahrpläne für die Stadtbahnnetze von Köln und Montpellier untersucht werden. Dabei werden die aktuellen Stadtbahnnetze beider Städte, das historische Kölner Netz aus dem Jahr 2001 und das für das Jahr 2020 geplante Kölner Netz inklusive des im Bau befindlichen Nord-Süd-Tunnels betrachtet. Auch hier wird das Laufzeitverhalten der erstellten Anwendung betrachtet.

Die Arbeit schließt in Kapitel 7 mit einer Zusammenfassung des Erreichten und einem Ausblick auf das mögliche weitere Vorgehen.

2. Parallele Simulationsverfahren

2.1. Einführung und Abgrenzung

Im Rahmen einer Simulationsstudie wird ein reales oder geplantes System als Modell abgebildet (siehe hierzu und zu den folgenden Absätzen auch Banks, et al. in [5], S. 21ff.). Das Verhalten des Modells wird mithilfe verschiedener Parameter und Eingabedaten untersucht, so erhofft man sich Rückschlüsse auf das Verhalten des realen Systems unter den entsprechenden Bedingungen.

Der Fokus dieser Arbeit richtet sich auf den Bereich der analytischen Simulation, die komplett auf Software-Ebene abläuft. Die beschriebenen Verfahren können jedoch grundsätzlich auch in *hardware-in-the-loop-* oder *human-in-the-loop*-Anwendungen (z.B. in der Simulation virtueller Welten) benutzt werden.

Simulationsmodelle bestehen aus einer Reihe von Entitäten, die physische oder logische Bestandteile des zu simulierenden Systems abbilden, inklusive ihres Verhaltens und ihrer Beziehungen untereinander. Diese Entitäten lassen sich unterteilen in residente Entitäten (kurz auch Residenten), die über den kompletten Simulationszeitraum hinweg existieren und sich im Modell nicht bewegen, und transiente Entitäten (kurz auch Transienten), die im Laufe der Simulationszeit entstehen, durch das Modell wandern und vergehen können. Bei der Simulation eines Produktionsprozesses könnten Lager, Werkbänke und deren Verbindungsbänder als residente Entitäten modelliert werden, die zu verarbeitenden Werkstücke und Zwischenprodukte als transiente Entitäten. In der Literatur wird anstelle des Begriffs Modellentitäten auch von logischen Prozessen gesprochen (engl. *logical process*, vgl. Fujimoto in [17], S. 51).

Die in dieser Arbeit untersuchten Verfahren und Modelle gehören zum Bereich der diskreten Simulation. In Abgrenzung zur kontinuierlichen Simulation (siehe hierzu auch Banks, et al. in [5], S. 32), wie sie z.B. im Bereich der Materialwissenschaften häufig Anwendung findet, ändert bei einer diskreten Simulation das Modell seinen Zustand in diskreten Zeitpunkten der Simulationszeit. Als Simulationszeit (auch Modellzeit, engl. *simulation time* oder *model time*) wird die simulierte Zeit bezeichnet, die während der Berechnung des Modells aus Sicht der simulierten Entitäten vergeht (siehe hierzu und

zu den nächsten Absätzen Fujimoto in [17], S. 27ff.). Hier kann je nach Kontext ein simulierter Zeitpunkt oder eine simulierte Zeitspanne gemeint sein. Diese Simulationszeit ist abzugrenzen von der während der Durchführung des Simulationslaufs vergehenden Uhrzeit (in der Literatur auch engl. *wallclock time* genannt).

Im Rahmen eines diskreten Simulationsmodells hat die Simulationszeit eine zeitliche Auflösung, diese entspricht dem minimalem Zeitfortschritt. In einem Produktionsmodell könnte die Auflösung z.B. bei einer Sekunde liegen, bei anderen Modellen bei 50 Millisekunden oder einer Stunde. In vielen Fällen werden Simulationsmodelle so schnell wie möglich berechnet (engl. *as fast as possible execution*). In bestimmten Anwendungen ist jedoch das Binden der Simulationszeit an die Uhrzeit wünschenswert. Hier spricht man dann von Echtzeitausführung (engl. *real time execution*) oder skalierter Echtzeitausführung (engl. *scaled real time execution*).

Der Zeitfortschritt kann in fixen oder variablen Schritten erfolgen (siehe hierzu und zu den folgenden Absätzen Fujimoto in [17], S. 30ff.).

Bei Modellen mit fixem Zeitfortschritt (engl. *fixed time step*) schreitet die Simulationszeit in gleichen Zeitinkrementen voran. Hier gleicht die Berechnung einer Schleife über den Simulationszeitraum mit fester Schrittweite Δt, die Modellentitäten können ihren Zustand zu jedem dieser Zeitpunkte ändern. Die Kommunikation der Entitäten untereinander erfolgt über direkte oder in der Zukunft terminierte Nachrichten. Diese Verfahren haben den Vorteil, dass eine Kopplung an die Uhrzeit, also die Ausführung in (skalierter) Echtzeit, sehr einfach möglich ist.

Bei Modellen mit variablem Zeitfortschritt (engl. *variable time step*) wird der Fortschritt der Simulationszeit durch die Verarbeitung von Simulationsereignissen bewirkt, die zu Änderungen im Modellzustand führen. In einem Modell aus dem Bereich der Produktionssimulation könnte so ein Ereignis den Beginn der Bearbeitung eines Werkstücks an einer Werkbank abbilden, das darauf folgende Ereignis das Ende der Bearbeitung, das nächste das Überführen des Werkstücks auf ein Transportband. Der zentralen Bedeutung dieser Ereignisse folgend, bezeichnet man die Berechnung von Modellen mit variablem Zeitfortschritt daher auch als ereignisbasierte Simulation. Jedes dieser Ereignisse besitzt einen Zeitstempel (engl. *time stamp*), der den Zeitpunkt seines „Eintretens" markiert. Weiterhin verfügt es über einen Wert, der den Typ des Ereignisses beschreibt und i.d.R. über verschiedene weitere Daten wie eine Liste der vorgesehenen Empfänger und die Kennung der sendenden Entität.

Die Ereignisse werden in einer Prioritätswarteschlange verwaltet, dort liegen sie aufsteigend sortiert gemäß ihrer Ereigniszeit. Zur Berechnung der Simulation wird das Ereignis mit dem geringstem Zeitstempel entnommen, der Wert der aktuellen Simulationszeit

auf die Ereigniszeit gesetzt und das Ereignis verarbeitet. Dabei ändert sich i.d.R. der Modellzustand, indem die entsprechenden Zustandsvariablen verändert werden. Bei der Verarbeitung können neue Ereignisse entstehen, die dann in die Prioritätswarteschlange eingefügt werden. Diese Liste zu verarbeitender Ereignisse wird in der Literatur auch *Future Event List (FEL)* genannt. Ihre Implementierung entscheidet überwiegend über die Effizienz einer solchen ereignisbasierten Simulationsanwendung. Für eine nähere Betrachtung einer großen Zahl von geeigneten Datenstrukturen siehe Lückemeyer in [38], S. 40ff.

Beide Modellarten sind ineinander überführbar: Durch Einfügen von Null-Ereignissen zu jedem möglichen Simulationszeitpunkt lässt sich in einem Mechanismus, der für variablen Fortschritt ausgelegt ist, ein fixer Zeitfortschritt simulieren; zentraler Ereignistyp ist dann der Zeitfortschritt. Bei hinreichend hoher Ereignisdichte wird diese fixe Schrittweite gerade in großen Modellen von allein erreicht. Im Rahmen einer mit fixem Inkrement fortschreitenden Schleife lässt sich variabler Fortschritt simulieren, indem jede Modellentität jeweils nur angesprochen wird, wenn die globale Simulationszeit gleich einem von dieser Entität kommunizierten Wert ist - dieser entspricht dann dem Zeitstempel des nächsten Ereignisses.

Die im Folgenden beschriebenen Methoden sind von der Art des Zeitfortschritts weitgehend unabhängig, werden aber oft als Verfahren zur ereignisbasierten Simulation propagiert. Auch Modelle aus anderen Bereichen, wie z.B. der prozessbasierten oder agentenbasierten Modellierung, lassen sich auf ereignisbasierte Modelle, und damit auch auf Modelle mit fixem Zeitinkrement, zurück führen.

Um den Ablauf einer Simulation zu beschleunigen, lässt sich die Abarbeitung auf parallele Prozesse verteilen. Übliche Ziele sind dabei eine möglichst schnelle Ausführung oder das Berechnen in (skalierter) Echtzeit. Im Folgenden wird von *As fast as possible*-Ausführung als Ziel ausgegangen. Ein für eine gewünschte Echtzeitbindung zu schneller Ablauf lässt sich ohne Probleme auf die Wunschgeschwindigkeit abbremsen. Zentral ist dabei, dass eine parallel ausgeführte Simulation identische Ergebnisse liefern muss wie die sequentielle Ausführung des gleichen Modells. Die Technik der Simulation darf also nicht das Modellverhalten beeinflussen (siehe auch Fujimoto in [17], S. 52ff.), man spricht hier auch von der Vermeidung von Simulationsartefakten.

Die zentrale Messgröße für die Leistungsfähigkeit eines Parallelisierungsverfahrens ist der Speedup. Dieser bestimmt das Verhältnis der Laufzeit bei sequentieller Ausführung zur Laufzeit bei paralleler Ausführung (siehe Formel 2.1).

$$speedup = \frac{t_{sequentiell}}{t_{parallel}} \tag{2.1}$$

Ziel der Parallelisierungsverfahren ist natürlicherweise das Erreichen möglichst hoher Speedup-Werte bei einer gleichbleibenden Zahl eingesetzter Prozessoren. Zu berücksichtigen ist dabei, dass auch eine parallel rechnende Anwendung zum Teil aus sequentiell zu verrichtenden Arbeiten besteht. Typischerweise sind dies Aufgaben wie das Einlesen des Modells vor Simulationsbeginn, die Ausgabe der Ergebnisse nach Beendigung der Läufe und andere Tätigkeiten, die von einem einzelnen Prozessor ausgeführt werden müssen. Als paralleler Anteil wird der Anteil der Berechnungsarbeit bezeichnet, der wirklich parallel ausgeführt werden kann, z.B. die eigentliche Berechnung eines Simulationslaufs. Da die sequentiellen Anteile von der Art der Parallelisierung kaum berührt werden, liegt der Schwerpunkt hier auf der Betrachtung der parallelen Anteile. Die obere Grenze des zu erreichenden Speedups wird durch Amdahls Gesetz (formuliert von Amdahl in [1]) beschrieben: Bei einem sequentiellen Anteil von $1/z$ am gesamten Berechnungsaufwand beträgt der maximal erreichbare Speedup z.

Im Bereich der diskreten Simulation sind eine Reihe möglicher Herangehensweisen zur Parallelisierung bekannt:

Das *Ausführen paralleler Experimente* erscheint dabei als die einfachste Vorgehensweise: Die Eingangsdaten für die Simulationsanwendungen werden auf den beteiligten Rechnern abgelegt, dann werden die einzelnen Simulationsläufe zeitgleich, jeweils sequentiell ausgeführt. Nach Beendigung der Läufe werden dann lediglich noch die Ergebnisdaten eingesammelt und aufbereitet. Die Gesamtlaufzeit ist also von der längsten Einzellaufzeit abhängig. Durch ein zusätzliches Protokoll können jedoch bei Abschluss einiger Läufe bereits Zwischenergebnisse zusammen gefasst werden.

Da bei diesem Verfahren die Ausführung eines einzelnen Simulationslaufs im Vergleich zur sequentiellen Ausführung nicht beschleunigt wird, kommt es für Modelle mit (skalierten) Echtzeitanforderungen nicht in Frage. Auch in anderen Bereichen, die die besonders schnelle Berechnung einzelner Läufe erfordern (um z.B. bei Online-Simulationen ungeeignete Lösungsstrategien ablehnen zu können), ist das parallele Ausführen von Experimenten nicht geeignet. Weiterhin muss das Modell klein genug sein, um vollständig in den Speicher der einzelnen Rechner zu passen. Als letzter Punkt sollte nicht verschwiegen werden, dass das gleichzeitige Ausführen mehrerer Experimente als Methode der Parallelisierung etwas plump erscheint.

Ist jedoch eine hohe Zahl von Läufen mit relativ kleinen Modellen gefragt, die ohne weitere Anforderungen schnell berechnet werden sollen, ist das gleichzeitige Ausführen eine einfache und effektive Vorgehensweise. Ein weiterer Vorteil ist die häufig vorhandene Kompatibilität zu bereits bestehender, sequentieller Simulationssoftware. So schlagen Merz und Bröcker die Methode in [48] für die Anwendung mit dem kommerziellen Simu-

lationswerkzeug Dymola vor.

Eine weitere Möglichkeit, sich vorhandene Ressourcen zur Parallelberechnung zunutze zu machen, ist das *Auslagern von Hilfsfunktionen.* Hier wird die Ereignisverarbeitung von einem einzelnen Prozessor zentral kontrolliert, einige aufwändige Unterfunktionen werden jedoch auf andere Rechenknoten ausgelagert. Hierfür kommen Funktionen wie die Erzeugung großer Mengen von Zufallszahlen und andere zeitkomplexe Berechnungen, oder auch der Zugriff auf Datenbanken und Ein-/Ausgabegeräte in Frage. Die Methode kann den Ablauf beschleunigen, ist aber vergleichsweise aufwändig umzusetzen und skaliert naturgemäß nicht gut.

Ein Spezialfall des Auslagerns von Hilfsfunktionen ist die *Parallelisierung des Verwaltens von Datenstrukturen*: Bei vielen ereignisbasierten Simulationsanwendungen wird die zentrale Prioritätswarteschlange sehr groß, so dass das Einfügen, Suchen und Entfernen von Simulationsereignissen einen erheblichen Teil der Laufzeit in Anspruch nimmt (siehe hierzu nochmals die Untersuchungen von Lückemeyer in [38], S. 40ff.). Durch Ausnutzen paralleler Ressourcen kann hier Zeit gespart werden, entweder durch paralleles Bearbeiten einer zentralen Liste oder durch geeignetes Verwalten mehrerer Teillisten. Wichtig ist dabei, dass das Modellverhalten durch die Art der Verwaltung nicht beeinflusst werden darf: Aus Sicht des Modells handelt es sich um eine einzelne, zentrale Ereignisliste. Eine vergleichende Übersicht über eine Reihe von Verfahren findet sich bei Rönngren und Ayani in [54].

Bei der *parallelen Ereignisverarbeitung* werden die Simulationsereignisse weiterhin in einer zentralen Ereignisliste verwaltet. Die Bearbeitung von Ereignissen mit dem gleichen Zeitstempel kann jedoch auf mehreren Prozessoren parallel erfolgen. Durch geeignete Protokolle muss dabei sicher gestellt werden, dass die Ergebnisse der Verarbeitung konsistent bleiben.

Bei der *zeitbasierten Parallelisierung* (siehe hierzu Fujimoto in [18], S. 152f.) wird der zu simulierende Zeitraum in Intervalle unterteilt, die dann den beteiligten Prozessoren zur Berechnung zugewiesen werden. Hierzu muss jedoch sicher gestellt werden, dass der Zustand zum Ende des simulierten Intervalls $[t_{i-1}, t_i]$ dem Zustand zu Beginn des zeitlich nächsten Intervalls $[t_i, t_{i+1}]$ entspricht. Dazu ist nötig, dass ein Intervall i berechnet werden kann, ohne auf die Ergebnisse der Simulation der Intervalle $1, ..., i-1$ angewiesen zu sein.

Ein mögliches Vorgehen ist dabei, den Simulator zu Beginn der Berechnung eines Intervalls i dessen Anfangszustand schätzen zu lassen. Wenn später die Ergebnisse der Simulation von Intervall $i-1$ vorliegen, wird dessen Endzustand mit dem geschätzten Anfangszustand des Nachfolgeintervalls verglichen. Gegebenenfalls muss jetzt die Simulation

von Intervall i durch zusätzliche Berechnungen korrigiert werden, im Zweifel geschieht dies durch Neuberechnung.

Im schlechtesten Fall müssen außer dem ersten Intervall (dessen Eingangsbedingungen vor Beginn der Berechnungen bekannt sind), sämtliche Intervalle neu berechnet werden. Der Zeitbedarf entspricht dann dem einer sequentiellen Ausführung. Bei sehr spezifischen Modellen, bei denen eine präzise Schätzung der Zwischenzustände möglich ist, ist mit dieser Technik eine massiv parallele Berechnung und damit ein sehr hoher Speedup möglich. Einige (wenige) Anwendungen der zeitbasierten Parallelisierung sind bekannt, so etwa in der Simulation von Petri-Netzen (siehe Greenberg, et al. in [21]) oder von Cache-Speichern (siehe Heidelberger und Stone in [23]).

Bei der *modellbasierten Parallelisierung* wird die im Modell vorhandene Parallelität genutzt. Dazu wird das Modell in Teilmodelle dekomponiert, die dann den zur Verfügung stehenden Prozessoren zur Ausführung zugewiesen werden (siehe hierzu und zu den folgenden Absätzen auch Fujimoto in [17], S.39ff.) Die Kommunikation zwischen den Entitäten verschiedener Teilmodelle erfolgt mittels Nachrichten, die von den ausführenden Prozessoren über den gemeinsamen Cache oder das verbindende Netzwerk gesendet werden. Diese Nachrichten kapseln dabei Simulationsereignisse, die von dem Zielprozessor bearbeitet werden müssen.

Die Prozessoren p_1 bis p_k aus der Menge der beteiligten Prozessoren P sind jeweils für die Berechnung eines Teilmodells verantwortlich und können als Knoten eines Graphen angesehen werden. Prozessoren mit Teilmodellen, die im Laufe der Simulation Nachrichten austauschen können, sind dann durch gerichtete Kanten verbunden. Für das Verständnis ist wichtig, dass diese Kanten Verbindungen der Teilmodelle untereinander darstellen und nicht unbedingt identisch sind mit der Konfiguration des physischen Netzwerks, das die Prozessoren verbindet (dies wird in Abschnitt 3.1 ausführlicher betrachtet).

Das Ziel der modellbasierten Parallelisierung ist das Ausnutzen der im Modell vorhandenen Parallelität durch paralleles Ausführen von Ereignissen, die in verschiedenen Bereichen des Modells auftreten. Die Grundannahme ist dabei, dass diese Ereignisse oft unabhängig voneinander ausführbar sind, also nur relativ selten Kommunikation über die Grenzen der Teilmodelle hinweg notwendig ist. So wirken sich bei der Simulation eines Stadtbahnnetzes die Bremsmanöver eines Fahrzeug in einem Teil des Netzes nicht unmittelbar auf die Manöver eines Fahrzeugs in einem anderen Teil aus. Die beiden Fahrzeuge können also im Großteil der Fälle unabhängig voneinander simuliert werden. Treten jedoch Abhängigkeiten zwischen den Teilmodellen auf, kommen die in Abschnitt 2.2 beschriebenen Synchronisierungsmethoden zum Tragen.

Die modellbasierte Parallelisierung wird mitunter auch als raumbasierte Parallelisierung oder *space-parallel execution* bezeichnet. Sie ist die Art der Parallelisierung, die in realen Anwendungen meist verwendet wird. Sie skaliert i.d.R. besser als die bereits beschriebenen Methoden, der Grad der Parallelität der Ausführung ist dabei jedoch entscheidend von der im Modell angelegten Parallelität abhängig.

Zentral für die modellbasierte Parallelisierung ist die sorgfältige Synchronisierung der Ausführung der Teilmodelle. Insbesondere muss die sog. lokale Kausalitätsbedingung (engl. *local causality constraint*, vgl. Fujimoto in [17], S. 52ff.) eingehalten werden. Diese lokale Kausalitätsbedingung schreibt vor, dass jede Modellentität die sie betreffenden Simulationsereignisse gemäß einer nicht-absteigenden Sortierung nach der Ereigniszeit ausführen muss. Wird die lokale Kausalitätsbedingung nicht eingehalten, können Kausalitätsfehler entstehen, die die Ergebnisse der Simulation ungültig machen.

Sei angenommen, dass in einer Stadtbahnsimulation ein Prozessor p_1 einen Betriebstag bis zur Simulationszeit 12.30 Uhr simuliert habe, ein Prozessor p_2 jedoch erst bei 12.05 Uhr angelangt sei. Nun verlässt ein Fahrzeug den Modellbereich von p_2. Dieser Prozessor schickt eine Nachricht an p_1 und übergibt darin das Fahrzeug zur Weitersimulation ab 12.06 Uhr. Diese Nachricht stammt aus Sicht von p_1 24 Simulationsminuten aus der Vergangenheit, in diesem Simulationszeitabschnitt hat p_1 die von dem Fahrzeug zu belegenden Ressourcen bereits an andere Bahnen vergeben. Die übergebene Nachricht kann ohne weitere Maßnahmen nicht mehr sinnvoll verarbeitet werden, die Simulation muss mit einer Fehlermeldung beendet werden.

Es entsteht also ein Synchronisierungsproblem: Das Parallelisierungsverfahren muss mit einem Synchronisierungsmechanismus sicher stellen, dass alle Ereignisse nach Ereigniszeit geordnet berücksichtigt werden. Dies ist eine zentrale Voraussetzung dafür, dass die parallele Ausführung einer Simulation wie gefordert zu den selben Ergebnissen kommt wie eine sequentielle Ausführung.

Wichtig für die Bearbeitung des Synchronisierungsproblems ist der Begriff des Lookaheads: Befindet sich ein logischer Prozess oder ein Teilmodell in Simulationszeit t, dann garantiert ein Lookahead von L, dass vor dem Simulationszeitpunkt $t + L$ keine weiteren Simulationsereignisse generiert werden (siehe hierzu Fujimoto in [17], S. 57ff.). Bei Verfahren, die rein auf fixen Zeitfortschritt setzen, entspricht der Lookahead dem Wert dieses Fortschritts und ist daher immer größer als null. Bei ereignisbasierten Simulationen kann sich der Wert des Lookaheads im Laufe der Simulation ändern, unter Umständen ist hier auch ein Wert von null möglich.

Zwei grundsätzliche Ansätze zur modellbasierten Parallelisierung existieren: Bei konservativen Verfahren wird durch technische Maßnahmen verhindert, dass Simulationser-

eignisse in der falschen Reihenfolge berechnet werden können, die Einhaltung der Kausalitätsbedingung wird somit immer garantiert. Optimistische Verfahren führen Simulationsereignisse so schnell wie möglich aus und nehmen dabei die Möglichkeit der Verletzung der Kausalitätsbedingung in Kauf. Falls solche Fehler auftreten, stellen die Methoden die Kausalität durch Verwerfen entsprechender Teile der bereits durchgeführten Simulation wieder her.

Bei der folgenden Betrachtung wird davon ausgegangen, dass zwischen den Prozessen gesendete Nachrichten auch sicher ankommen und in der Reihenfolge des Absendens beim Empfänger eingehen. Dies ist für das Funktionieren der beschriebenen Techniken nicht unbedingt notwendig, vereinfacht aber die Beschreibung, da so nicht auf Sonderfälle und Fehlerbehandlungen eingegangen werden muss. Zudem werden die Begriffe Ereignisse und Nachrichten synonym benutzt, da sie bei den Verfahren zur diskreten Simulation dieselbe Funktion haben, nämlich das Übermitteln von Informationen zum Verhalten der sendenden Entität an eine oder mehrere empfangende Entitäten.

2.2. Modellbasierte Parallelisierungsverfahren

Das oben beschriebene Synchronisierungsproblem ist die Ursache dafür, dass viele allgemeine Verfahren aus dem *Parallel Computing* nicht direkt zur effizienten Parallelisierung diskreter Simulationsmodelle eingesetzt werden können. Zu den auch in anderen Problemdomänen üblichen räumlichen oder regionalen Abhängigkeiten der Komponenten kommen in der diskreten Simulation die beschriebenen zeitlichen Abhängigkeiten der Modellentitäten hinzu: Wird ausschließlich auf eine hohe Auslastung abgezielt, können einzelne Prozessoren von Nachrichten aus der (aus lokaler Sicht) Vergangenheit erreicht werden, die die Simulation ungültig machen. Hieraus resultieren dann über das unabdingbare Neuaufnehmen der Simulation hinaus Komplikationen in so verschiedenen Bereichen wie Datenausgabe, Speichermanagement und Fehlerbehandlung (siehe Abschnitt 2.2.1.2).

Darum müssen effiziente Verfahren nicht nur eine hohe Auslastung der Prozessoren, sondern unter Beachtung der Synchronisierungsbedingung auch das gleichmäßiges Voranschreiten in der Simulationszeit in allen Teilmodellen zum Ziel haben. Die hierzu entwickelten Verfahren stellen dabei durch technische Maßnahmen entweder sicher, dass die Kausalität immer garantiert ist, oder bemerken und reparieren missachtete Kausalität.

Hohe Auslastung und gleichmäßiger Fortschritt widerstreben sich dann, wenn in einzelnen Teilmodellen deutlich mehr Aktivität herrscht als in anderen, oder sich die Last im Laufe der Simulation zwischen den Teilmodellen verlagert. Einzelne Prozessoren sind dann stark belastet, während andere auf deren Zwischenergebnisse warten, da sie nicht

selbständig in der Simulationszeit fortschreiten können. Da diese Modelldynamik typisch für diskrete Simulationsmodelle ist, spielen dynamische Lastausgleichsverfahren eine wichtige Rolle. Diese können jedoch nicht einfach Last zwischen schnellen und langsamen Prozessoren verlagern: Im Modell benachbarte Entitäten kommunizieren typischerweise viel untereinander, weit auseinander liegende typischerweise wenig. Eine rein auf gleichmäßige Auslastung zielende Lastverteilung kann daher den Effekt haben, dass sehr viel Kommunikationslast zwischen den Prozessoren entsteht, und so Teile der gewonnenen Laufzeitvorteile zunichte gemacht werden. Bei der Arbeit mit räumlich expliziten Modellen (wie z.B. Modellen aus dem Verkehrsbereich) sollte ein effektives Lastausgleichsverfahren daher Rücksicht auf den regionalen Zusammenhang des Modells nehmen. Achtet das Verfahren bei der Auswahl der zu verschiebenden Entitäten und der Zielprozessoren darauf, dass im Modell benachbarte Entitäten im selben Teilmodell bleiben, reduziert dies den Kommunikationsaufwand und trägt so sowohl zur hohen Auslastung als auch zum gleichmäßigen Simulationszeitfortschritt bei.

Die modellbasierten Parallelisierungsverfahren sind unterteilbar in allgemeine und anwendungsbezogene Verfahren. Die allgemeinen Methoden sind prinzipiell für alle diskreten Simulationsmodelle geeignet, sie unterscheiden sich in erster Linie durch die jeweils verwendete Synchronisierungsmethode. Des weiteren gibt es anwendungsbezogene Verfahren, die die allgemeinen Methoden anpassen oder erweitern. Die verschiedenen Ansätze werden in den folgenden Abschnitten kurz beleuchtet.

2.2.1. Allgemeine Parallelisierungsverfahren

Die allgemeinen, modellbasierten Parallelisierungsverfahren lassen sich wie beschrieben jeweils einer von zwei Kategorien zuordnen, die sich in der Art der Sicherung der Kausalitätsbedingung unterscheiden: Bei konservativen Verfahren wird zu jedem Zeitpunkt die Einhaltung der Kausalität garantiert, indem nur Simulationsereignisse verarbeitet werden, die explizit als sicher gelten. Optimistische Verfahren verzichten auf die strikte Einhaltung der Kausalität, jeder logische Prozess führt Ereignisse so schnell wie möglich aus. Empfängt nun ein Prozess ein Ereignis mit einem Zeitstempel, der aus lokaler Sicht in der Vergangenheit liegt, verwirft er entsprechende Teile der bereits durchgeführten Simulation und stellt die verletzte Kausalität durch Neuberechnen wieder her.

Im Folgenden werden die Grundlagen einer Auswahl von wichtigen konservativen und optimistischen Parallelisierungsverfahren beschrieben. Die beschriebenen Parallelisierungsverfahren wurden insbesondere für ereignisbasierte Systeme mit variablem Zeitfortschritt konzipiert, sind aber ebenso für Modelle mit fixem Zeitfortschritt geeignet.

2.2.1.1. Konservative Verfahren

Synchronisierung mit Null-Nachrichten Die Synchronisierung mittels Null-Nachrichten wurde als erstes konservatives Synchronisierungsverfahren für ereignisbasierte Simulationen unabhängig voneinander von Bryant in [6] und Chandy und Misra in [9] entwickelt (siehe hierzu und zu den nächsten Absätzen Fujimoto in [17], S. 54ff.).

Auch hier gilt wieder, dass die die einzelnen Teilmodelle berechnenden Prozessoren p_1 bis p_k aus P als Knoten eines Graphen angesehen werden können. Kann ein Prozessor p_i im Laufe der Simulation Nachrichten an einen Prozessor p_j senden, so existiert eine gerichtete Kante zwischen diesen Knoten.

Das im Folgenden beschriebene Verfahren setzt voraus, dass ein Prozessor p_i einem Prozessor p_j Nachrichten geordnet nach nicht-absteigendem Zeitstempel sendet. Eingehende Nachrichten speichert ein Prozessor in einer Reihe von FIFO-Warteschlangen, die jeweils einer eingehenden Kante zugeordnet sind. Aus der Voraussetzung folgt, dass Nachrichten in jeder dieser Warteschlangen in nicht-absteigender Ordnung gemäß ihrer Zeitstempel vorliegen. Bei einer Simulation mit variablem Zeitfortschritt werden zusätzlich lokale Nachrichten oder Ereignisse in einer eigenen Prioritätswarteschlange verwaltet.

Das Verfahren erklärt eine Nachricht mit Zeitstempel t dann für sicher, wenn in jeder Eingangs-Warteschlange mindestens eine Nachricht mit nicht-niedrigerem Zeitstempel als t anliegt. Das Vorliegen dieser Nachrichten bedeutet gemäß der Voraussetzung, dass kein Prozessor zukünftig Nachrichten senden kann, die in der Simulationszeit vor t liegen. Nun wählt der Prozessor die Nachricht N mit dem geringsten Zeitstempel aus allen eingehenden Warteschlangen und ggf. der lokalen Ereignisliste. Da nachträglich keine Nachricht mit geringerem Zeitstempel eintreffen kann, bleibt bei der Verarbeitung von N die Kausalitätsbedingung gewahrt.

Bei der Verarbeitung eines Ereignisses werden ggf. weitere Ereignisse mit gleichem oder höherem Zeitstempel an benachbarte Prozessoren gesendet. Ohne weitere Maßnahmen kann es zu Deadlocks kommen: Sind bei keinem Prozessor alle Warteschlangen an den eingehenden Kanten gefüllt, so wartet jeder Prozessor auf das Eintreffen von Nachrichten der anderen Prozessoren (siehe Abbildung 2.1). Daher können keine Ereignisse für sicher erklärt werden, die Simulation blockiert.

Um dieses Problem zu lösen, schlagen Bryant in [6] und Chandy und Misra in [9] vor, dass jeder Prozessor nach dem Verarbeiten einer Nachricht sog. Null-Nachrichten an sämtliche benachbarten Prozessoren schickt. Diese Nachrichten erhalten als Zeitstempel die aktuelle Simulationszeit plus den Lookahead-Wert L des Prozessors.

Die Handhabung von Null-Nachrichten durch den empfangenden Prozessor entspricht der von regulären Nachrichten. Allerdings wird bei der Bearbeitung einer Null-Nachricht

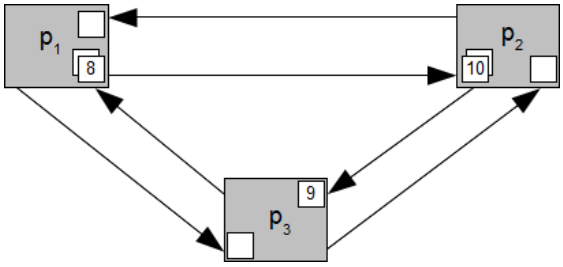


Abbildung 2.1.: Deadlock: Jeder Prozessor wartet auf das Eintreffen von Nachrichten an den Eingangs-FIFOs (nach Fujimoto in [17], S. 56)

lediglich die Simulationszeit auf deren Zeitstempel angepasst, ansonsten werden keine Änderung am Modellzustand vorgenommen. Algorithmus 2.1 zeigt den Chandy/Misra/Bryant-Algorithmus zur Synchronisation mittels Null-Nachrichten (beschrieben z.B. von Fujimoto in [17], S. 57f.). Durch das Versenden von Null-Nachrichten bei jeder Ereignisverarbeitung bleibt sicher gestellt, dass in den FIFO-Warteschlangen der Prozessoren zu jedem Simulationszeitpunkt Nachrichten vorhanden sind. Ein Deadlock, wie er in Abbildung 2.1 gezeigt wird, ist dadurch ausgeschlossen.

Die Effizienz des Verfahrens ist weitgehend von der Höhe des Lookahead-Werte abhängig: Ein geringer Lookahead führt dazu, dass sehr viele Null-Nachrichten verschickt und bearbeitet werden müssen. Hinzu kommt, dass das Modell keine Kreise im Graphen mit Lookahead von $L = 0$ enthalten darf, da sonst Deadlock-Situationen möglich werden. Hier werden dann von den beteiligten Prozessoren ausschließlich Null-Nachrichten verarbeitet und (wegen $L = 0$) weitere Null-Nachrichten mit dem gleichen Zeitstempel aneinander versendet. Die Simulationszeit schreitet nie voran, die Anwendung ist in einer Endlosschleife gefangen.

Deadlocks erkennen und auflösen Der beschriebene Chandy/Misra/Bryant-Algorithmus vermeidet das Auftreten von Deadlocks durch das Versenden von Null-Nachrichten. Unter Umständen kommt es dabei zu einem relativ hohen Verwaltungsoverhead, da eine große Zahl von Null-Nachrichten nur sehr selten auftretenden Deadlocks gegenüber stehen kann. Eine alternative Herangehensweise an dieses Problem ist, das Entstehen von Deadlocks zunächst zuzulassen, ihr Auftreten zu erkennen und sie dann durch geeignete Maßnahmen zu beheben (siehe hierzu und zu den nächsten Absätzen Fujimoto in [17], S. 60ff.). Das im Folgenden beschriebene Verfahren wurde erstmals von Chandy und Misra in [10] vorgestellt. Dabei arbeiten alle Prozessoren als sicher erkannte Ereignisse so schnell wie möglich ab, zunächst ohne auf das Vermeiden von Deadlocks zu achten.

Algorithmus 2.1 Chandy/Misra/Bryant-Algorithmus (nach Fujimoto in [17], S. 57)

```
Solange (Simulation ist nicht beendet) {
        Warte, bis jede Eingangs-FIFO mindestens eine
                Nachricht enthält;
        Nimm Nachricht N mit dem kleinsten Zeitstempel
                aus ihrem FIFO;
        simulationszeit = Zeitstempel von N;
        Führe N aus;
        Berechne Lookahead L;
        Sende Null-Nachrichten mit (simulationszeit+L) an
                alle benachbarten Teilmodelle;
        }
```

Mit einem einfachen Verfahren können dann dabei auftretende Deadlocks behoben werden. Ausgangsannahme dazu ist, dass sich alle Prozesse im Deadlock befinden, ein einzelner Prozessor ist als Controller ausgezeichnet. Sicher ist die Ausführung der Nachricht mit dem über alle Prozessoren geringsten Zeitstempel; diese Nachricht würde in einer sequentiellen Simulation als nächstes verarbeitet. Die einzelnen Prozessoren können aus ihren lokalen Datenbeständen jedoch nicht ersehen, welche Nachricht dies ist. Um dieses Problem zu lösen, fordert der Controller nun von jedem Prozessor den minimalen Zeitstempel der dort unverarbeitet anliegenden Nachrichten an. Dies ist ohne weiteres möglich, da die Ereignisverarbeitung bei allen Prozessoren blockiert ist, also aktuell keine Nachrichten abgearbeitet werden. Der Controller kann aus den Angaben der einzelnen Prozessoren die Nachricht(en) mit dem global niedrigsten Zeitstempel bestimmen. Diese Nachrichten werden für sicher erklärt, der entsprechende Prozessor aus dem Deadlock befreit.

Das Anfordern der minimalen Zeitstempel kann durch direkte Nachrichten an die beteiligten Prozessoren geschehen, diese antworten dann direkt an den Controller. Eine Alternative, insbesondere geeignet bei sehr vielen beteiligten Prozessoren, ist das Bilden eines Spannbaums aus den Prozessoren, dessen Wurzel der Controller ist. Die Prozessoren senden die Nachrichten dann jeweils an ihre Töchterknoten. Falls ein Prozessor in diesem Spannbaum ein Blatt ist, so sendet er den minimalen Wert direkt an seinen Vaterknoten zurück, innere Knoten bilden das Minimum aus den von den Töchtern übertragenen Werten und dem eigenen kleinsten Zeitstempel und senden ihn dann an ihren Vaterknoten weiter. Der Controller erhält so den global minimalen Wert und kann damit die Ereignisse mit diesem Zeitwert für sicher erklären und den entsprechenden Prozessor anstoßen.

Ausschließlich Ereignisse mit einem einzelnen Zeitstempelwert für sicher zu erklären, ist dabei arg konservativ. Die im Modell angelegte Parallelität kann besser ausgenutzt werden, indem unter Verwendung des Lookaheads eine größere Menge von Nachrichten für sicher erklärt wird. Bei einem globalen Lookahead von L und dem Prozess mit der niedrigsten lokalen Simulationszeit t können dort alle Nachrichten im Intervall $[t, t + L]$ für sicher erklärt werden.

Um Deadlocks beheben zu können, muss deren Auftreten zunächst erkannt werden. Auch für hierzu genutzte Verfahren gilt wieder die Ausgangsannahme, dass sich alle Prozesse im Deadlock befinden, wieder ist ein einzelner Prozessor als Controller ausgezeichnet. Der Controller sendet nun wie beschrieben einer Menge von Prozessoren die Nachricht, dass einige Ereignisse sicher sind und verarbeitet werden können. Die Prozessoren verarbeiten diese sicheren Ereignisse und generieren damit weitere Nachrichten, die im besten Fall benachbarte Prozessoren aus der Blockade holen. Dieses Ausbreiten der Berechnungstätigkeit kann als Baum angesehen werden, mit dem Controller als Wurzel. Blockierte Prozessoren sind dabei nicht Teil des Baumes. Werden Nachrichten an einen blockierten Prozess geschickt und dieser so aus dem Deadlock befreit, so wird dieser Prozessor Teil des Baums, mit dem sendenden Prozess als Vaterknoten. Bekommt ein bereits aktiver Prozess eine Nachricht geschickt, so sendet er sofort eine Antwort an den sendenden Prozessor zurück, dass der Baum sich nicht ausgeweitet hat. Wenn nun ein Prozessor, der ein Blatt dieses Baums ist, keine sicheren Nachrichten mehr ausführen kann und deswegen blockieren muss, sendet er ein Signal an seinen Vater und entfernt sich selbst aus dem Baum. Jeder Prozessor führt Buch über die Anzahl der versendeten und unbeantwortet gebliebenen Nachrichten. Ist die Anzahl gleich null, so ist der Prozessor ein Blatt im Prozessor-Baum. Wenn nun der Controller zum Blatt in diesem Baum wird, müssen sich wieder alle Prozessoren im Deadlock befinden. Das Verfahren kann nun erneut angewendet werden.

Im Gegensatz zur Synchronisierung mit Null-Nachrichten sind bei dem beschriebenen Verfahren Kreise mit einem Lookahead $L = 0$ zulässig. Ein weiterer Vorteil tritt bei der Verwendung mit ereignisbasierten Simulationsmodellen auf: Beim beschriebenen Verfahren wird der Wert des Zeitstempels des nächsten unbearbeiteten Ereignisses beachtet, hier kann ein u.U. großer Sprung in der Simulationszeit erfolgen, unabhängig vom Wert des Lookaheads. Bei der Synchronisierung mit Null-Nachrichten ist dies anders, dort ist der Zeitfortschritt allein vom (möglicherweise geringen) Lookahead abhängig.

Synchrone Ausführung Bei mit synchroner Ausführung arbeitenden Verfahren (siehe hierzu und zu den nächsten Absätzen Fujimoto in [17], S. 65ff.) führt jeder Prozessor die

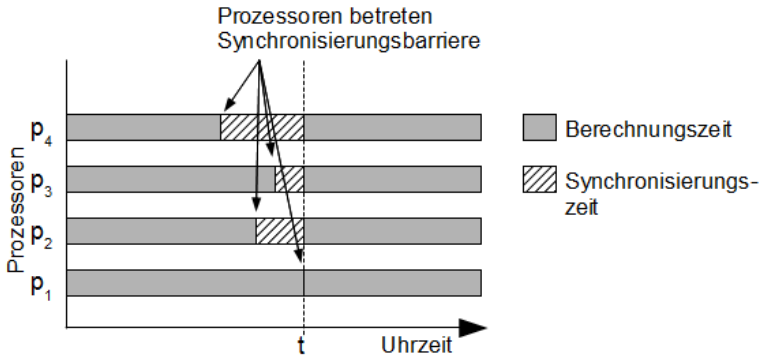


Abbildung 2.2.: Berechnungs- und Synchronisierungszeiten bei synchroner Ausführung (Quelle: Fujimoto in [17], S. 65)

als sicher erkannten Ereignisse oder (bei fixem Zeitfortschritt) Simulationsschritte eines sicheren Simulationszeitintervalls aus. Nachdem dies erledigt ist, betritt der Prozessor eine Synchronisierungsbarriere. Hier wartet jeder Prozessor darauf, dass auch alle anderen Prozessoren den Bearbeitungsschritt beendet haben. Danach wird das nächste, nun für sicher erklärte Intervall berechnet. Es gibt also einen definierten (Uhr-)Zeitpunkt, an dem alle Prozessoren mit der Berechnung eines bestimmten Simulationszeitintervalls fertig sind, und bevor sie mit dem Berechnen des nächsten Zeitintervalls beginnen (siehe Abbildung 2.2).

Algorithmus 2.2 zeigt ein einfaches Protokoll zur synchronen Ausführung, wie es ähnlich von Nicol in [53] und von Steinman in [64] vorgeschlagen wird. Zum Erkennen der sicheren Ereignisse wird hier wieder der Lookahead verwendet. $t(i)$ ist dabei die Simulationszeit der nächsten unbearbeiteten Nachricht in Prozessor p_i, $L(i)$ ist dessen Lookahead-Wert. t_L sei der minimale Wert von $t(i) + L(i)$ über alle Prozessoren. Sicher sind dann alle Nachrichten, die mit Zeitstempeln von bis zu t_L versehen sind.

Die Synchronisierungsbarriere kann dabei auf verschiedene Arten realisiert werden, einige häufig verwendete Typen werden im Folgenden kurz beschrieben.

Bei der Synchronisierung mit *Baumbarrieren* werden die Prozessoren als balancierter Spannbaum angesehen, der Wurzel-Prozessor ist wieder als Controller ausgezeichnet. Ein Blatt-Prozessor, der mit dem Berechnungsschritt fertig ist und in die Barriere eintreten will, sendet eine *barrier*-Nachricht an seinen Vaterknoten im Baum und wartet dann auf Antwort. Ein innerer Knoten, der in die Barriere eintreten will, wartet auf Nachrichten seiner Tochterknoten. Sind diese vollständig, sendet er wiederum eine *barrier*-Nachricht an seinen Vaterknoten und wartet dann auf dessen Antwort. Ist der Controller mit der Berechnung des Intervalls fertig und hat von allen Töchtern *barrier*-Nachrichten empfan-

Algorithmus 2.2 Einfaches synchrones Simulationsprotokoll (nach Fujimoto in [17], S. 75)

```
Solange (Simulation ist nicht beendet) {
        Tmin = Minimales (t(i)+L(i)) über alle
                Prozessoren P(i);
        S = Menge aller anliegenden Nachrichten des lokalen
                Prozessors mit (Zeitstempel<=Tmin);
        Führe Ereignisse in S aus;
        Führe Barriere aus;
}
```

gen, befinden sich sämtliche Prozessoren in der Barriere. Um die Barriere zu lösen und die nächste Berechnungsphase einzuleiten, versendet der Controller jetzt *release*-Nachrichten an seine Töchter, die diese wiederum an ihre Töchter weiter senden.

Ein Spezialfall von Baumbarrieren ist die sog. *zentrale Barriere*. Hier werden alle Prozessoren unmittelbar von einem Controller synchronisiert. Eine zentrale Barriere lässt sich als Baum der Tiefe 1 ansehen, der Controller ist dabei die Wurzel des Baumes, alle anderen Prozessoren sind Blätter. Nachteil bei der ansonsten sehr effizient zu implementierenden zentralen Barriere ist das lineare Wachstum der Nachrichtenanzahl, die vom Controller verarbeitet werden muss. Bei einer sehr großen Anzahl von beteiligten Prozessoren kann dies zu Engpässen führen.

Eine für die Synchronisierung einer großen Zahl von Prozessoren geeignete Methode ist die *Schmetterlingsbarriere*. Um die Betrachtung dieser Methode zu vereinfachen, soll dabei von einer Zahl von $k = 2^n$ beteiligten Prozessoren ausgegangen werden. Jeder Prozessor p_i muss im Laufe der Synchronisierung log k paarweise Barrieren ausführen. Für diese paarweisen Barrieren wird ein einfaches Handshake-Protokoll genutzt: Prozessor p_i sendet für die Synchronisierung mit Prozessor p_j eine *barrier*-Nachricht an p_j und wartet dann auf dessen Antwort in Form einer weiteren *barrier*-Nachricht. Ist diese eingetroffen, wurde die paarweise Synchronisierung durchgeführt.

Im Rahmen der Schmetterlingsbarriere wird der Index i eines Prozessors p_i als Binärstring der Länge log k angesehen. Der Prozessor p_i führt nun paarweise Barrieren mit einer Reihe von Prozessoren aus, beginnend mit dem Prozessor, dessen Index sich nur an der letzten Binärstelle von seinem eigenen Index unterscheidet. Die nächste paarweise Barriere wird dann mit dem Prozessor durchgeführt, dessen Binärstring sich an der vorletzten Stelle unterscheidet. Dieser Prozess wird log k mal ausgeführt, von der letzten bis zur ersten Binärstelle.

Schritt	Index Prozessor i	Index Prozessor j
1	101**1**	101**0**
2	10**1**1	10**0**1
3	1**0**11	1**1**11
4	**1**011	**0**011

Tabelle 2.1.: Ausführung der paarweisen Barrieren aus Sicht von Prozessor $i = (1011)b$ in einer Schmetterlingsbarriere für 16 Prozessoren

In Tabelle 2.1 wird das Vorgehen aus Sicht eines Prozessors mit dem Index $(1011)b = (11)d$ in einem System mit 16 Prozessoren gezeigt: Nach der ersten paarweisen Barriere weiß Prozessor 11, dass Prozessor 10 mit den Berechnungen fertig ist. Im zweiten Schritt kommt Prozessor 9 hinzu, der seinerseits in Schritt 1 mit Prozessor 8 synchronisiert wurde. Im dritten Schritt kommt Prozessor 15 hinzu, der sich wiederum in Schritt 1 mit Prozessor 14 und in Schritt 2 mit Prozessor 12 synchronisiert hat, der sich wiederum mit Prozessor 13 synchronisiert hat. Nach dem dritten Schritt sind also die Prozessoren 8, 9, 10, 11, 12, 13, 14 und 15 miteinander synchronisiert. Im vierten Schritt wird Prozessor 11 mit Prozessor 3 synchronisiert, der in den vorherigen Schritten die Prozessoren 1, 2, 4, 5, 6 und 7 synchronisiert hat. Nach vier Schritten sind also alle 16 Prozessoren miteinander synchronisiert.

Die Synchronisierungsnachrichten können für das Versenden von Datenwerten genutzt werden, wie Lookahead-Werte bei den *barrier*-Nachrichten und die Grenzen der für sicher erklärten Zeitintervalle in den *release*-Nachrichten.

Außer dem Vorhandensein eines positiven Lookaheads zur Bestimmung der Größe der im nächsten Schritt zu berechnenden Simulationszeitintervalle sind für das beschriebene Verfahren keine Voraussetzungen nötig. Insbesondere gibt es keine Anforderungen an die Verbindungen zwischen den einzelnen Teilmodellen, diese können sich im Laufe des Modells ändern, da auf den Füllstand (oder das Vorhandensein) von FIFO-Wartenschlangen nicht geachtet werden muss. Allerdings ist der Simulationsfortschritt des Gesamtsystems - wie bei anderen konservativen Parallelisierungsverfahren auch - vom Fortschritt des langsamsten Teilmodells abhängig. Ein Lastausgleichsverfahren, das Entitäten aus Teilmodellen auf Prozessoren mit Überlast löst und auf Prozessoren mit Unterlast verschiebt, kann den Ablauf beschleunigen. So ein Verfahren kann bei der synchronen Ausführung im Rahmen der sowieso stattfindenden Synchronisierungsschritte günstig implementiert werden, da hier die Berechnung von Nachrichten aussetzt und relativ einfach Operationen an den eingefrorenen Teilmodellen vorgenommen werden können.

2.2.1.2. Optimistische Verfahren

Das wichtigste optimistische Verfahren wird von Jefferson in [28] vorgestellt, er prägt dafür den schönen Namen „Time Warp" (hierzu und zu den folgenden Abschnitten siehe auch Fujimoto in [17], S. 97ff.). Die Aufgaben des Parallelisierungsverfahren sind dabei aufgeteilt in einem lokalen und einem globalen Kontrollmechanismus. Die im lokalen Mechanismus durchgeführten Berechnungen finden lokal auf jedem Prozessor statt, die Prozessoren können dabei weitgehend unabhängig voneinander arbeiten. Vom globalen Mechanismus werden Tätigkeiten wie Ein- und Ausgabe und das Sammeln von nicht mehr benötigtem Speicher durchgeführt, hierzu ist eine Synchronisierung der Prozessoren nötig.

Lokaler Kontrollmechanismus Wie bei anderen ereignisbasierten Methoden auch werden durch die einzelnen Prozessoren Ereignisse aus der lokalen *Future Event List (FEL)* ausgeführt, dabei werden ggf. die Zustandsvariablen des Modells verändert. Die Ereignisse werden nach der Bearbeitung jedoch nicht verworfen, sondern in einer weiteren Liste, der *Processed Event List (PEL),* gespeichert. Trifft eine Nachricht von einem anderen Prozessor ein, deren Zeitstempel größer oder gleich der aktuellen Simulationszeit des lokalen Teilmodells ist, so wird sie in die FEL eingefügt und normal verarbeitet.

Trifft eine Nachricht N ein, deren Zeitstempel t vor der lokalen Simulationszeit liegt (in der Literatur auch als *straggler message* bezeichnet), so muss das Modell auf seinen Zustand zum Zeitpunkt t zurück gesetzt werden (man spricht hier von einem Rollback), die durch das Auftreten des Stragglers ungültig gewordenen Änderungen ab diesem Zeitpunkt müssen rückgängig gemacht werden. Weiterhin müssen die bereits bearbeiteten Ereignisse mit einem größeren Zeitstempel als t aus der PEL herausgeholt und zur Neuverarbeitung wieder in die FEL eingefügt werden. Die Nachricht N wird ebenfalls in die FEL eingefügt.

Um diesen Rollback durchzuführen, bieten sich zwei Verfahren an: Beim vollständigen Sichern (engl. *copy state saving*) werden vor jeder Ereignisverarbeitung die Werte aller Zustandsvariablen gespeichert. Trifft nun ein Straggler ein, wird der gesicherte Zustand mit dem entsprechenden Zeitstempel in die Zustandsvariablen zurück kopiert, die gemachten Änderungen werden so verworfen. Die wie beschrieben aus der PEL in die FEL zurück kopierten Ereignisse können nun unter Berücksichtigung des Stragglers verarbeitet werden, die Kausalitätsfehler sind behoben. Beim inkrementellen Sichern (engl. *incremental state saving*) wird vor jeder Änderung einer Zustandsvariablen ein Logeintrag verfasst, der die aktuelle Simulationszeit, einen Verweis auf die entsprechende Zustandsvariable und deren Wert vor der Änderung enthält. Dieser Logeintrag wird dann in einen

Stack-Speicher eingefügt, der über alle Änderungen der Zustandsvariablen Buch führt. Trifft nun ein Straggler ein, so werden alle Logeinträge mit einem größeren Zeitstempelwert als dem des Stragglers vom Stack genommen und die gespeicherten Zustandswerte nacheinander in die Arbeitsvariablen zurück übertragen. So wird der Modellzustand zur entsprechenden Simulationszeit wieder hergestellt, der Straggler kann mit den aus der PEL zurück kopierten Ereignissen in die FEL eingefügt und verarbeitet werden.

Falls bei einem bestimmten Modell während der Ereignisverarbeitung viele Variablen verändert werden, bietet sich ein vollständiges Sichern an. Falls nur wenige Variablen verändert werden, bietet sich inkrementelles Sichern an, da hier nur die Variablen gespeichert werden, die auch wirklich geändert werden. Beide Techniken können auch gemeinsam eingesetzt werden: In bestimmten Abständen oder bei einer großen Anzahl geänderter Variablen wird eine vollständige Sicherung durchgeführt, kleinere Änderungen werden inkrementell gesichert.

Unter Umständen muss nicht nur der lokale Modellzustand zurück gesetzt werden: Falls im Nachhinein für ungültig erklärte Nachrichten an andere Prozesse geschickt wurden, müssen diese Nachrichten zurück geholt werden (engl. *unsending messages*) und deren Effekte beim Empfängerprozess rückgängig gemacht werden. Dazu verwendet das Time Warp-Verfahren sogenannte Anti-Nachrichten (engl. *anti-messages*). Jede Anti-Nachricht N_A korrespondiert mit genau einer regulären gesendeten Nachricht N. Beim Eintreffen einer Anti-Nachricht bei einem Prozessor wird automatisch die dazu gehörige reguläre Nachricht aus der entsprechenden Datenstruktur (FEL oder PEL) gelöscht. Die Anti-Nachricht wird dabei ebenfalls vernichtet.

Von Nachrichten, die an andere Prozesse geschickt werden, speichert der Prozess eine Kopie in einer lokalen Output Queue. Wird jetzt das Modell auf seinen Zustand zum Simulationszeitpunkt t zurück gesetzt, wird zu jeder dort gespeicherten Nachricht mit Zeitstempel größer als t eine Anti-Nachricht N_A an den entsprechenden Zielprozess gesendet.

Beim Eintreffen von N_A gibt es zwei Möglichkeiten: Falls die Nachricht N vom Zielprozessor noch nicht verarbeitet wurde, wird sie aus der FEL gelöscht. Die Anti-Nachricht N_A wird dann ebenfalls vernichtet. Weitere Maßnahmen sind nicht nötig, die Simulation kann fortgesetzt werden. Falls die Nachricht N bereits verarbeitet wurde, muss ein Rollback bis zu dem Simulationszeitpunkt durchgeführt werden, der dem Zeitstempel von N entspricht. Gegebenenfalls müssen nun weitere Anti-Messages an dritte Prozesse versendet werden, die dann eventuell wieder Rollbacks ausführen müssen. Im Laufe dieser Kaskade von Rollbacks und Anti-Messages werden alle inkorrekten Berechnungen rückgängig gemacht, die Kausalität wird wieder hergestellt. Die Simulation kann nun wieder

aufgenommen werden.

Zu beachten ist, dass Rollbacks sich nicht auf den Simulationsstand kleiner oder gleich t auswirken, d.h. Berechnungen bis zum Simulationszeitpunkt der *Straggler*-Nachricht erhalten bleiben. Daher kann gesagt werden, dass zumindest die Verarbeitung des Ereignisses mit dem systemweit kleinsten Zeitstempel nicht zurück genommen werden wird. Es gibt also eine untere Schranke der Simulationszeit, unter deren Wert nicht mehr zurück gegangen wird.

Globaler Kontrollmechanismus Für die Funktion des globalen Kontrollmechanismus ist der Begriff der globalen virtuellen Zeit (engl. *Global Virtual Time)* zu einem Uhrzeitpunkt t (GVT_t) zentral: Dies ist der minimale Zeitstempel aller unverarbeiteten oder teilweise verarbeiteten Ereignisse über alle beteiligten Prozessoren zu einem Uhrzeitpunkt t. Wie bereits beschrieben, ist sicher gestellt, dass keine Rollbacks auf Zeitpunkte kleiner als GVT_t stattfinden.

Bei der Berechnung von GVT_t müssen bereits abgesendete, beim Empfänger aber noch nicht eingegangene Nachrichten beachtet werden. Da diese transienten Nachrichten beim Empfänger einen Rollback auslösen und so die lokale Simulationszeit verringern können, kann nicht ohne weiteres zum Zeitpunkt t die Simulation eingefroren und das Minimum über alle lokalen Simulationszeiten gebildet werden. Zur Lösung dieses Problems bietet sich ein einfaches Protokoll an, bei dem jeder Empfänger einer Nachricht N diesen Empfang beim Sender bestätigt. Bis zum Eingang dieser Bestätigung ist der Sender für die Nachricht N verantwortlich und muss sie in die Berechnung des lokalen Minimums einbeziehen, danach der Empfänger von N. Somit ist garantiert, dass die Simulationszeit von N in die Berechnung eingeht.

Eine Reihe von synchronen und asynchronen Verfahren zur Berechnung von GVT_t sind bekannt (hierzu und zu den nächsten Abschnitten siehe auch Fujimoto in [17], S. 112ff.) Ein einfaches, synchrones Verfahren stützt sich auf die in Abschnitt 2.2.1.1 beschriebenen Synchronisierungsbarrieren: Ein ausgezeichneter Controller signalisiert den Prozessoren, dass sie die Modellberechnung unterbrechen sollen. Diese bestätigen den Eintritt in die Barriere beim Controller. Im nächsten Schritt signalisiert der Controller, dass die Prozessoren den minimalen Zeitstempel berechnen sollen aus allen nicht oder teilweise berechneten lokalen Ereignissen, Nachrichten, Anti-Nachrichten und denjenigen gesendeten Nachrichten, für die noch keine Empfangsbestätigung eingegangen ist. Diese lokalen Minima werden an den Controller gesendet, der das globale Minimum berechnet und den Prozessoren mitteilt. Daraufhin kann die Berechnung der Simulation fortgesetzt werden. Ein Nachteil dieses synchronen Verfahrens ist, dass die Ereignisverarbeitung bei

allen Prozessoren angehalten werden muss. Um dies zu vermeiden, wurden eine Reihe von asynchronen Verfahren entwickelt, die es ermöglichen, GVT_t ohne globale Synchronisierung zu berechnen (siehe z.B. Samadi in [55] und Mattern in [44]).

Die berechnete GVT_t wird dann von den einzelnen Prozessoren für eine Reihe von Verwaltungsarbeiten eingesetzt. Zu den wichtigsten gehört dabei das sog. Einsammeln von Fossil-Zuständen: Ohne weitere Maßnahmen steigt der Speicherverbrauch durch das wiederholte Anlegen von Sicherungskopien des Modellzustands im Laufe der Simulation immer weiter an. Da die Simulation nicht unter den Wert der Global Virtual Time zurück genommen werden kann, können Sicherungskopien zu Modellzeitpunkten kleiner als GVT_t gelöscht werden, deren Speicher kann also freigegeben werden.

Zudem können Ein-/Ausgabe-Operationen i.d.R. nicht zurück genommen werden. Daher dürfen Ausgaben zu Simulationsereignissen erst gemacht werden, wenn die aktuelle GVT bis zumindest zum Simulationszeitpunkt des Ereignisses fortgeschritten ist.

Ein weiterer zu beachtender Sonderfall ist die Bearbeitung von Programm- und Berechnungsfehlern: Diese können aufgrund von Kausalitätsfehlern auftreten, z.B. bei einer negativen Zahl von Flugzeugen auf einem Rollfeld oder einer durch eine falsch berechnete Zahl von Werkstücken entstehende Division durch null. Das Programm darf dann nicht einfach beendet werden, da diese Fehler ggf. durch Rollbacks wieder rückgängig gemacht werden. Ein festgestellter Fehler darf dem Benutzer erst dann angezeigt werden, wenn die aktuelle GVT bis zum Zeitstempels der Fehlers fortgeschritten ist.

2.2.1.3. Vergleich der Verfahren

Das bestmögliche Verfahren zur Simulation eines konkreten Modells ist weitgehend abhängig von dessen Eigenschaften, kein einzelnes Verfahren ist für alle Anwendungen optimal (siehe hierzu und zu den folgenden Abschnitten auch Fujimoto in [17], S. 172ff.)

Konservative Verfahren sind tendenziell weniger komplex im Aufbau und arbeiten mit geringerem Overhead als optimistische Methoden. Sie arbeiten nur mit einem einzelnen Satz Zustandsvariablen, das Anlegen und Verwalten von Sicherungen ist nicht nötig.

Falls bereits bestehende sequentielle Implementierungen für den Parallelbetrieb umgerüstet werden sollen, ist die Nutzung eines konservativen Verfahrens mitunter günstiger, da keine Änderungen für Zustandssicherung, Ein-/Ausgabe, Fehlerbehandlung oder Speichermanagement in das Programm eingefügt werden müssen. Einschränkend gilt allerdings, dass bei einigen Verfahren, wie der Synchronisierung mit Null-Nachrichten oder dem Entdecken und Auflösen von Deadlocks, die Topologie des Modells zu Beginn der Simulationsläufe bekannt sein muss. Verfahren wie die synchrone Ausführung erlauben auch dynamische Topologien.

Da konservative Verfahren nur explizit für sicher erklärte Ereignisse oder Zeitinkremente ausführen, sich also an Worst-Case-Szenarien orientieren, nutzen sie nicht das komplette Parallelisierungspotential des Modells aus. Konservative Verfahren sind also ggf. übermäßig pessimistisch. Generell gilt: Je höher der Lookahead-Wert ist, umso mehr Ereignisse oder Zeititerationen können parallel verarbeitet werden, und umso mehr kann die modellinhärente Parallelität genutzt werden. Falls aber ein Modell eine hohe Nachrichtendichte aufweist, insbesondere falls viele Ereignisse mit gleichem Zeitstempel auftreten, ist auch ein niedriger Lookahead nicht nachteilig.

Ein großer Vorteil optimistischer Verfahren ist sicherlich, dass auch Modelle mit einem Lookahead von null ohne weitere Einschränkungen effizient berechenbar sind. Zudem ist auch hier die Ausführung von Modellen möglich, bei denen zu Simulationsbeginn noch nicht bekannt ist, welche Teilmodelle im Laufe der Simulation miteinander kommunizieren. Die parallele Ausführung wird dabei nicht wie bei konservativen Verfahren gehindert von potentiellen Abhängigkeiten zwischen Teilmodellen, sondern nur von tatsächlich auftretenden Abhängigkeiten.

Sind diese Abhängigkeiten hoch oder sind die Prozessoren durch sich dynamisch verändernde Aktivitäten im Modell unterschiedlich ausgelastet, verhalten sich diese Verfahren zu optimistisch, so dass eine Kaskade von Fehlberechnungen durchgeführt wird, die durch aufwändige Rollback-Operationen wieder zurück genommen werden muss (siehe hierzu auch Lubachevsky, Schwartz und Weiss in [36]). Um dies zu ermöglichen, ist bei jeder Veränderung ein komplettes oder inkrementelles Sichern des Modellzustands nötig. Dies kostet Aufwand bei Entwicklung, Rechenzeit und Speicher. Unter anderem um den hierdurch entstehenden Speicherbedarf begrenzen zu können, ist der Einsatz von Mechanismen zur Berechnung der Global Virtual Time nötig. Für Tätigkeiten wie Ein-/Ausgabe, Fehlerbehandlung oder Speichermanagement, für die in konservativen Verfahren wie beschrieben die üblichen Bibliotheksfunktionen genutzt werden können, benötigen optimistische Verfahren eigens implementierte, Rollback-sichere Funktionen.

Zusammenfassend lässt sich sagen, dass optimistische Verfahren dazu neigen, komplexer und aufwändiger zu sein als konservative Methoden. Gerade bei bekanntem - und im Idealfall im Vergleich zur Ereignisdichte großem - Lookahead wirkt sich der geringere Overhead der konservativen Verfahren positiv auf die Leistung aus. Falls aber ein Lookahead-Wert nicht bekannt oder im Vergleich zur Ereignisdichte sehr gering ist, haben die optimistischen Verfahren Vorteile in der Performanz.

2.2.2. Anwendungsbezogene Parallelisierungsverfahren

Bei beiden Familien von Parallelisierungsverfahren bleibt wie bereits beschrieben der Simulationsfortschritt des Gesamtmodells vom Fortschritt des langsamsten Teilmodells abhängig. Ein Lastausgleichsverfahren, das Entitäten aus Teilmodellen auf Prozessoren mit Überlast löst und auf Prozessoren mit Unterlast verschiebt, kann den Ablauf der Simulation deutlich beschleunigen. Um effektiv zu arbeiten, sollte ein solches Verfahren dabei die Charakteristika des Modells und ebenso die Leistungsfähigkeit der einzelnen Prozessoren berücksichtigen. Im Folgenden sollen einige Lösungen betrachtet werden, die sich in konkreten Anwendungsfällen bewährt haben.

Die Kausalitätsbedingung muss natürlich auch hier eingehalten werden. Die anwendungsbezogenen Verfahren können daher jeweils als Anpassungen oder Erweiterungen der allgemeinen Parallelisierungsverfahren angesehen werden.

2.2.2.1. Parallele Simulation von Schaltkreisen

Als erstes anwendungbezogenes Verfahren wird ein Verfahren zur Parallelisierung der Simulation des dynamischen Verhaltens von logischen Schaltungen betrachtet, das von Schlagenhaft, et al. in [58] und von Schlagenhaft in [57] beschrieben wird. Es handelt sich dabei um eine parallele, ereignisbasierte Simulation, die gemäß des optimistischen Time Warp-Verfahrens abläuft. Die Rechner stehen der Anwendung dabei nicht exklusiv zur Verfügung, sondern werden auch durch Drittprozesse genutzt. Schlagenhaft, et al. beschreiben ein dynamisches und adaptives Lastausgleichsverfahren, um die vorhandenen Ressourcen bestmöglich zu nutzen.

Die untersuchten logischen Schaltungen bestehen aus Schaltelementen, zwischen denen Abhängigkeiten in Form von binären Signalen bestehen. Im Modell wird jedes Schaltelement als Entität abgebildet; diese werden im Rahmen der statischen Partitionierung vor dem Simulationslauf zu Clustern zusammengefasst, die dann zu Partitionen zusammen gefügt und den einzelnen Prozessoren zugewiesen werden. Diese Cluster werden einzeln verwaltet, es existiert pro Cluster je eine FEL. So können Entitäten während des Lastausgleichs clusterweise migriert werden. Ein weiterer Vorteil der Aufteilung der Partitionen in einzelne Cluster kommt im Falle eines Rollbacks zum Tragen: Hier muss nicht für die gesamte Partition die Simulation zurück genommen und neu berechnet werden, sondern nur für wenige, im Idealfall sogar nur für ein einzelnes Cluster.

Als Partitionierungsmethode wird der von Sporrer und Bauer in [63] beschriebene Corolla-Ansatz genutzt: Hierbei werden Modellregionen mit einem hohen Verhältnis interner zu externer Verbindungen erkannt und diese sog. Corollas zu Clustern verbunden,

so dass die Zahl der Verbindungen zwischen Clustern minimiert wird. Die Cluster werden dann zu den Partitionen zusammen gefasst.

Da die Bestandteile des Modells je nach Modellzustand und Eingabedaten unterschiedlich aktiv sein können, wird ein dynamischer Lastausgleich benötigt. Da außerdem von durch Drittprozesse belasteten Prozessoren und Übertragungsnetzwerken ausgegangen wird, ist das vorgestellte Ausgleichsverfahren zudem adaptiv.

Der globale Kontrollmechanismus des Time Warp wird um ein Lastausgleichsverfahren erweitert, das einzelne Cluster zwischen den Partitionen verschieben kann. Hierzu werden im Rahmen der Berechnung der GVT die drei Stufen Lastmessung, Lastbewertung und Lastverschiebung durchgeführt.

Zur Lastmessung wird der virtuelle Zeitfortschritt (Virtual Time Progress, VTP) genutzt. Hierzu wird zuerst durch jeden Prozessor die virtuelle Simulationszeit (Integrated Virtual Time, IVT) berechnet (siehe Formel 2.2).

$$\begin{aligned} IVT_0 &= 0 \\ IVT_{i+1} &= IVT_i + \Delta T_i \qquad falls\, \Delta T_i \geq 0 \\ IVT_{i+1} &= IVT_i \qquad\qquad\quad falls\, \Delta T_i < 0 \end{aligned} \tag{2.2}$$

Der Wert ΔT_i entspricht dem Fortschritt der Simulationszeit zwischen zwei Messungen, der Uhrzeitpunkt der Messung i ist dabei t_i. Bei der regulären Simulation ist ΔT größer null, bei Rollbacks kleiner null. Bei der Berechnung der IVT werden also später rückgängig gemachte Zeitfortschritte berücksichtigt, da deren Berechnung ja reale Last verursacht. Nun kann der virtuelle Zeitfortschritt berechnet werden (siehe Formel 2.3).

$$VTP_i = \frac{IVT_i - IVT_{i-1}}{t_i - t_{i-1}} \tag{2.3}$$

Diese Berechnung beinhaltet sowohl den Fortschritt der Simulationszeit (inkl. später zurück genommenen Berechnungen) als auch die Dauer der Berechnung. Daher beachtet dieses Lastmaß sowohl innere als auch äußere Störungen: Kommt es z.B. im verwalteten Teilmodell während einer bestimmten Simulationszeitspanne zu einer veränderten Anzahl zu bearbeitender Ereignisse, so wirkt sich dies auf den virtuellen Zeitfortschritt innerhalb des Messzeitraums und somit auf das Lastmaß aus; Änderungen in der Leistungsfähigkeit der Prozessoren wirken sich über die Dauer der Abarbeitung der Ereignisse und somit den zeitlichen Abstand zwischen zwei Messungen ebenfalls auf das Lastmaß aus.

Im Rahmen der Lastbewertung werden der gemäß VTP schnellste Prozessor p_f und der langsamste Prozessor p_s als Quell- resp. Zielprozessor für die Lastverschiebung ausgewählt. Basierend auf der Differenz der VTP-Werte der beiden Prozessoren wird in p_s

nach jenem Cluster c gesucht, das am besten geeignet ist, durch die Verlagerung nach p_f diese Differenz auszugleichen.

Nun wird eine Rentabilitätsabschätzung durchgeführt, bei der der Zeitaufwand t_M für die Migration mit eingerechnet wird, in der die Prozessoren p_s und p_f nicht simulieren können: Dazu wird unter Berücksichtigung von t_M eine Prognose VTP_{neu} über den Zeitfortschritt zu einem vorgegebenen maximalen Uhrzeitwert unter der Annahme einer durchgeführten Verlagerung berechnet, zum Vergleich wird ein Wert von VTP_{min} ohne durchgeführte Verlagerung angenommen. Falls nun TVP_{neu} größer ist als VTP_{min}, dann wird Cluster c von p_s nach p_f verschoben.

Um die Lastverschiebung durchzuführen, wird die Ereignisverarbeitung der Prozessoren p_s und p_f angehalten, und dann sowohl die statische Modellstruktur des Clusters c als auch der aktuelle Zustand der einzelnen Entitäten von p_s an p_f übertragen. Nachdem dann noch die FEL des Clusters übertragen wurde, kann p_f die Simulation fortsetzen. Zum Abschluss teilt p_s den anderen Prozessoren die Zugehörigkeit von c zum von p_f verwalteten Teilmodell mit und setzt dann die Abarbeitung der Simulation fort. Falls noch einzelne Nachrichten für c bei p_s eintreffen, werden sie an p_f weiter gesendet.

Schlagenhaft, et al. betrachten zuerst die Auswirkungen der Verwendung von Clustern im Vergleich zur direkten Verwaltung der Modellentitäten durch die Prozessoren. Unter idealen Bedingungen (fünf Sun-Workstations stehen durchgehend und exklusiv zur Verfügung, das Netzwerk ist unbelastet) messen sie eine leichte Verschlechterung der Leistung wegen des zusätzlichen Aufwands durch die Verwaltung der Cluster. Unter als realistisch angenommenen Bedingungen zeigen sich jedoch Verbesserung um bis zu 30% im Bereich von 60 bis 90 Clustern mit ca. 200 bis 400 Schaltungselementen pro Cluster.

Diese Parameter werden dann als Grundlage für das Lastausgleichsverfahren verwendet, bei dessen Einsatz bereits auf zwei Prozessoren und einer Belastung durch externe Prozesse (siehe Schlagenhaft, et al. in [58]) eine Verbesserung der Laufzeit um ca. 24% erreicht wird. Schlagenhaft berichtet in [57] dann von Verbesserungen von bis zu 60% beim Einsatz von sechs Prozessoren in belasteten Netzwerken.

Ein ähnliches Verfahren wird von Avril und Tropper in [2] und [3] beschrieben. Dort wird die Veränderung der Kommunikationslast durch den Lastausgleich beachtet, die von Schlagenhaft et al. als unkritisch beurteilt wird. Wenn ein Lastausgleich nötig wird, wählen Avril und Tropper als Quell- und Zielpartition nicht diejenigen mit der höchsten resp. niedrigsten Last aus, sondern bilden Mengen möglicher Quell- und Zielpartitionen aus ähnlich belasteten Partitionen. Ihr Verfahren geht dann alle zu den möglichen Quellpartitionen gehörenden Cluster durch und überprüft für jede mögliche Zielpartition die Auswirkungen der Verlagerung dieses Clusters auf die Kommunikation zwischen den

Partitionen. Ausgewählt werden dann ein zu verschiebendes Cluster und eine Zielpartition so, dass diese Kommunikation minimiert wird. Avril und Topper kommen zu grob vergleichbaren Ergebnissen wie Schlagenhaft, et al., sie messen für ihr Verfahren eine Beschleunigung des Ablaufs von 40% im Vergleich zum einfachen Time Warp.

Offensichtlich lohnt sich im beschriebenen Fall der Einsatz eines Lastausgleichsverfahrens. Die Methode beachtet dabei aufbauend auf einem geeigneten Lastmaß sowohl innere als auch äußere Störungen, ist also dynamisch und adaptiv. Die Bildung der einzelnen Cluster und deren Zusammensetzung zu Partitionen werden so durchgeführt, dass möglichst wenig Kommunikation zwischen den Prozessoren nötig wird. Avril und Topper beachten im Rahmen des dynamischen Lastausgleichs zusätzlich bei der Auswahl eines zu verschiebenden Clusters und dessen Zielpartition, dass die Kommunikationslast durch den Lastausgleich nach Möglichkeit weiter sinkt.

Obwohl Abhängigkeiten der Cluster voneinander durch die abgebildeten Signalverbindungen zwischen den Schaltelementen vorhanden und bekannt sind, wird dieses Wissen über die Topologie des Modells nicht genutzt. Die Cluster werden nicht entlang dieser Verbindungen zu benachbarten Partitionen verschoben, sondern zumindest bei Schlagenhaft, et al. rein nach ihrem Gewicht gehend. Um die Kommunikation zu minimieren, ist aber die Beachtung der Lage der Cluster im Modell sicherlich von Vorteil. Auch Avril und Topper beachten dieses Wissen über Nachbarschaften nur implizit, da die Verschiebung von Clustern entlang von Signalwegen an benachbarte Partitionen oft die für die Kommunikationslast günstigste Variante sein wird.

2.2.2.2. Parallele Simulation biologischer Alterung

In der von Meisgen in [45] und [46] beschriebenen Simulation des biologischen Alterungsprozesses wird der Lebenszyklus von Lebewesen modelliert. Die Individuen werden geboren, altern, pflanzen sich fort und sterben entweder durch Unfälle und Krankheiten oder zu von Eigenschaften ihrer Gensequenz bestimmten Zeitpunkten.

Das diskrete Simulationsmodell wird in fixen Zeitinkrementen berechnet, die jeweils einem Schritt von einem Jahr entsprechen. Die simulierte Auslese kann im Laufe der Simulation dazu führen, dass große Teile der Population von wenigen Individuen abstammen, während die anderen aussterben. Ohne Lastausgleich kommt es so zu einer starken Überlast bei einigen Prozessoren, bei anderen zu langen Idle-Zeiten.

Die Anwendung nutzt ein konservatives Parallelisierungsverfahren mit synchroner Ausführung, die die Zeitinkremente trennende Baumbarriere wird zum Austausch von Messwerten für den Lastausgleich genutzt. Meisgen untersucht das Verhalten seiner Methode auf einem exklusiv zur Verfügung stehenden, homogenen Parallelrechner und in inhomo-

genen Workstation-Clustern, deren Kommunikationsnetzwerk und Rechenknoten durch externe Prozesse belastet sein können. Sein Lastausgleich soll auch diese äußeren Störungen beachten.

Im Modell bestehen nur sehr geringe Abhängigkeiten der Entitäten voneinander, ihr Verhalten wird nur beeinflusst von der zwischen den Simulationsschritten erhobenen Gesamtzahl der Individuen. Aufgrund dieser mangelnden Abhängigkeiten ist das Modell nicht unbedingt typisch für die Problemstellungen im Bereich der modellbasierten Parallelisierung. Wegen der quasi lehrbuchhaften Verwendung eines konservativen Parallelisierungsverfahrens inklusive eines dazu gehörigen dynamischen und adaptiven Lastausgleichs wird die Anwendung hier trotzdem vorgestellt.

Wie bereits beschrieben, werden die Berechnungen der einzelnen Zeitschritte synchron ausgeführt. In der Baumbarriere werden die Größe der Gesamtpopulation berechnet und der Lastausgleich durchgeführt. Das Ausgleichsverfahren besteht auch hier wieder aus den drei Stufen Lastmessung, Lastbewertung und Lastverschiebung.

Die Last $a_t(p)$ eines Prozessors p zu einem Zeitpunkt t entspricht der Größe der von p simulierten Population, die durchschnittliche Last über alle Prozessoren wird mit $\overline{a}$ bezeichnet. Die Last $a(P')$ der Prozessormenge P' ist als Summe aller Einzellasten definiert. Zudem wird für jeden Prozessor p ein Leistungskoeffizient $w(p)$ dynamisch erhoben, der seine Leistungsfähigkeit im Vergleich zu einem Benchmark-Prozessor p_1 ausdrückt; der Wert wird berechnet als $w(p) = \frac{t_c(p_1,i)}{t_c(p_1,i)}$, also als Verhältnis der Berechnungszeit $t_c(p_1, i)$ eines Simulationsjahrs auf Benchmark-Prozessor p_1 zur Berechnungszeit auf Prozessor p. Der Leistungskoeffizient $w(P')$ einer Prozessormenge P' entspricht der Summe der Einzelwerte. Die optimale Last a_{opt} eines Prozessors p entspricht dann der mit seinem Leistungskoeffizienten gewichteten Durchschnittslast: $a_{opt,i} = w(p) * \overline{a}$.

Im Laufe der Synchronisierung in der Baumbarriere berechnet jeder Prozessor p lokal die Über- oder Unterlastung des Teilbaums B_p, dessen Wurzel p ist. Hierzu beginnt jedes Blatt p des Baumes mit der Berechnung von $w(p)$ und $a_t(p)$ und sendet diese Werte an seinen Vaterknoten. Innere Knoten summieren die Werte ihrer Kinder, addieren die eigenen hinzu und senden diese ebenfalls zu ihrem Vaterknoten. Am Ende dieser Phase verfügt die Wurzel r des Prozessorbaumes dann über die summierten Werte aller Prozessoren. Nun wird die Durchschnittslast $\overline{a} = \frac{a(B_r)}{w(B_r)}$ berechnet und durch den Baum zurück an alle Prozessoren gesendet.

Der Lastbewertung liegt die Annahme zugrunde, dass die Last eines Prozessors in Schritt t proportional zur Berechnungsdauer des Teilmodells für den nächsten Zeitschritt $t + 1$ ist.

Um eine Überkompensation entstehender Imbalancen zu vermeiden, wird ein Para-

meter q eingeführt, der die maximal zulässige Lastunausgeglichenheit bestimmt. Die Lastverschiebung wird nur dann durchgeführt, wenn der maximal belastete Prozessor mindestens q mal so schwer belastet ist wie der minimal belastete, wenn also die Bedingung $\frac{a_{max}}{a_{min}} > q$ erfüllt ist.

Jeder Knoten prüft nun für seinen Vaterknoten, ob er an diesen Last abgeben oder von ihm anfordern muss. Ist die Last des Teilbaums B_p größer als die mit dem Leistungskoeffizienten gewichtete Durchschnittslast, gilt also Formel 2.4, dann müssen Individuen an den Vaterknoten von p abgegeben werden, anderenfalls muss von dort weitere Last angefordert werden.

$$a(B_p) - w(B_p) * \overline{a} > 0 \tag{2.4}$$

Die Lastverschiebung besteht dann aus dem Übertragen einer angemessenen Zahl an Individuen vom überlasteten Prozessor an den unterlasteten Prozessor. Sie kann relativ simpel durchgeführt werden, da diese Prozessoren im Baum direkt verbunden sind und alle Prozessoren die Berechnung für den Synchronisierungsschritt eingestellt haben, auf laufende Berechnungen also nicht geachtet werden muss. Das verwendete Lastmaß berücksichtigt den Fortschritt in der Simulationszeit (hier fix je ein Jahr) und durch den dynamisch erhobenen Leistungskoeffizienten auch den Uhrzeitbedarf für die Berechnung dieses Fortschritts. Somit werden sowohl äußere als auch innere Störungen abgebildet.

Meisgen führt eine Reihe von Experimenten aus, dabei nutzt er sowohl einen exklusiv zur Verfügung stehenden, homogenen Parallelrechner, als auch inhomogene Workstations, die durch ein Local Area Network (LAN) resp. ein Wide Area Network (WAN) verbunden sind. Er geht mit seinen Versuchen also bis in den Bereich der verteilten Simulation hinein.

Der homogene Parallelrechner mit 64 Prozessoren benötigt für einen Lauf ohne Lastausgleich 5.923 Sekunden, davon verbringt jeder Prozessor im Durchschnitt 4.297 Sekunden im Idle-Betrieb. Mit Lastausgleich ($q = 1,05$) sinkt die Laufzeit auf 28,7% oder 1.698 Sekunden, davon im Schnitt nur 56 Sekunden im Idle-Betrieb. Der starke Einfluss des Lastausgleichs kann hier durch die enorme Dynamik der Last erklärt werden: Zum Ende des Laufs ist die Population auf einigen Rechnern quasi ausgestorben, andere tragen jeweils einen Großteil der Population. Der am wenigsten ausgelastete Prozessor verwaltet 21.055 Individuen, die größte Teilpopulation enthält über 2 Mio. Individuen.

Im inhomogenen LAN-Netzwerk mit Belastung durch externe Prozesse gelingt eine Reduktion der Laufzeit auf 46% (mit $q = 1,5$), die Idle-Zeit erhöht sich auf 369 Sekunden. Meisgen misst hier höhere Variationen der Laufzeit, die Simulation verbringt mehr Zeit in der Synchronisierungsbarriere. Auch bei Versuchen mit durch ein WAN mit seiner

vergleichsweise hohe Latenz verbundenen Workstations wird die Laufzeit mit $q = 2,0$ immer noch auf 46% reduziert.

Weitere Experimente zeigen, dass die Güte des Lastausgleichs von der Größe des zu berechnendem Modell abhängt. Gerade in LAN und WAN sind die zwischen zwei Schritten stattfindenden Übertragungen über das Netzwerk ein entscheidender Kostenfaktor. Je schneller die für die Synchronisierung nötigen Nachrichten übertragen werden oder je größer das Verhältnis der Berechnungszeit zur Übertragungszeit ist, umso besser skaliert das Verfahren.

Ein geeigneter Wert des Parameters q ist abhängig von der konkreten Rechnerplattform, insbesondere von den Übertragungszeiten des Netzwerks. Der Parameter wird bei Meisgen per Hand gesetzt. Wünschenswert wäre hier ein Regelkreis, der den Parameter in Abhängigkeit von der Reaktion der Umgebung dynamisch anpasst, so dass auch Veränderungen in der Auslastung des Netzwerks und den damit einhergehenden Übertragungszeiten ausgeglichen werden können.

2.2.2.3. Parallele Simulation des Ausbreitens von Krankheiten

Als dritte Anwendung von modellbasierten Parallelisierungsverfahren soll nun noch die Simulation der Ausbreitung von Lyme-Borreliose betrachtet werden, die von Deelman, Szymanski und Caraco in [13] und Deelman und Szymanski in [12] beschrieben wird. Diese Krankheit wird durch Zeckenbisse übertragen und befällt Säugetiere. Räumlich gesehen breitet sich die Krankheit durch die Bewegung der Wirtstiere aus, die Zecken reisen mit ihnen und sind sonst quasi unbeweglich.

Das räumlich explizite Modell arbeitet ereignisbasiert und wird mit einer Adaption des Time Warp-Verfahrens simuliert. Die Partitionierungsmethode, das verwendete Lastmaß und das Lastausgleichsverfahren nutzen die Zweidimensionalität des Modells aus. Der zu simulierende Bereich ist in quadratische, 400 Quadratmeter große Gebiete aufgeteilt, die jeweils als Entität betrachtet werden. Die Wirte werden als sich frei bewegende Individuen modelliert, die Zecken wegen ihrer hohen Anzahl (bis zu 1.200 pro Gebiet) und ihrer Bewegungslosigkeit als statischer Teil des Hintergrunds. Sie befallen möglicherweise bereits erkrankte Individuen, reisen mit diesen mit, fallen nach einer Weile ab und und übertragen die Krankheit ggf. an nachfolgende Wirte.

Während die meisten Simulationsereignisse lokal sind, also nur ein einzelnes Gebiet betreffen, treten auch eine Reihe nicht-lokaler Ereignisse wie die Wanderung von Individuen über Gebietsgrenzen auf. Dabei werden dann nicht nur Ereignisnachrichten an benachbarte Gebiets-Entitäten versendet, sondern auch einzelne Individuen in Nachrichten gekapselt. Wird ein Individuum an einen Nachbarn verschickt, wird es von der senden-

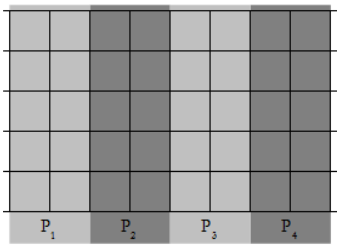


Abbildung 2.3.: Streifenweise Zuordnung der modellierten Gebiete zu den Prozessoren

den Entität in eine *Ghost List* kopiert. Muss dann im Laufe der Simulation ein Rollback erfolgen, kann das Objekt der Ghost List entnommen und wieder eingesetzt werden.

Wie bereits beschrieben, besteht der modellierte Bereich aus einer Matrix von 20x20 Meter messenden Quadraten. Diese Matrix wird zu einem Torus geschlossen und ist somit in einer Dimension endlos. Sie wird zur statischen Partitionierung vor dem Simulationslauf in Spalten aufgeteilt, nebeneinander liegende Spalten werden dann einem Prozessor zugeordnet. Das gesamte Modell wird also in einzelne, eine oder mehrere Spalten breite Streifen aufgeteilt (siehe Abbildung 2.3). In einem torusförmigen Modell hat jeder Prozessor genau zwei Nachbarn, die Prozessoren bilden also einen Ring. Die dynamische Lastverschiebung erfolgt ausschließlich zwischen unmittelbaren Nachbarn, und nur über die Verschiebung der äußeren Spalten des dem Prozessor zugewiesenen Bereichs.

Auch in diesem Verfahren lässt sich das Lastausgleichsverfahren wieder in die drei Abschnitte Lastmessung, Lastbewertung und Lastverschiebung einteilen. Deelman und Szymanski verwenden eine zukunftsorientierte Lastabschätzung: Dabei werden die Ereignisse in der FEL gezählt und gemäß ihres Zeitstempels gewichtet. So sollen sogenannte *Hot Spots*, also Simulationszeitspannen mit besonders hoher Ereignisdichte, schon vor der Bearbeitung erkannt und somit entsprechend ausgeglichen werden können. Da jedoch ständig neue Ereignisse generiert werden, altert die Abschätzung schnell und muss häufig wiederholt werden.

Die geschätzte Last F_j einer Spalte j wird wie in Formel 2.5 dargestellt berechnet.

$$F_j = \sum_{0 \leq k < n(j)} 2^{-time(y_k)} \tag{2.5}$$

Dabei bezeichnet y_k das k-te Ereignis aus der FEL der Spalte j, die die Länge $n(j)$ hat. Das Ereignis hat den Zeitstempel $time(y_k)$.

Die lokale Lastmessung erfolgt während der für den Time Warp notwendigen Berechnung der Global Virtual Time, die Werte werden von einem ausgezeichneten Controller

gesammelt und von dort aus an alle beteiligten Prozessoren verteilt.

Die Last eines Prozessors p_i wird vor dem Lastausgleich mit a_i bezeichnet, nach dem erfolgten Lastausgleich mit b_i. Die durchschnittliche Last über alle Prozessoren wird wieder mit $\overline{a}$ bezeichnet. Gesucht ist für jeden Prozessor p_i ein Wert x_i, der die optimale, zwischen p_i und $p_{i\oplus 1}$ zu verschiebende Last bestimmt. Die Operatoren $\oplus$ und $\ominus$ bezeichnen hier eine Addition resp. Subtraktion modulo k. Ein Wert von $x_i > 0$ bezeichnet eine Lastverschiebung von p_i nach $p_{i\oplus 1}$, ein $x_i < 0$ bezeichnet eine Lastverlagerung in umgekehrter Richtung. Da die Last eines Prozessors nie negativ werden kann, gilt dabei für jeden Prozessor i: $-a_{i\oplus 1} \leq x_i \leq a_i$.

Der Erfolg des Lastausgleichs wird danach bewertet, wie hoch die größte Einzellast nach dem erfolgten Lasttransfer ist. Gesucht ist also für jedes Prozessorpaar p_j und $p_{j\oplus 1}$ ein Lasttransfer x_j, so dass der über alle Prozessoren p_i größte Wert von b_i minimiert wird. Deelman und Szymanski lösen dieses Problem mit einem exakten Verfahren: In einem Ring aus k Prozessoren können $k*(k-1)$ verschiedene Ketten mit mindestens zwei Prozessoren gebildet werden. Daher können durch einfaches Enumerieren der möglichen Ketten in $O(k^2)$ diejenigen Ketten benachbarter Prozessoren identifiziert werden, die die maximale Durchschnittslast tragen. Die längste dieser Ketten wird als dominante Kette bezeichnet. Ausgehend von dieser dominanten Kette werden die x_i so gesetzt, dass die die Durchschnittslast $\overline{a}$ übersteigende Last aus der Kette an die Nachbarprozessoren übertragen wird, die Last wird innerhalb der Kette dabei gleich verteilt. Im besten Fall kann die Last in einem einzelnen Schritt optimal verteilt werden. Da x_i jedoch durch $-a_{i\oplus 1}$ und a_i begrenzt ist, können in einigen Situationen mehrere Iterationen notwendig sein, z.B. wenn mehrere stark unterlastete Prozessoren nebeneinander liegen.

Damit ein Prozessor p_i (angenommen sei $x_i > 0$, der Fall $x_i < 0$ und damit $x_{i\ominus 1} > 0$ lässt sich durch eine Indexverschiebung im Ring um eine Stelle nach links abbilden) nun die Lastverschiebung durchführen kann, muss die Anzahl der zu migrierenden Spalten berechnet werden. Dazu werden vom Rand des vom Prozessor verwalteten Streifens beginnend die Lasten der nebeneinander liegenden Spalten so lange addiert, bis die Zunahme einer weiteren Spalte die zu verschiebende Gesamtlast x_i überträfe. Die Spalten werden dabei als zu übertragen markiert. Hier geht die Exaktheit der Lösung verloren, denn die Summe der Spaltenlast entspricht i.d.R. nicht genau dem optimalen Werte x_i. Um keine Überkompensation vorzunehmen und so ein durch wiederholtes Hin- und Zurückübertragen derselben Spalten entstehendes Flattern (engl. *Thrashing*) zu verhindern, wird im Zweifel eine geringere Last übertragen. Die markierten Gebiete werden dann spaltenweise an den Zielprozessor verschoben, inklusive der dazu gehörenden Individuen und des Inhalts der Future Event List.

Besondere Rücksicht muss genommen werden, falls eine von Prozessor p_i nach $p_{i\oplus 1}$ verschobene Spalte das Ziel der nicht-lokalen Bewegung eines Individuums ist: Falls das Startgebiet dieser Bewegung weiterhin von p_i verwaltet wird, entspricht die Verschiebung der Zielspalte dem Senden einer das Individuum kapselnden Nachricht von p_i nach $p_{i\oplus 1}$. Daher muss p_i Vorkehrungen für einen möglichen Rollback treffen und für das betreffende Individuum einen Eintrag in seiner Ghost List vornehmen.

Deelman und Szymanski führen Experimente auf einem Parallelrechner mit 28 Prozessoren durch, der exklusiv zur Verfügung steht. Im Vergleich zu einem (bewusst ungleich partitionierenden) statischen Verfahren verkürzt der dynamische Lastausgleich die Laufzeit um bis zu 20%.

Das beschriebene Verfahren arbeitet dynamisch, jedoch nicht adaptiv, da die Last allein durch eine Abschätzung der Belastung durch die Ereignisverarbeitung gemessen wird. Von Beginn an unterschiedliche oder im Laufe der Simulationsläufe schwankende Rechenkapazitäten werden ignoriert. Das Verfahren eignet sich daher nicht für den Einsatz in einem belasteten Netzwerk oder auf durch Drittprozesse genutzten Rechnern, sondern zeigt nur auf exklusiv zur Verfügung stehenden, homogenen Parallelrechnern die gewünschte Beschleunigungswirkung.

Das Verfahren nutzt die Räumlichkeit des Modells teilweise aus, das Modell wird spaltenweise an die einzelnen Prozessoren verteilt. Diese starre Aufteilung wird damit begründet, dass eine exakte Lastbewertung auf diese Weise einfach durchzuführen sei, dies sei bei komplexeren Nachbarschaftsstrukturen nicht möglich. Die Exaktheit der in diesem Schritt erreichten Lösung geht jedoch schon bei der Bestimmung der zu migrierenden Spalten wieder verloren. Möglicherweise wäre eine heuristische Lösung, die jedoch eine freiere räumliche Partitionierung mit mehr als zwei Nachbarn pro Prozessor ermöglicht, hier effizienter.

2.2.2.4. Vergleich der Verfahren

Jede der beschriebenen Anpassungen oder Erweiterungen allgemeiner Verfahren nutzt Wissen über Charakteristika des zugrunde liegenden Simulationsmodells aus:

Das von Schlagenhaft, et al. beschriebene Verfahren teilt die Modellentitäten in Cluster ein, da ein Großteil des Nachrichtenaustauschs innerhalb kleiner Gruppen von Schaltelementen stattfindet. Dieser Umstand wird ausgenutzt, um bei Rollback-Schritten Laufzeit zu sparen, und außerdem den Kommunikationsaufwand zwischen den Partitionen niedrig zu halten. Zudem können so im Rahmen des Lastausgleichs mit geringem Aufwand mehrere Entitäten gleichzeitig verschoben werden, ohne das komplette Modell neu partitionieren zu müssen.

Bei dem von Meisgen beschriebenen Modell sind die Teilmodelle weitgehend unabhängig, zwischen den simulierten Zeitintervallen müssen jedoch einige Werte global errechnet und kommuniziert werden. Meisgen wählt daher ein mit synchroner Ausführung arbeitendes Verfahren, hier kann die Kommunikation zwischen den Simulationsschritten in der Synchronisierungsbarriere stattfinden.

Deelman und Szymanski berücksichtigen bei ihrem Verfahren die Räumlichkeit des Modells. Durch die statische Partitionierung an Spaltengrenzen entlang reduzieren sie Kommunikationsaufwand, und halten diesen auch durch Migration kompletter Spalten im Rahmen des dynamischen Lastausgleichs niedrig.

Die verwendeten Lastausgleichsverfahren sind dynamisch und zum Teil auch adaptiv. Alle drei Verfahren nutzen für die Durchführung des Lastausgleichs die durch das Synchronisierungsverfahren vorgegebene Infrastruktur, wie die Berechnung der Global Virtual Time bei den auf Time Warp aufsetzenden Beispielen, und die Synchronisierungbarriere beim synchron arbeitenden Verfahren.

Bei der Simulation des dynamischen Verhaltens von Schaltkreisen werden Entitäten clusterweise verlagert. Dies spart im Vergleich zur Einzelübertragung mehrerer Schaltelemente Aufwand und hält die Kommunikationskosten niedrig, da wie beschrieben ein Großteil der häufig genutzten Signalpfade innerhalb der Cluster verläuft. Das Verfahren nutzt jedoch die räumlichen Zusammenhänge des Modells nicht aus, obwohl diese bekannt sind.

Beim von der Simulation biologischer Alterung verwendeten Lastausgleichsverfahren bestimmt ein Parameter q, ab welchem Grad von Unausgeglichenheit eine Lastverlagerung stattfindet. So wird eine Überkompensation vermieden, zudem werden durch die Trägheit der Netzwerke entstehende Verzögerungen reduziert. Durch die Verwendung eines zugelassenen Grades an Unausgeglichenheit wird so verhindert, dass beim Versuch, den Ausgleich zu gut zu machen, überproportional viel Aufwand betrieben wird, und die positiven Laufzeiteffekte des Lastausgleichs dadurch wieder vernichtet werden. Der Parameter q muss entsprechend der Modellgröße und der Geschwindigkeit des verwendeten Netzwerks manuell gesetzt werden. Naheliegend wäre hier eine Erweiterung um einen sich dynamisch anpassenden Wert, der so auch wechselnde Leistung des Netzwerks abfedert.

Bei der Lastmessung im Rahmen der Simulation des Ausbreitens von Krankheiten werden lediglich Eigenschaften des Modells einbezogen, insbesondere die Anzahl und Zeitstempel der Ereignisse in der Future Event List. Das Verfahren ist nicht adaptiv, da ein Bezug zum für die Simulation nötigen Uhrzeitbedarf oder die vorhandenen Ressourcen fehlt. Die spaltenweise Verlagerung der simulierten Gebiete ist ein erster Versuch zum Ausnutzen der Räumlichkeit des Modells. Hierdurch wird im Vergleich zum Über-

tragen willkürlicher Gebiete aus der Mitte des verwalteten Gebietes der Kommunikationsaufwand gering gehalten. Die grobe Verlagerung an starren Grenzen und das relativ aufwändige exakte Verfahren zur Lastbeurteilung, dessen Ergebnis im nächsten Schritt dann aber seine Optimalität verliert, machen jedoch ein gewisses Verbesserungspotential deutlich.

Die beschriebenen Experimente zeigen, dass die Anwendung eines dynamischen Lastausgleichsverfahrens einen deutlichen Laufzeitgewinn bringen kann. Insbesondere bei Modellen mit starker dynamischer Modellaktivität treten bei fehlendem Lastausgleich hohe Ineffizienzen auf: Bei optimistischen Verfahren werden sehr viele Rollbacks nötig, wenn ein Teil der Prozessoren zu schnell in der Simulationszeit fortschreitet. Bei konservativen Verfahren wird im gleichen Fall ein hoher Synchronisierungsoverhead nötig, da - je nach Verfahren - viele Null-Nachrichten, viele Deadlock-Situationen oder lange Idle-Zeiten in der Synchronisierungsbarriere in Kauf genommen werden müssen.

Die beschriebenen Verfahren zeigen nochmals, dass für parallele Simulationen spezielle Lastausgleichsverfahren nötig sind, die neben der möglichst hohen Auslastung der Prozessoren zugleich auch ein möglichst gleichmäßiges Voranschreiten in der Simulationszeit als Ziel haben.

3. Ein Ansatz zur parallelen Simulation von Stadtbahnfahrplänen

In diesem Kapitels wird ein modellbasiertes Parallelisierungsverfahren entworfen, das insbesondere zur Simulation von Stadtbahnfahrplänen geeignet ist. Die für die Modellierung dieser Fahrpläne notwendigen Konzepte werden in Abschnitt 6.1.1 ausführlich beschrieben, daher erfolgt hier nur eine kurze Zusammenfassung der wichtigsten Begriffe.

Ein Stadtbahnnetz besteht aus Haltepunkten, Weichen und den sie verbindenden Streckenabschnitten, die nach den Vorgaben eines Fahrplans von einzelnen Fahrzeugen befahren werden. Jede dieser Bahnen verlässt zu Beginn des Betriebstags eines der zum Netz gehörenden Depots und führt dann jeweils eine vorgegebene Reihe von Fahrten aus, die zu einzelnen Linien gehören. Im Rahmen jeder dieser Fahrten bedient das Fahrzeug eine vorgegebene Reihe von Haltepunkten. Üblicherweise werden die meisten Strecken dabei im Laufe des Betriebstags nur in eine Richtung befahren. Außer durch den Fahr- und Streckenplan wird die Bahn vom sie umgebenden Verkehr und von Signalen beeinflusst, die Streckenabschnitte, Weichen oder Haltepunkte zum Befahren freigeben oder sperren, oder andere Verhaltensregeln wie einzuhaltende Höchstgeschwindigkeiten oder Haltezeiten bestimmen können. Zum Ende des Betriebstags kehrt jedes Fahrzeug wieder in ein Depot zurück.

Bei der diskreten Modellierung müssen die beschriebenen Systemkomponenten abgebildet werden. Die physischen Komponenten Haltepunkte, Strecken und Weichen lassen sich dabei als residente Entitäten modellieren. Sie können als Knoten eines Modellgraphen angesehen werden, die über gerichtete Kanten miteinander verbunden sind. Dabei bleiben im Stadtbahnnetz benachbarte Komponenten auch im Modell benachbart. Fahr-, Linien- und Umlaufpläne sind logische Systemkomponenten und müssen ebenfalls abgebildet werden. Dies ist möglich in Form von Listen, die von Fahrzeugen oder anderen Komponenten verwaltet werden. Die Fahrzeuge lassen sich als transiente Entitäten modellieren, die während des Simulationslaufs entsprechend der durch Pläne und Signale gemachten Vorgaben durch das Modell reisen.

Ein auf diese Weise erstelltes Modell weist eine Reihe von Eigenschaften auf, die im

zu entwickelnden Parallelisierungsverfahren ausgenutzt werden sollen:

- Das Modell ist räumlich explizit und kann als dünn besetzter Graph angesehen werden. Für eine Abbildung eines Stadtbahnnetzes ist das offensichtlich gegeben, ebenso für viele andere Straßen-, Schienen- und Flugverkehrsmodelle. Das Modell beachtet dabei die Räumlichkeit des Stadtbahnnetzes, benachbarte Systemkomponenten sind auch im Modell benachbart.

- Die Abhängigkeiten im Modell sind typischerweise regional. Auch dies ist bei Stadtbahnmodellen gegeben: Leitet eine simulierte Bahn ein Bremsmanöver ein oder schließt den Passagierwechsel ab, so beeinflusst das ggf. die benachbarten Fahrzeuge, aber nicht Entitäten in weit entfernten Bereichen des Modells.

- Die Modellaktivität verlagert sich während des Simulationslaufs durch das Modell. Die Modellaktivität wird im Wesentlichen von den transienten Entitäten verursacht, die sich während des Betriebstags durch das modellierte Netz bewegen. Also ist auch die Rechenlast dynamisch über die Modellbereiche verteilt.

- Diese Lastverlagerung erfolgt gutartig: die Last verlagert sich stetig und relativ langsam über mehrere Simulationsschritte hinweg. Auch dies ist im betrachteten Modell gegeben: Die simulierten Bahnen fahren vorgegebene Routen stetig ab, kein Fahrzeug verlässt das Netz plötzlich und taucht an einem entfernten Haltepunkt wieder auf.

- Die Ereignisdichte ist im Vergleich zur Auflösung der Simulationszeit relativ hoch. Bei einer zeitlichen Auflösung von einer Sekunde ändern in einem Stadtbahnmodell die Bahnen sekündlich Position und Geschwindigkeit, zugleich sind relativ viele Entitäten im Modell aktiv. Signale wechseln häufig ihren Zustand, dabei halten sie sich an bekannte Strategien. Stadtbahnnetze lassen sich ohne Probleme so modellieren, dass der Lookahead zumindest der Auflösung der Simulationszeit entspricht, also größer null ist.

Diese Eigenschaften treten nicht ausschließlich bei Modellen von Stadtbahnnetzen auf, sondern sind auch bei anderen Modellen aus der Verkehrssimulation und weiteren Bereichen gegeben. Daher kann der beschriebene Ansatz auch bei der Simulation dieser Modelle eingesetzt werden.

Das Verfahren soll über die Ausnutzung der beschriebenen Eigenschaften hinaus noch weitere Bedingungen beachten: Ein auf Basis des Verfahrens entwickeltes Simulationsmodul soll auf einem vom Benutzer auch für andere Arbeiten verwendeten Desktop-PC

der Notebook laufen, es muss daher auf andere Nutzerprozesse Rücksicht nehmen. Die Anwendung soll auch freie Kapazitäten von in einem LAN zur Verfügung stehenden, anderen Rechnern nutzen, muss also in inhomogenen Rechnernetzen arbeiten können. Insbesondere muss auch hier darauf geachtet werden, dass auch diese Rechner möglicherweise nicht exklusiv zur Verfügung stehen, sondern z.B. im Dialogbetrieb oder für andere Zwecke genutzt werden. Die Methode muss deswegen auf innere und äußere Störungen geeignet reagieren können und beinhaltet daher ein dynamisches und adaptives Lastausgleichsverfahren.

3.1. Idee des Verfahrens

Die Grundlagen des Verfahrens werden zunächst anhand eines abstrakten Modells beschrieben und in späteren Kapiteln auf konkrete Simulationsmodelle übertragen. Als Beispiel dient ein Modell, in dessen Rahmen ein Graph von Last verursachenden, transienten Komponenten durchwandert wird. Die transienten Komponenten werden als Token dargestellt, durch deren Wanderung sich die Modellaktivität im Laufe der Simulation verlagert. Es wird dabei davon ausgegangen, dass die in einem Knoten entstehende, dynamische Rechenlast proportional zur Anzahl der von ihm verwalteten Token ist. Zusätzlich kommt noch eine gewisse Grundlast hinzu, die auch ein leerer Knoten erzeugt. Zur besseren Übersicht wird in den folgenden Beispielen ein planarer Graph verwendet. Planarität ist keine Voraussetzung für die Anwendung des beschriebenen Verfahrens, viele reale Systeme sind allerdings als (fast) planarer Graph abbildbar.

Das im Folgenden beschriebene Verfahren arbeitet auf drei Abstraktionsebenen, die als übereinander liegende Graphen angesehen werden können:

- Das physische Rechnernetz bildet einen ungerichteten Graphen $G_R(P, Q)$ mit der Knotenmenge P der Prozessoren oder Prozessorkerne (p_1 bis p_k) und der Kantenmenge Q der Kommunikationskanäle (q_1 bis q_l) zwischen diesen Prozessoren. Schleifen und Mehrfachkanten sind dabei ausgeschlossen. Da die Anwendungen in erster Linie auf Notebooks oder Desktop-PCs ausgeführt werden sollen, wird von einer sternförmigen Struktur des Rechnernetzes ausgegangen: In modernen Mehrkernprozessoren sind die Kerne durch einen gemeinsamen Cache verbunden. LANs bestehen üblicherweise aus ebenfalls sternförmig durch einen Switch verbundenen PCs, können aber auch eine Baumstruktur mit mehreren Switches als innere Knoten haben. Hier werden dann alle vorhandenen Switches gemeinsam als eine einzelne Black Box angesehen, so dass sich wieder eine sternförmige Verbindung der Rechenknoten ergibt. Verfügt ein Prozessor über mehrere Kerne, so werden diese

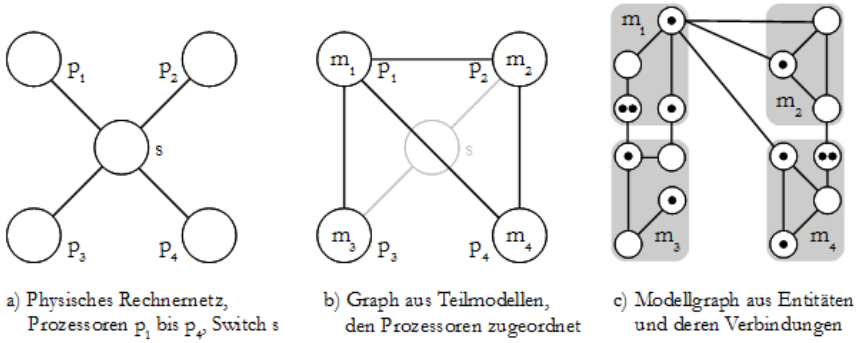


Abbildung 3.1.: Drei Betrachtungsebenen

einzeln betrachtet. In diesem Sinne wird ein Vierkernprozessor als ein Graph aus den Prozessoren p_1 bis p_4 und einem zentralen Switch angesehen (siehe Abbildung 3.1, (a)).

- Auf jedem an der Simulation beteiligten Prozessor läuft zu jedem Zeitpunkt genau ein Simulatorprozess ab, daher werden die Begriffe synonym genutzt. Jedem der Prozessoren p_i ist ein Teilmodell m_i eindeutig zugeordnet, so dass auch diese beiden Begriffe synonym verwendet werden. Jedes Teilmodell kann im Laufe der Simulation mit einer vor dem Simulationslauf statisch festgelegten Menge andere Teilmodelle kommunizieren. Die den Teilmodellen zugeordneten Prozessoren bilden so wieder einen Graph $G_T(P, U)$, mit den Prozessoren P als Knotenmenge und der Menge U der Kanten, über die die Kommunikation zwischen den Teilmodellen abläuft (siehe Abbildung 3.1, (b)).

- Auf diesem Graphen aus Simulatoren und ihren Verbindungen liegt als oberste Ebene das Simulationsmodell. Das Modell besteht aus Modellknoten, die die residenten Entitäten abbilden, und deren Verbindungen untereinander, über die die transienten Entitäten (hier als Token dargestellt) wandern. So ergibt sich ein dritter Graph, der eigentliche Modellgraph $G_M(V, E)$ mit den Modellknoten $v \in V$ und den diese verbindenden ungerichteten Kanten $e \in E$. Auch dieser Graph ist schleifenfrei und beinhaltet keine Mehrfachkanten (siehe Abbildung 3.1, (c)). Die Teilmenge der Knoten $v \in V$, die einem Prozessor $p \in P$ zugeordnet sind, wird im Folgenden mit V_p bezeichnet.

Der Ablauf des Verfahrens soll nun kurz beschrieben werden, in den Abschnitten 3.2.1 bis 3.2.4 wird auf die Details der Umsetzung eingegangen.

Im Laufe des Simulationsschritts wird die von den Knoten und Token erzeugte Last durch die Prozessoren in P verarbeitet. Die Berechnungsroutinen sind dabei den Knoten zugeordnet, sie werden jeweils über den Fortschritt der Simulationszeit oder den Eingang einer Nachricht in Kenntnis gesetzt. Die Knoten sind außerdem für die Verwaltung der Transienten zuständig. Ist ein Prozessor mit der Berechnung des Simulationsschritts fertig, betritt er wie in Abschnitt 2.2.2.2 beschrieben eine Synchronisierungsbarriere. Im Rahmen dieser Barriere wird der dynamische Lastausgleich durchgeführt.

Vor Beginn des Simulationslaufs wird das Modell im Rahmen eines statischen Lastausgleichs partitioniert und die entstehenden Teilmodelle auf die Prozessoren $p \in P$ verteilt (siehe Abbildung 3.2). Durch Anwendung einer Heuristik wird dabei darauf geachtet, dass die Zahl der Kanten zwischen den Partitionen möglichst gering ist, so dass während des Simulationslaufs so wenig wie möglich Kommunikation zwischen den Prozessoren erforderlich ist.

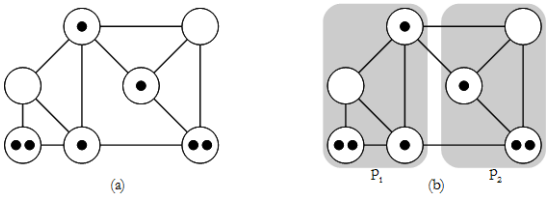


Abbildung 3.2.: Modell vor (a) und nach (b) Partitionierung und Zuordnung zu den Prozessoren p_1 und p_2

Im Laufe der Simulation wandern die Token entlang der Kanten von Knoten zu Knoten (siehe Abbildung 3.3). So entstehen Lastasymmetrien, die dynamisch ausgeglichen werden müssen. Im Rahmen des dynamischen Lastausgleichs prüft deshalb jeder Prozessor p_s zuerst, ob er voll ausgelastet ist. Falls dies zutrifft und p_s benachbart ist zu einem unausgelasteten Prozessor p_f, werden Modellknoten aus V_{p_s}, die Kanten zu Knoten in V_{p_f} haben, von Prozessor p_s zu Prozessor p_f verschoben (siehe Abbildung 3.4). Dieser Ausgleich erfolgt iterativ über mehrere Simulationsschritte hinweg, bis ein stabiler Zustand erreicht ist. Dabei wird ein dynamisches Lastmaß verwendet, das auch die von den Prozessoren zur Verfügung gestellte Leistung beachtet. Das Lastausgleichsverfahren ist somit adaptiv.

Die Methode nutzt Wissen über regionale Abhängigkeiten im Modell aus, indem sie bei der Verlagerung von Last Nachbarschaftsrelationen der Modellentitäten beachtet. So findet auch nach der Verlagerung ein großer Teil der Kommunikation zwischen den Entitäten lokal statt. Wird die regionale Struktur nicht beachtet, sondern die Entitäten auf einen beliebigen, nicht benachbarten Prozessor verschoben, kann in der Folge beim

Nachrichtenaustausch zwischen den Entitäten deutlich mehr Kommunikation zwischen den Prozessoren nötig sein.

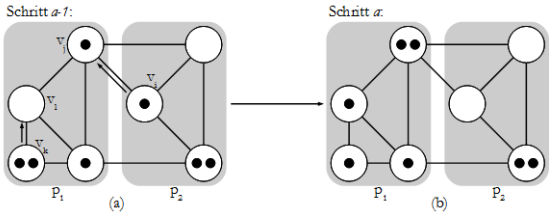


Abbildung 3.3.: Lastverlagerung während des Simulationslaufs

Im Laufe mehrerer Berechnungsschritte kann durch Agglomeration von Token in einem Knoten v_i eine Situation eintreten, in der ein Lastausgleich durch Verlagern von Knoten allein nicht möglich ist (siehe Abbildung 3.5, (a)): Selbst wenn ein Prozessor ausschließlich zur Berechnung von v_i genutzt würde, hätte er gegenüber anderen Prozessoren eine Überlast an Token. Um hier einen erfolgreichen Ausgleich durchführen zu können, soll die Last feingranularer werden: Dazu wird Knoten v_i in zwei Knoten v_i' und v_j aufgeteilt und die Last auf die beiden neuen Knoten verteilt. Dabei muss der Graph lokal neu konstruiert werden, die Verfahrensweise wird dabei vom Modell bestimmt: Falls der Knoten in zwei sequentielle Knoten aufgeteilt werden soll, wie z.B. bei der Repräsentation eines Gleisabschnitts (siehe Abbildung 3.6, (a)), müssen die von v_i ausgehenden Kanten gemäß der Modellvorgaben auf die beiden Knoten v_i' und v_j aufgeteilt werden; diese müssen zusätzlich durch eine Kante verbunden werden (siehe Abbildung 3.5, (b)). Falls der Knoten in zwei parallel liegende Knoten aufgeteilt werden soll, wie z.B. bei der Modellierung von Bedienstationen in einem prozessorientierten Modell (siehe Abbildung 3.6, (b)), müssen die Knoten v_i' und v_j jeweils mit allen Nachbarn von v_i verbunden werden (siehe Abbildung 3.5, (c)). Nach der Teilung sind Knoten v_i' und v_j zuerst demselben Prozessor zugeordnet, im folgenden Berechnungsschritt kann dann der Lastausgleich wie oben beschrieben erfolgen.

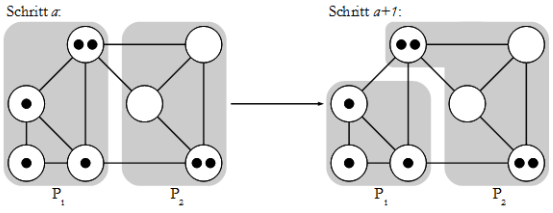


Abbildung 3.4.: Dynamischer und adaptiver Lastausgleich

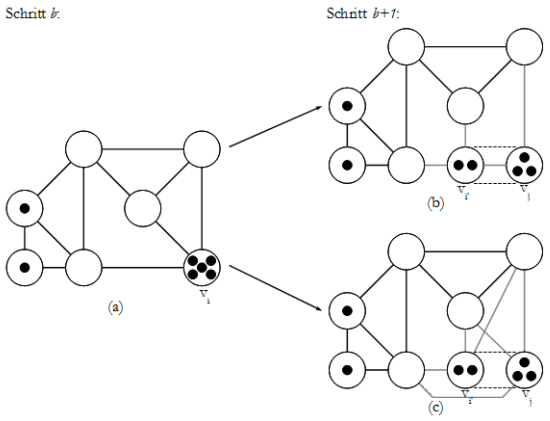


Abbildung 3.5.: Trennen von Knoten bei Überlast

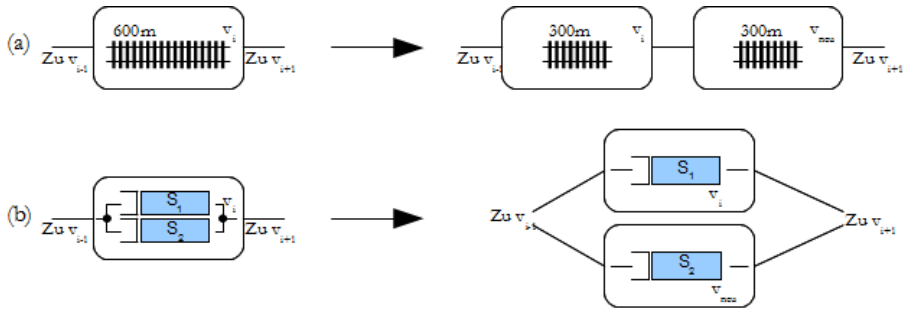


Abbildung 3.6.: Trennung von Knoten

Analog zur Aufteilung von Knoten bei Überlast können zwei Knoten v_i und v_j bei Unterlast zu einem einzelnen Knoten v_i' verschmolzen werden (siehe Abbildung 3.7). Die von v_i und v_j ausgehenden Kanten werden dazu umgehängt an Knoten v_i', die v_i und v_j verbindende Kante gelöscht.

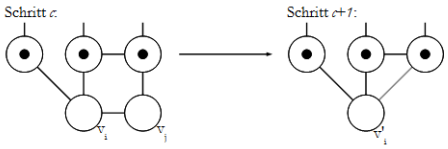


Abbildung 3.7.: Verschmelzen von Knoten bei Unterlast

Ein willkommener Nebeneffekt des Teiles und Verschmelzens von Knoten bei hoher resp. niedriger Belastung ist, dass Modellbereiche mit hoher Aktivität, die in der Regel ausführlicher analysiert werden sollen, in einer höheren Auflösung betrachtet werden können als Bereiche mit geringer Aktivität. Die zur Verfügung stehende Rechenleistung

kann so in Hinsicht auf die erzielte Aussagekraft der Simulationsergebnisse bestmöglich genutzt werden.

Die beschriebene Architektur ermöglicht, neben Modellen mit fixem Zeitfortschritt auch ereignisbasierte Modelle zu implementieren. Das Erstellen von prozessbasierten oder agentenbasierten Anwendungen ist ebenfalls möglich. Im nächsten Abschnitt soll der skizzierte Ansatz detaillierter betrachtet und Konzepte zur Umsetzung als Software-Framework vorgestellt werden.

3.2. Umsetzung

Die Methode ist sowohl mit Hilfe von konservativen als auch mittels optimistischer Verfahren implementierbar, da die Parallelisierungsmechanismen aus Sicht der Anwendung transparent bleiben. Für die Demonstration der Mechanismen wird von einem konservativen Parallelisierungsverfahren mit fixem Zeitfortschritt ausgegangen, das mit synchroner Ausführung arbeitet.

Insbesondere die Koordination der Prozessoren durch einen einzelnen Controller bedeutet dabei eine gewisse Begrenzung hinsichtlich der Skalierfähigkeit, da der Aufwand für die Koordination linear wächst (siehe Abschnitt 3.3). Für die Zielplattformen des Projekts CATS, also Desktop-PCs und Notebooks mit Mehrkernprozessoren, ist dieses Vorgehen sicherlich angemessen. Soll die Methode jedoch auf massiv parallelen Rechnern eingesetzt werden, muss dieser Flaschenhals beseitigt werden. Hierzu kann z. B. das in Abschnitt 2.2.1.1 beschriebene Schmetterlingsverfahren eingesetzt werden, mit dem sich die Komplexität der Koordination auf ein logarithmisches Wachstum reduzieren lässt. Ebenfalls denkbar sind (optimistische) Verfahren, bei denen sich die jeweils benachbarten Prozessoren koordinieren, und die so ganz ohne zentralen Controller auskommen. Da die Implementierung des Simulationsmodells (beschrieben in Kapitel 4) von den Mechanismen der Parallelisierung isoliert ist, ist ein späterer Umbau ohne Anpassungen des Modells möglich.

3.2.1. Ablauf der Simulation

Ein einzelner Prozessor p_1 ist als Controller ausgezeichnet, er organisiert den sequentiellen Teil der Simulation. Insbesondere liest er vor Beginn der Simulationsläufe das Simulationsmodell ein und verteilt anschließend die Knoten initial an die teilnehmenden Prozessoren (siehe Abschnitt 3.2.2). Zu Beginn der Simulation wird die aktuelle Simulationszeit auf die vorgegebene Startzeit gesetzt. Dann beginnt die eigentliche Simulationsschleife (siehe Algorithmus 3.1).

Algorithmus 3.1 Simulationsschleife

```
// --- Simulationslauf ausführen
fertig = falsch;
simulationszeit = startzeit;
Solange(nicht fertig und simulationszeit<=maxzeit) {
    Synchronisiere;
    fertig = wahr;
    Für jeden Knoten v(i) aus V(p) {
        Falls (v(i) nicht im Pausenmodus) Dann {
            Berechne v(i);
            fertig = falsch;
        }
    }
    Permutiere V(p);
    Erhöhe simulationszeit;
    Knoten verschmelzen;
    Knoten teilen;
}
```

Jede Iteration der Schleife beginnt mit der Synchronisation der beteiligten Prozessoren durch eine Synchronisierungsbarriere (wie in Abschnitt 2.2.1.1 beschrieben). So wird erreicht, dass alle Prozessoren zu einem gemeinsamen Zeitpunkt die Berechnung eines Simulationsschritts i abgeschlossen haben, bevor einer der Prozessoren mit der Berechnung des Schritts $i+1$ beginnt. Dies wird zur Durchführung des dynamischen und adaptiven Lastausgleichs (siehe Abschnitt 3.2.3) und dem Verschieben der Token genutzt (siehe Algorithmus 3.2).

Algorithmus 3.2 Synchronisierung der Prozessoren

```
Versende Token an andere Prozessoren;
Führe Synchronisierungbarriere, Teil 1 aus;
Empfange Token von anderen Prozessoren;
Führe Lastmessung durch;
Führe Lastbewertung durch;
Versende zu verlagernde Knoten;
Führe Synchronisierungbarriere, Teil 2 aus;
Empfange und gliedere Knoten ein;
```

Nach der Synchronisierung werden alle Knoten in V_p in einer bestimmten Reihenfolge angesprochen. Falls sich der aktuelle Knoten v_i nicht im Pausenmodus befindet, wird

seine Berechnungsroutine ausgeführt und er so über das Vergehen der Simulationszeit in Kenntnis gesetzt. Im Laufe dieser Modellberechnungen entscheiden die Modellknoten, ob ein Token verschoben werden soll und, falls ja, an welchen Nachbarknoten. Falls der Zielknoten von einem anderen Prozessor verwaltet wird, werden die Token im Rahmen des nächsten Synchronisierungsschritts an diesen gesendet und dort eingegliedert. Falls der Zielknoten vom selben Prozessor verwaltet wird, reicht hier ein Umhängen von Zeigern.

Im Anschluss an diesen Simulationsschritt wird die Reihenfolge des Ansprechens der Knoten permutiert, wodurch wie beschrieben das Entstehen von Artefakten vermieden werden soll. Zuletzt wird der Wert der aktuellen Simulationszeit erhöht. Bei vielen Modellen sind - je nach Lookahead und Ereignisdichte - Simulationsschritte der Größe $\Delta t > 1$ möglich. Der Einfachheit halber wird hier eine fixe Schrittweite von $\Delta t = 1$ angenommen. Die beschriebenen Schritte werden wiederholt, bis die maximale Simulationszeit überschritten ist oder alle Knoten im Pausenmodus sind. Im Anschluss an den eigentlichen Simulationslauf können Statistiken generiert und ausgegeben werden.

3.2.2. Statischer Lastausgleich

Die Modellknoten sollen vor Beginn des Simulationslaufs so verteilt werden, dass die Prozessoren durch die zu erwartende Rechenlast möglichst gleich belastet werden. Als zweites Ziel soll die Kommunikationslast - also Übertragungen von Token und Verlagerung von Knoten zwischen zwei Prozessoren - möglichst gering sein.

Da es vor Beginn eines Laufs noch keine Erfahrungen zu der zu erwartenden Rechenlast gibt, wird für jeden Knoten eine Einheitslast angenommen. Als Indikator für die Kommunikationslast wird die Anzahl der zwischen zwei Partitionen des Graphen bestehenden Kanten herangezogen. Die Knoten sollen daher so verteilt werden, dass die Anzahl der Kanten zwischen den Partitionen möglichst gering ist.

In der Literatur ist das Aufteilen eines Graphen $G(V, E)$ mit n Knoten in k fast gleich große Komponenten mit einer Größe $|V_p| \leq e * \frac{n}{k}$ (mit festem $e \geq 1$) und einer minimalen Anzahl von die Komponenten verbindenden Kanten unter dem Namen Graph Partition bekannt. Graph Partition ist wie von Garey, Johnson und Stockmeyer in [19] gezeigt NP-vollständig, kann also (falls wie vermutet $P \neq NP$ gilt) nicht auf effiziente Weise exakt gelöst werden. Eine exakte Lösung ist für den beschriebenen Anwendungsfall nicht nötig, da im Laufe der Berechnungen Knoten dynamisch zwischen den Prozessoren verschoben werden, eine optimale Aufteilung also schnell wieder zerstört wird.

Eine einfache Heuristik für Graph Partition wird von Kernighan und Lin in [30] beschrieben, die für das entworfene Parallelisierungsverfahren verwendet werden soll. Die Methode arbeitet ausgehend von einer gegebenen Anfangszerlegung, für die sich bei

vielen Verkehrsmodellen eine geographische Zerlegung anbietet. Das iterative Verfahren nimmt dann hinsichtlich der Kommunikationskosten lokale Optimierungen im Stile eines Hillclimber-Algorithmus vor.

Kernighan und Lin beschreiben zuerst ein Verfahren zur Partitionierung eines Graphen in zwei Partitionen A und B. Ausgehend von einer Ausgangsverteilung wird hier für je ein Knotenpaar (v_i, v_j) mit $v_i \in A$, $v_j \in B$ die Änderung der externen Kosten K, also der Anzahl der Kanten zwischen den Partitionen A und B, bei einer Vertauschung von v_i und v_j berechnet. Steht eine Verringerung von K in Aussicht, werden die Knoten tatsächlich vertauscht. Das Verfahren iteriert solange, bis keine weiteren Verbesserungen mehr möglich sind, und damit ein lokales Optimum erreicht ist. Kernighan und Lin geben in [30] für dieses Verfahren eine untere Schranke von $O(n^2)$ an.

Hieraus leiten Kernighan und Lin ein Verfahren zur Aufteilung eines Graphen in $k > 2$ Partitionen her. Dazu werden alle k Partitionen ausgehend von einer initialen Verteilung jeweils paarweise mit dem beschriebenen Verfahren für zwei Partitionen optimiert. In der Regel werden mehrere Iterationen durchgeführt, so dass das Verfahren zur Partitionierung von zwei Partitionen dabei $c * (k-1) * (k-2) = O(k^2)$ mal ausgeführt wird. Die Zeitkomplexität beträgt nun für k Partitionen mit je $\frac{n}{k}$ Knoten $O(k^2 * (\frac{n}{k})^2) = O(n^2)$, sie ist also unabhängig von der Anzahl der eingesetzten Prozessoren.

Kernighan und Lin stellen bei ihren Experimenten fest, dass nach zwei Iterationen ihrer Methode im Mittel 95% des möglichen Gewinns erreicht sind.

3.2.3. Dynamischer Lastausgleich

Bei dem Entwurf des dynamische Lastausgleichsverfahren werden zwei Ziele verfolgt: Das Verfahren muss für eine ihrer Leistung angemessene Auslastung der einzelnen Prozessoren bei einem gleichmäßigen Fortschritts in der Simulationszeit sorgen. Weiterhin soll es durch Ausnutzung der regionalen Abhängigkeiten im Modell im Laufe der Simulation den für die Kommunikation nötigen Aufwand möglichst reduzieren.

Der Lastausgleich wird von den beteiligten Prozessoren jeweils in drei Stufen durchgeführt (siehe auch Abbildung 3.8): Im Rahmen der Lastmessung ermittelt jeder Prozessor p_i seine Auslastung; diese Lastdaten werden im Anschluss zwischen den Prozessoren ausgetauscht. Im Rahmen der Lastbewertung trifft p_i eine Reihe von Entscheidungen: Sollen Modellknoten zu benachbarten Prozessoren migriert werden? Wie viele und welche Modellknoten sollen zu welchem Zielprozessor p_j migriert werden? Im dritten Schritt wird die Lastverschiebung dann ausgeführt.

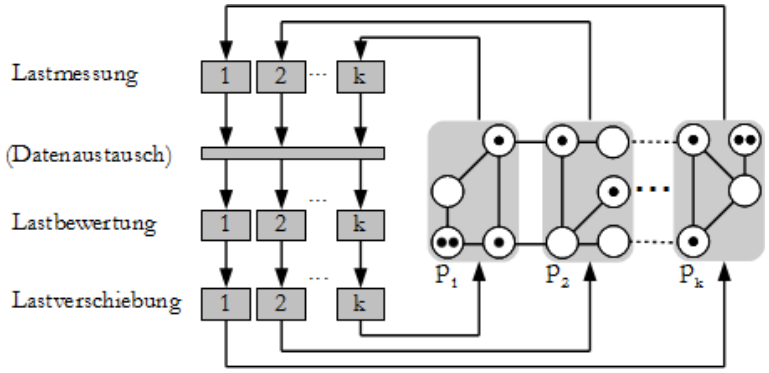


Abbildung 3.8.: Schritte des Lastausgleichsverfahrens

3.2.3.1. Lastmessung

Um bei Über- oder Unterauslastung einzelner Prozessoren gegensteuern zu können, wird in jedem Synchronisierungsschritt i die Auslastung der Prozessoren gemessen. Dies geschieht im Rahmen des Synchronisierungsschrittes zu einem Zeitpunkt $t(i)$, an dem jeder Prozessor $p \in P$ seine Berechnungen für Schritt i beendet hat.

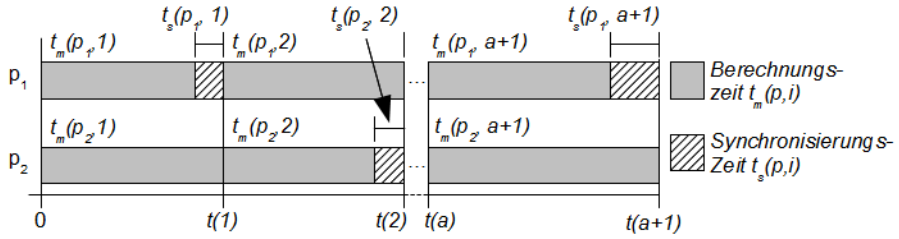


Abbildung 3.9.: Lastmessung in einem System mit zwei Prozessoren

Nun misst jeder Prozessor p die für die Berechnungen von Schritt i nötige modellabhängige Berechnungszeit $t_m(p,i)$ und die vom Ende seiner Berechnungen bis zum Zeitpunkt $t(i)$ vergehende Synchronisierungszeit $t_s(p,i)$ (siehe Abbildung 3.9). Als Grad der Auslastung eines Prozessors $p \in P$ zum Simulationsschritt i wird nun $l_p(i) = \frac{t_m(p,i)}{t_m(p,i)+t_s(p,i)}$ bezeichnet, wobei ein $l_p(i)$ nahe eins (also eine Synchronisierungszeit $t_s(p,i)$ nahe null) auf Basis der lokalen Lastdaten des Prozessors p sowohl auf Vollauslastung als auch auf Überlastung hindeuten kann. Um hier unterscheiden zu können, müssen die Lastdaten der anderen Prozessoren aus P betrachtet werden. Darum werden diese Daten im Rahmen des Synchronisierungsprozesses zwischen den beteiligten Prozessoren ausgetauscht. Die ungenutzte Kapazität errechnet sich analog als $f_p(i) = \frac{t_s(p,i)}{t_m(p,i)+t_s(p,i)}$. Die für die Berechnung eines Simulationsschritts nötige Gesamtzeit $t_g(i)$ setzt sich aus der Berechnungszeit,

der Synchronisierungszeit und der für den Lastausgleich und andere Verwaltungsarbeiten nötigen Kommunikationszeit $t_c(p,i)$ zusammen (siehe Formel 3.1). Der Wert von $t_g(i)$ ist über alle Prozessoren gleich.

$$t_g(i) = t_m(p,i) + t_s(p,i) + t_c(p,i) \tag{3.1}$$

Ein Absinken der Synchronisierungszeit eines Prozessors p zwischen den Schritten $i-1$ und i kann durch mehrere Umstände verursacht werden, die jeweils lokal oder global sein können: Lokale Ursachen sind eine erhöhte Aktivität im Teilmodell und dadurch eine Zunahme der zu verarbeitenden Last, oder der steigende Ressourcenbedarf von externen Prozessen und somit ein erhöhter Uhrzeitbedarf für die Berechnung des Simulationsschritts. Eine globale Ursache ist ein sinkender Berechnungszeitbedarf der anderen Prozessoren; sind diese schneller mit den Berechnungen fertig, sinkt u.U. die für den Simulationsschritt nötige Gesamtzeit $t_g(i)$.

Ein Anstieg der Synchronisierungszeit kann ebenfalls lokal oder global begründet sein: Lokale Ursache kann ein Sinken der Modellaktivität sein, so dass der Prozessor p schneller mit der Berechnung fertig ist. Ebenfalls eine mögliche Ursache für eine schnellere Berechnung und damit eine höhere Synchronisierungszeit ist das Freiwerden von Prozessorressourcen, die vorher von externen Prozessen belegt waren. Eine globale Ursache ist ein steigender Zeitbedarf für die Berechnung des Simulationsschritts durch andere Prozessoren; hierdurch steigt die für den Schritt nötige Gesamtzeit $t_g(i)$ an, so dass p länger in der Synchronisierungsbarriere bleibt.

Das auf $t_s(p,i)$ aufbauende Lastmaß beachtet also innere und äußere Störungen, es berücksichtigt sowohl den Fortschritt der Simulationszeit (der hier fix, also bekannt ist) als auch die zur Verfügung stehende Rechenleistung. Auf Basis dieser Lastmessung kann also eine dynamische und adaptive Lastbewertung erfolgen.

3.2.3.2. Lastbewertung

Im Rahmen der Lastbewertung hat jeder Prozessor p zu entscheiden, ob in Schritt i überhaupt ein Lastausgleich durchgeführt werden soll, und falls ja, wie viele und welche Knoten migriert werden sollen.

Das Durchführen des Lastausgleichs kostet Rechenzeit und benötigt Netzwerk-Ressourcen. Um nur bei längerfristigen Lastungleichgewichten aktiv zu werden, wird zur Entscheidung, ob ein Lastausgleich stattfinden soll, ein geglätteter Wert s_i betrachtet (siehe Formel 3.2). Die Wirkung kurzer Ausschläge der Prozessorlast auf die Lastbewertung wird so begrenzt. Die weiteren Schritte des Lastausgleichs werden nur dann ausgeführt,

wenn s_i unterhalb eines Schwellwerts β_i liegt.

$$s_i = \alpha * t_s(p, i) + (1 - \alpha) * s_{i-1} \tag{3.2}$$

Weiterhin sollen Überkompensationen verhindert werden, die durch lange Netzwerklaufzeiten oder durch den Versuch auftreten können, auch sehr kleine Imbalancen auszugleichen. Hier besteht die Gefahr des Thrashings, dass also eine Reihe von Knoten immer wieder zwischen zwei Prozessoren hin und her gesendet werden. Dies wirkt sich naheliegenderweise nicht positiv auf die Performanz aus. Der Wert β_i ist daher nicht für die Dauer der Simulation konstant, sondern ändert sich ausgehend von einem festgelegten Startwert β_0 in durch β_{min} und β_{max} vorgegebenen Grenzen: Wurden in den letzten Simulationsschritten oft Lastverschiebungen durchgeführt, wird dies als mögliches Anzeichen für Thrashing gesehen. Wird nun in Schritt i auch ein Lastausgleich ausgeführt, so wird der Schwellwert gesenkt: $\beta_{i+1} = \max(\beta_i/\gamma, \beta_{min})$, mit $\gamma \geq 1$. Weitere Lastverschiebungen werden also nur ausgeführt, wenn Prozessor p sehr stark ausgelastet resp. überlastet ist. Wird kein Lastausgleich durchgeführt, wird davon ausgegangen, dass die Schwelle wieder gelockert werden kann. In diesem Fall wird der Wert iterativ wieder erhöht: $\beta_{i+1} = \min(\beta_i * \gamma, \beta_{max})$

Migriert wird jeweils ein vorgegebener Anteil φ der vom Prozessor p verwalteten Knoten, mindestens jedoch ein Knoten: Die Anzahl zu verlagernder Knoten δ wird deshalb wie folgt bestimmt: $\delta = \max(1, \lfloor |V_p| * \varphi \rfloor)$.

Um zu bestimmen, welche Knoten von einem Prozessor p_s migriert werden sollen, werden alle Knoten des Teilgraphen V_{p_s} anhand der folgenden Kriterien in vier Prioritätsstufen eingeteilt:

- *Priorität 4* erhält jeder Knoten v_i aus V_{p_s}.
- *Priorität 3* erhält jeder Knoten v_i aus *Priorität 4*, der mindestens eine Kante zu einem Knoten v_j besitzt, der von einem beliebigen anderen Prozessor $p(v_j) \neq p_s$ verwaltet wird.
- *Priorität 2* erhält jeder Knoten v_i aus *Priorität 3*, der mindestens eine Kante zu einem Knoten v_j besitzt, der von einem zum aktuellen Zeitpunkt nicht ausgelasteten Prozessor $p(v_j) \neq p_s$ verwaltet wird. Dieser Prozessor $p_f(v_i) := p(v_j)$ wird als Ziel einer möglichen Migration festgehalten.
- Bevorzugt verschoben werden von den Knoten der *Priorität 2* jene Knoten v_i, bei denen die Anzahl von Kanten (v_i, v_j) zu Knoten v_j mit $p(v_j) = p_f(v_i)$ größer ist

als die Anzahl der Kanten zu Knoten v_k mit $p_s = p(v_k)$. Diese Knoten erhalten die *Priorität 1*.

Durch das Verschieben von Knoten v_i der *Priorität 1* zum Prozessor $p_f(v_i)$ sinkt die Anzahl der Kanten zwischen den Modellpartitionen. Durch das Lastausgleichsverfahren wird also nicht nur unter Beachtung des Fortschritts in der Simulationszeit die Rechenlast gleichmäßig verteilt, sondern ggf. auch die erwartete Kommunikationslast gesenkt.

Werden über die Knoten der *Priorität 1* hinaus noch Kandidaten benötigt, wird auf die bislang unberücksichtigten Knoten der *Priorität 2* zurück gegriffen und schließlich auch auf unberücksichtigte Knoten der *Priorität 3*. Knoten, die allein *Priorität 4* zugeordnet sind, werden nicht verschoben.

Verschiebungen von einem überlasteten Prozessor p_s zu einem anderen ausgelasteten oder überlasteten Prozessor p'_s können mittelbar hilfreich sein, falls p_s keinen benachbarten unausgelasteten Prozessor hat, p'_s jedoch schon. Da in einem geschlossenen System niemals alle Prozessoren zugleich überlastet sein können, kann so in mehreren Schritten ein Lastausgleich erreicht werden.

3.2.3.3. Lastverschiebung

Da die Lastverschiebung ebenfalls im Rahmen des Synchronisierungsschritts stattfindet, können zu diesem Zeitpunkt ohne Rücksicht auf laufende Simulationsberechnungen Veränderungen am Modellgraphen vorgenommen werden. Jeder Prozessor p_s kodiert jeden zu verlagernden Modellknoten v als Nachricht N_v, sendet diese an den entsprechenden Zielprozessor p_f und setzt dann ggf. dritte Prozessoren, die auch über Knoten mit Kanten zu v verfügen, von dessen Verlagerung in Kenntnis.

Danach betritt der Prozessor p_s eine zweite Synchronisierungsbarriere. Durch diese Barriere wird sicher gestellt, dass alle Prozessoren alle zu migrierenden Knoten an ihre jeweiligen Ziele gesendet haben, bevor mit der Eingliederung der empfangenen Knoten begonnen wird. Dazu wird nach dem Verlassen der Barriere jede empfangene Nachricht N_v dekodiert und in einen neuen Knoten v umgewandelt, der in das Teilmodell von p_f eingegliedert wird.

3.2.4. Aufteilen und Verschmelzen von Knoten

Wie in Abschnitt 3.1 beschrieben, sollen Knoten mit besonders hoher Last aufgeteilt, und Knoten mit geringer Last zusammen gelegt werden.

Um Kandidaten für das Aufteilen von Knoten zu identifizieren, werden für jeden Knoten v_i zwei Bedingungen geprüft: Die von einem Knoten verursachte Rechenlast $L(v_i)$

muss deutlich höher sein als die Durchschnittslast (also $L(v_i) > \frac{t_m(p,i)}{|V_p|} * e_h$ mit $e_h \geq 1$), und der Knoten v_i muss sich selbst als teilbar beschreiben.

Ist ein geeigneter Kandidat v_i gefunden, wird zuerst auf Prozessor $p(v_i)$ ein neuer Knoten v_j gebildet, in den dann ein Teil des von v_i dargestellten Modellteils übertragen wird. Verwaltet v_i vor dem Aufteilen in einem Verkehrsmodell beispielsweise eine Schienenstrecke der Länge 600 Meter mit den darauf enthaltenen Verkehrssignalen und den überfahrenden Bahnfahrzeugen, so wird die Hälfte der Strecke an v_j übergeben, so dass jeder Knoten jeweils für 300 Meter Strecke und die dazu gehörenden Entitäten zuständig sind. Zuletzt müssen noch je nach Modell die zu v_i inzidenten Kanten entweder zwischen v_i und v_j aufgeteilt oder dupliziert werden (siehe nochmals Abbildung 3.5). Die beiden Möglichkeiten werden wie bereits beschrieben in Abbildung 3.6 verdeutlicht: Bei der Trennung serieller Teilmodelle (wie in Abbildung 3.6 (a) am Beispiel von Schienenstrecken gezeigt) werden die Kanten unter den neuen Knoten aufgeteilt, bei der Trennung von parallelen Teilmodellen (wie in den in Abbildung 3.6 (b) gezeigten Bedienstationen in einem prozessorientierten Modell) müssen die Kanten dupliziert werden.

Beim Verschmelzen von Knoten wird analog vorgegangen: Für jedes benachbarte Knotenpaar v_i und v_j mit $(v_i, v_j) \in E$ und $p(v_i) = p(v_j)$ wird geprüft, ob die Rechenlast beider Knoten deutlich unter dem Durchschnitt liegt (also $L(v_i) < \frac{t_m(p,i)}{|V_p|} * e_l$ und $L(v_j) < \frac{t_m(p,i)}{|V_p|} * e_l$ mit $e_l \leq 1$) und ob sich beide Knoten v_i und v_j als verschmelzbar beschreiben.

Sind Kandidaten identifiziert, werden die von den ausgewählten Knoten v_i und v_j repräsentierten Modellteile verschmolzen zu v_i'; die beiden Ursprungsknoten inklusive der sie verbindenden Kante (v_i, v_j) werden aus dem Graphen gelöscht. Dann werden alle zu v_i und v_j inzidenten Kanten an v_i' umgehängt (siehe nochmals Abbildung 3.7).

Das beschriebene Verfahren bietet wie bereits beschrieben über die reine Laufzeitverbesserung hinaus noch einen weiteren Vorteil: Bereiche mit hoher dynamischer Last sind für die Modellanalyse oft besonders interessant, während geringe Last häufig in weniger wichtigen Modellbereichen auftritt. Durch den Mechanismus des lastabhängigen Teilen und Verschmelzens findet automatisch eine besonders detaillierte Betrachtung wichtiger Bereichen des Modells statt, während uninteressante Bereiche nur grob betrachtet werden und so Rechenzeit frei machen.

3.3. Überlegungen zu Skalierbarkeit und Effizienz

Im Folgenden sollen einige Überlegungen angestellt werden, in welchem Rahmen vom beschriebenen Verfahren Laufzeitverbesserungen zu erwarten sind. Da dies offensichtlich

stark vom implementierten Modell abhängt, müssen dafür eine Reihe von Annahmen getroffen werden.

Dazu gehört die Annahme, dass das Verfahren von einem homogenen Parallelrechner ausgeführt wird, dessen Netzwerk und Prozessoren unbelastet durch externe Prozesse sind. Das Netzwerk hat zudem eine beliebig hohe Kapazität, so dass der für die Kommunikation betriebene Aufwand allein von der Zeitkomplexität abhängt. Von der Last wird angenommen, dass sie beliebig teilbar ist, also perfekt ausgeglichen werden kann. Zur Lastdynamik werden in Einzelfällen unterschiedliche Annahmen gemacht. Bei der Betrachtung wird zudem angenommen, dass das Modell linear mit der Anzahl eingesetzter Prozessoren wächst.

Betrachtet wird die Zeitkomplexität der eigentlichen Simulationsläufe, also des Ausführens der einzelnen Simulationsschritte, mit steigender Anzahl eingesetzter Prozessoren $k = |P| \geq 2$. Vor den Läufen liegende Arbeiten wie das Einlesen der Modellknoten aus einer Datenbank sind je nach eingebundenem Modell verschieden, für eine Partitionierung nach dem Kernighan/Lin-Verfahren fällt wie in Abschnitt 3.2.2 beschrieben die von k unabhängige Rechenkomplexität von $O(k^2 * (\frac{n}{k})^2) = O(n^2)$ (mit n als Anzahl der Knoten im Modell) an.

Ein einzelner Simulationsschritt i lässt sich für jeden Prozessor $p \in P$ in drei Phasen unterteilen (siehe Abbildung 3.10):

1. Berechnungszeit $t_m(p, i)$: Die eigentliche Berechnung des Simulationsschritts, abhängig vom Modell.

2. Synchronisierungszeit $t_s(p, i)$: Prozessor p wartet darauf, dass alle Prozessoren aus P die Arbeit am aktuellen Simulationsschritt abgeschlossen haben.

3. Kommunikationszeit $t_c(p, i)$: Der Lastausgleich wird durchgeführt, empfangene Transienten werden in das Teilmodell eingegliedert und andere Verwaltungsaufgaben verrichtet.

3.3.1. Komplexität der Berechnungs- und Synchronisierungsphasen

Die Synchronisierungszeit beginnt für jeden Prozessor mit dem Senden einer *barrier*-Nachricht nach dem Abschluss der Berechnungen und endet mit dem Empfang der *release*-Nachricht. Die beiden ersten Phasen $t_m(p, i)$ und $t_s(p, i)$ können daher gemeinsam betrachtet werden: Nachdem die Modellberechnung stattgefunden hat, wartet p darauf,

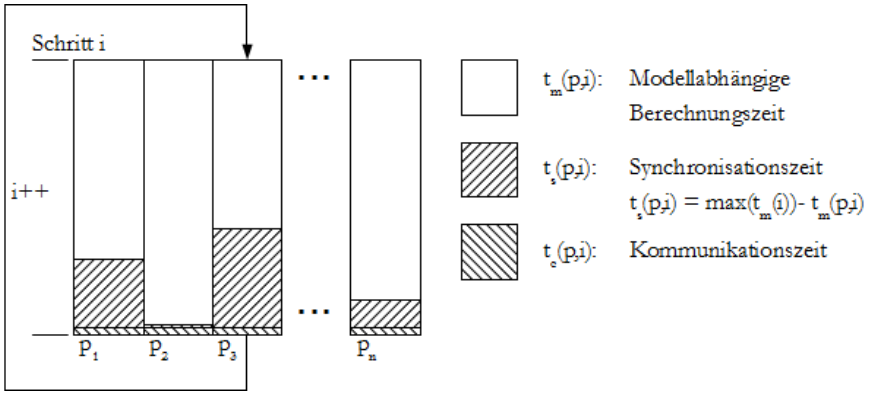


Abbildung 3.10.: Phasen der Simulationsschritte

dass die anderen Prozessoren die Berechnung ebenfalls beenden. Es gilt also:

$$t_m(p,i) + t_s(p,i) = \max_{p' \in P}(t_m(p',i)) =: t_m(i)$$

Die mit wachsendem k auftretende, gemeinsame Zeitkomplexität der beiden Phasen entspricht also dem Wachstum des Erwartungswerts von $t_m(i)$ bei k eingesetzten Prozessoren. Die einzelnen Werte von $t_m(p,i)$ sind modellabhängig und können in ihrer Höhe nicht voraus gesagt werden. Es können jedoch einige Aussagen getroffen werden, wie sich die Einzelwerte verhalten.

Im Folgenden sollen zwei Fälle genauer betrachtet werden, die jeweils einem einzelnen Simulationsschritt i aus dem Lauf entsprechen: Im ersten Fall wird davon ausgegangen, dass der dynamische Lastausgleich trotz hoher Modelldynamik nicht greift oder erst gar nicht durchgeführt wird, und die Berechnungszeiten daher im Laufe der Simulation willkürlich und voneinander unabhängig auseinander driften. Der zweite Fall geht von einem optimal greifenden dynamischem Lastausgleich aus und nimmt an, dass die Last so fein granular ist, dass sie vollkommen ausgeglichen ist.

3.3.1.1. Verhalten ohne Lastausgleich

Falls kein Lastausgleich durchgeführt wird, kann Prozessor p in einem Extremfall die komplette Last tragen, die Rechenzeit entspricht dann der sequentiellen Rechenzeit. Im anderen Extremfall verwaltet p gar keine Last, diese wird komplett von den anderen Prozessoren berechnet. Durch das Einführen einer modellabhängigen Konstante c_m können die Werte normiert werden mit 0 entsprechend dem leeren Teilmodell und 1 der Berechnung des kompletten Modells.

Die Werte $t_m(p,i)$ für alle $p \in P$ können als k unabhängige Ziehungen aus dem gleich verteilten Intervall $(0, c_m)$ der reellen Zahlen angenommen werden. Typischerweise sollte die Verteilung durch das stetige Verlagern der Last auch ohne den Einfluss des Lastausgleichs eher einer Normalverteilung entsprechen. Die Annahme einer Gleichverteilung entspricht hier dem Fall, dass sich die Lasten durch eine ungünstige Dynamik willkürlich entwickeln.

Gesucht ist nun der Erwartungswert E des Maximums von k Ziehungen aus $(0,1)$, abhängig von k. Hierzu wird eine Folge X_1 bis X_k von gleich verteilten und unabhängigen Zufallsvariablen mit Werten aus $(0,1)$ definiert. Bei einer gleich verteilten, unabhängigen Folge entspricht die Wahrscheinlichkeit, dass ein gezogener Wert kleiner oder gleich x ist, $Prob(X \leq x) = x$.

Für die Wahrscheinlichkeit $Prob(\max\{X_{1,}, ..., X_k\} \leq x)$, dass das Maximum von k Ziehungen kleiner oder gleich einem Wert x ist, ergibt sich:

$$\begin{aligned} Prob(\max\{X_{1,}, ..., X_k\} \leq x) &= Prob(X_1 \leq x) * ... * Prob(X_k \leq x) \\ &= x * ... * x = x^k = F(x) \end{aligned} \tag{3.3}$$

Um den Erwartungswert E der Zufallsziehungen $X = \max\{X_1, ..., X_k\}$ mit Werten $x \in (0,1)$ zu erhalten, wird nun das Integral über die Werte von x bezüglich des durch die Verteilungsfunktion $F(x)$ gegebenen Maßes gebildet:

$$E(X) = \int_0^1 x \, dF(x) \tag{3.4}$$

Durch Ableitung von $F(x)$ nach x erhält man:

$$\frac{dF(x)}{dx} = k * x^{k-1} \Rightarrow dF(x) = k * x^{k-1} dx \tag{3.5}$$

Dies kann jetzt in die in Formel 3.4 gezeigte Berechnung des Erwartungswertes eingesetzt werden:

$$E(X) = \int_0^1 x * (k * x^{k-1}) dx = k * \int_0^1 x^k dx \tag{3.6}$$

Durch weiteres Ausrechnen ergibt sich:

$$E(X) = k * \left[\frac{1}{k+1} x^{k+1}\right]_0^1 = 1 * k * \frac{1}{k+1} - 0 = \frac{k}{k+1} = 1 - \frac{1}{k+1} \tag{3.7}$$

Der Erwartungswert des Maximums aus k Ziehungen $E(X)$ entspricht also $(1 - \frac{1}{k+1})$,

er konvergiert bei wachsendem k gegen 1. Damit ergibt sich für den Erwartungswert $E(t_m(i))$:

$$E(t_m(i)) = E(\max_{p \in P}(t_m(p,i))) = c_m * (1 - \frac{1}{k+1}) \tag{3.8}$$

Dies wiederum entspricht den im beschriebenen Fall erwarteten Kosten der Phasen $t_m(p,i)$ und $t_s(p,i)$, die bei wachsendem k auf die Kosten des sequentiellen Falls zugehen.

3.3.1.2. Verhalten mit Lastausgleich

Hier wird wie beschrieben angenommen, dass der Lastausgleich die Last perfekt verteilt, der Wert von $t_m(p,i)$ daher für alle $p \in P$ exakt gleich ist. Da also $t_m(p,i) = t_m(i)$ gilt, gilt mit $t_m(p,i) + t_s(p,i) = t_m(i)$ auch $t_s(p,i) = 0$ für alle p, und zwar unabhängig von der eingesetzten Zahl k der Prozessoren. Die zur Berechnung eines Simulationsschritts nötige Gesamtzeit liegt daher bei $t_g(i) = t_m(i) + t_c(p,i)$.

3.3.2. Komplexität der Kommunikationsphase

Nun müssen noch die von k abhängigen Kosten der dritten Phase $t_c(p,i)$ betrachtet werden. In dieser Phase führt das Verfahren die folgenden sechs Schritte aus:

1. Synchronisierungsbarriere I: Alle Prozessoren haben Berechnung des Simulationsschritts i beendet.
2. Transienten empfangen und eingliedern
3. Lastmessung und -bewertung durchführen, Lastinformationen austauschen
4. Lastverschiebung durchführen: Modellknoten zu benachbarten Prozessoren migrieren
5. Synchronisierungsbarriere II: Alle Prozessoren haben die Schritte 2, 3 und 4 ausgeführt
6. Migrierte Modellknoten empfangen und eingliedern

Die Synchronisierungsbarrieren in den Schritten 1 und 5 klammern die dazwischen liegenden Schritte 2, 3, 4 ein. Wie schon für die Phasen $t_m(p,i)$ und $t_s(p,i)$ ist hier also die Komplexität für den am längsten rechnenden Prozessor gesucht.

Schritt 1 besteht aus zwei Unterschritten: Im ersten Unterschritt a) wartet ein ausgezeichneter Prozessor p_1 auf *barrier*-Nachrichten jedes anderen Prozessors aus P. Die

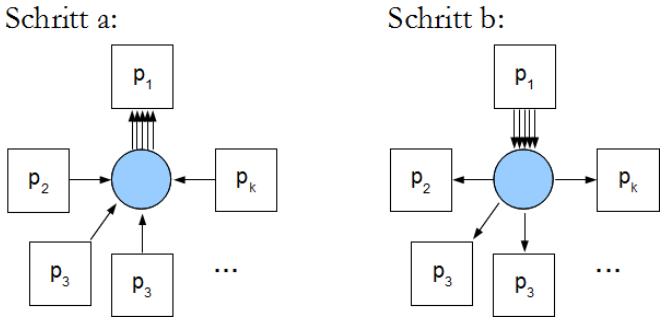


Figure 3.11.: Verlauf der Synchronisierung

Nachrichten werden dabei gepuffert übertragen, so dass das Protokoll auch funktioniert, wenn p_1 als letzter Prozessor die Barriere erreicht. Hat p_1 die Nachrichten aller Prozessoren aus P erhalten, sendet er in Unterschritt b) jedem dieser Prozessoren ein *release*-Signal zum Fortfahren (siehe Abbildung 3.11). Der Schritt ist mittels der MPI-Funktion *MPI_ Barrier()* realisiert. Die konkrete Implementierung der Funktion ist unbekannt, in einem sternförmigem Netz müssen jedoch $2 * (k - 1)$ Nachrichten gesendet werden, so dass sich für diesen Schritt Kosten von $2 * (k - 1)$ ergeben.

In Schritt 2 werden die im Laufe des vorherigen Simulationsschritts von einem nichtlokalen Knoten $v_s \notin V_p$ an einen lokalen Knoten $v_e \in V_p$ versendeten Transienten empfangen und den entsprechenden Objekten zugeordnet. Die Anzahl der insgesamt gesendeten Transienten c_n ist dabei abhängig vom Modell. Unter der Annahme, dass die Wahrscheinlichkeiten, dass ein Modellknoten $v_s \in V$ einem Knoten $v_e \in V$ eine Nachricht sendet, gleich sind, zeigen die Formeln 3.9 und 3.10 die Wahrscheinlichkeiten dafür, dass der empfangende Knoten v_e in V_p ist, resp. der sendende Knoten v_s nicht Teil von V_p ist.

$$Prob(v_e \in V_p) = \frac{|V_p|}{|V|} \tag{3.9}$$

$$Prob(v_s \notin V_p) = \frac{|V| - |V_p|}{|V|} \tag{3.10}$$

Beide Bedingungen sind gemeinsam erfüllt mit der folgenden Wahrscheinlichkeit:

$$Prob(v_e \in V_p, v_s \notin V_p) = \frac{|V_p|}{|V|} * \frac{|V| - |V_p|}{|V|} \tag{3.11}$$

Davon ausgehend, dass $|V_p|$ für alle $p \in P$ gleich ist, ergibt sich bei k Prozessoren

$$Prob(v_e \in V_p, v_s \notin V_p) = \frac{1}{k} * (1 - \frac{1}{k}) = \frac{1 - \frac{1}{k}}{k} = \frac{k-1}{k^2} \tag{3.12}$$

Jeder der Prozessoren muss also in diesem Schritt $c_n * \frac{k-1}{k^2}$ Transienten eingegliedern, wobei c_n abhängig vom zu berechnenden Modell ist. Die von k abhängigen Kosten von Schritt 2 sinken also mit steigender Anzahl der Prozessoren.

Die in Schritt 3 durchgeführten Arbeiten sind zum großen Teil von der Anzahl der von p verwalteten Modellknoten abhängig. Im von k abhängigen Teil wird von jedem Prozessor eine feste Anzahl von Variablenwerten an den Controller p_1 übertragen, der die gesammelten Werte dann an alle anderen Prozessoren weiter reicht. Realisiert wird diese Funktion mittels der MPI-Funktionen *MPI_ Gather()* und *MPI_ BCast()*, die sich in einem sternförmigen Netz analog zur in Schritt 1 verwendeten Funktion *MPI_ Barrier()* verhalten. Es müssen also wieder $2 * (k-1)$ Nachrichten versendet werden.

Schritt 4 wird wie auch die Schritte 2 und 3 geklammert von den Synchronisierungsbarrieren in den Schritten 1 und 5. Daher soll hier die maximale Komplexität betrachtet werden. Die tritt auf, wenn ein einzelner Prozessor $p \in P$ ausgelastet ist, alle anderen Prozessoren jedoch nicht ausgelastet sind. Dann sendet p bis zu c_l Nachrichten an $k-1$ Prozessoren, wobei die Anzahl der Nachrichten c_l wieder durch Eigenschaften des Modells bestimmt ist. Die anderen Prozessoren senden keine Nachrichten, müssen aber an der Barriere in Schritt 5 auf p warten. Die Zeitkomplexität dieses Schrittes liegt also bei $c_l * (k-1)$.

Schritt 5 ist wie Schritt 1 eine Synchronisierungsbarriere mit der Komplexität $2*(k-1)$.

In Schritt 6 muss ein Prozessor p im schlechtesten Fall c_l übertragene Knoten von allen anderen $k-1$ Prozessoren eingliedern. Die Komplexität liegt dann analog zu Schritt 4 bei $c_l * (k-1)$.

Die gesamten Kosten der sechs Schritte der Phase $t_c(p,i)$ werden in Formel 3.13 beschrieben, wobei allerdings die Anzahl c_n der zwischen den Modellknoten verschobenen Transienten und die Anzahl c_l der im schlechtesten Fall zu migrierenden Knoten berücksichtigt werden müssen.

$$\begin{aligned} &\underbrace{2 * (k-1)}_{Schritt\,1} + \underbrace{c_n * \frac{k+1}{k^2}}_{Schritt\,2} + \underbrace{2 * (k-1)}_{Schritt\,3} + \underbrace{c_l * (k-1)}_{Schritt\,4} + \underbrace{2 * (k-1)}_{Schritt\,5} + \underbrace{c_l * (k-1)}_{Schritt\,6} \\ &= 6 * (k-1) + \underbrace{c_n * \frac{k+1}{k^2}}_{Abhängig\ von\ c_n} + \underbrace{2 * c_l * (k-1)}_{Abhängig\ von\ c_l} \end{aligned} \tag{3.13}$$

Die Rechenkomplexität des nur von k abhängigen Teils der Phase $t_c(p,i)$ steigt also mit einem kleinen Faktor linear an. Der von der Zahl c_n der Transienten abhängige Summand sinkt bei steigendem k. Der Wert von c_l wird vom Lastausgleichsverfahren bestimmt. Ist dieses abgeschaltet oder versendet keine Knoten, liegt er bei null.

3.3.3. Gesamtkomplexität eines Simulationsschritts

Bei der Bestimmung der gesamten, von k abhängigen Rechenkomplexität wird wieder zwischen den beiden in Abschnitt 3.3.1 beschriebenen Fällen unterschieden: Im ersten Fall greift der dynamische Lastausgleich trotz sehr ungünstiger Dynamik nicht oder ist abgeschaltet, im zweiten Fall ist die Last optimal zwischen den Prozessoren verteilt.

Im ersten Fall ergibt sich durch Addition der Einzelkomplexitäten der drei Phasen eines Simulationsschritts i die in Formel 3.14 beschriebene Gesamtkomplexität, in die wieder die vom Simulationsmodell abhängige Last c_m und die Zahl c_n der verschobenen Transienten eingehen.

$$t_g(i) = \underbrace{c_m * (1 - \frac{1}{k+1})}_{t_m(p,i)+t_s(p,i)} + \underbrace{6 * (k-1) + c_n * \frac{k+1}{k^2}}_{t_c(p,i)} \tag{3.14}$$

Der bestimmende Faktor ist hier die modellabhängige Gesamtrechenlast c_m, da diese typischerweise sehr groß ist, und $\frac{1}{k+1}$ für steigende k gegen null konvergiert. Der Summand $6*(k-1)$ entspricht der Anzahl zu sendender Nachrichten, macht also im angenommenen homogenen Parallelrechner nur wenig aus. Der dritte Summand $c_n * \frac{k+1}{k^2}$ sinkt bei steigendem k, der Anteil eines Prozessors p an der Gesamtzahl c_n der während des Schritts versendeten Transienten sinkt also bei gleichbleibender Modellgröße. Die Konstante c_l wird als null angenommen, da der Lastausgleich nicht arbeitet.

Im zweiten Fall, also bei optimal verteilter Last, ergibt sich die in Formel 3.15 beschriebene Rechenkomplexität.

$$t_g(i) = \underbrace{\frac{c_m}{k}}_{t_m(i)} + \underbrace{6 * (k-1) + c_n * \frac{k+1}{k^2} + 2 * c_l * (k-1)}_{t_c(p,i)} \tag{3.15}$$

Hier ist wieder der Faktor c_m bestimmend, allerdings wird die Berechnungslast gleichmäßig auf k Prozessoren verteilt. Der Summand $6 * (k-1)$ entspricht wieder der Anzahl zu sendender Nachrichten. Der Summand $c_n * \frac{k+1}{k^2}$ sinkt wie bereits beschrieben mit steigendem k. Der letzte Summand $2 * c_l * (k-1)$ wächst linear mit k, die Zahl c_l der im ungünstigsten Fall zu migrierten Knoten ist von der Dynamik des Modells bestimmt.

Enthält das Modell keine dynamische Lastveränderung, liegt der Wert bei null.

Um die Rechenkomplexität eines Simulationslaufs zu bestimmen, müssen die bestimmten Werte noch mit der ebenfalls modellabhängigen Anzahl c_s der Simulationsschritte multipliziert werden.

3.3.4. Speedup und Skalierbarkeit

Da die Anzahl c_s der Simulationsschritte unabhängig von k ist, wird auch hier wieder nur ein einzelner Schritt betrachtet. Die Synchronisierungsnachrichten müssen unabhängig von den Eigenschaften der Lastverteilung versendet werden, sie treten daher in allen Fällen auf und wachsen linear in k mit einem kleinen Faktor. Bei der asymptotischen Betrachtung der Skalierbarkeit gilt für die zu verteilende Gesamtlast $c_m(k) = c_m(1) * k$, für die Zahl der versendeten Transienten gilt $c_n(k) = c_n(1) * k$.

Im beschriebenen Fall ohne Lastausgleich und mit willkürlicher, hoher Lastdynamik wird die Laufzeit durch die Berechnungs- und Synchronisierungszeit $t_m(i) = c_m(k) * (1 - \frac{1}{k+1})$ bestimmt. Der Faktor $(1 - \frac{1}{k+1})$ konvergiert bei steigendem k auf eins zu, $t_m(i)$ geht daher gegen $c_m(k)$. Die Laufzeit steigt also bei steigendem k auf die Laufzeit des sequentiellen Falls zu. Im beschriebenen Spezialfall skaliert das Verfahren also sublinear.

Bei perfekter Lastverteilung ist die Last mit $t_m(i) = \frac{c_m(k)}{k}$ gleichmäßig verteilt, so dass dieser Summand bei $c_m(k) = k * c_m(1)$ linear mit einem Faktor $1,0$ skaliert. Der von k abhängige Aufwand für den Lastausgleich, in Abschnitt 3.3.2 bestimmt mit $2 * c_l * (k-1)$, steigt ebenfalls linear an. Das Verfahren skaliert also linear, der Skalierungsfaktor hängt bei in erster Linie vom Verhältnis der Rechenlast zur Kommunikationslast $\frac{c_m(1)}{c_l}$ ab, die Anzahl der zu versendenden Token c_n spielt mit steigendem k eine immer kleiner werdende Rolle. Das Verfahren ist also insbesondere verwendbar für große Probleme mit nur mäßiger Dynamik, was ja bereits in den zu Beginn von Kapitel 3 formulierten Ansprüchen an die Eigenschaften geeigneter Simulationsmodelle zum Ausdruck kam.

Im typischen Fall werden bei Modellen mit gutartiger Lastverteilung langsam auseinander driftende Teillasten dynamisch ausgeglichen, so dass die im ersten Fall voraus gesetzten, zwischen null und c_m gleich verteilten Lasten nicht auftreten. Der bei einer konkreten Anwendung tatsächlich realisierte Skalierungsfaktor wird daher in erster Linie vom Verhältnis der Rechenlast zur Kommunikationslast $\frac{c_m(1)}{c_l}$ abhängen, also vom Verhältnis der Problemgröße zum durch die Modellstruktur und die Geschwindigkeit des (realen) Netzwerks nötig werdenden Aufwand für den Lastausgleich. Als grobe Richtschnur lässt sich (analog zur in Abschnitt 2.2.2.2 beschriebenen Methode) festhalten: Je größer das zu berechnende Problem ist, umso geeigneter ist das Verfahren.

3.3.5. Effizienz

Aus der in Abschnitt 3.3.1 berechneten Komplexität der Berechnungs- und Synchronisierungsphasen lassen sich einige Überlegungen zur Effizienz des Verfahrens ableiten. Dabei sollen wieder die bereits beschriebenen Fälle betrachtet werden: Im ersten Fall wird von einer völlig willkürlichen Entwicklung der Prozessorlasten ausgegangen, im zweiten Fall von einer perfekten Verteilung der Last.

Sind die Prozessorlasten gleich verteilt, so liegt der Erwartungswert der gemeinsamen Kosten der Phasen $t_m(p,i)$ und $t_s(p,i)$ für jeden Prozessor $p \in P$ in einem Simulationsschritt i wie bereits beschrieben bei $E(t_m(i)) = (1-\frac{1}{k+1})$, wobei die Werte auf den Bereich von 0 bis 1 normiert sind, und die modellabhängige Konstante c_m nicht berücksichtigt wird.

Der Erwartungswert des Maximums der Berechnungszeiten konvergiert für wachsende k mit $(1-\frac{1}{k+1})$ gegen 1 (siehe Formel 3.16).

$$\lim_{k\to\infty}(1-\frac{1}{k+1}) = 1 \tag{3.16}$$

Da die Berechnungszeiten gemäß Annahme normiert und gleich verteilt sind, liegt der Ex-Post-Erwartungswert jeder einzelnen Stichprobe für einen Prozessor p bei $E(t_m(p,i)) = 0,5 * (1-\frac{1}{k+1})$. Hieraus lässt sich nun der Erwartungswert der Synchronisierungszeit $E(t_s(p,i))$ eines nicht ausgelasteten Prozessors p berechnen (siehe Formel 3.17).

$$\begin{aligned} E(t_s(p,i)) &= (1-\tfrac{1}{k+1}) - E(t_m(p,i)) \\ &= (1-\tfrac{1}{k+1}) - \tfrac{1}{2} * (1-\tfrac{1}{k+1}) \\ &= \tfrac{1}{2} * (1-\tfrac{1}{k+1}) \\ &= \tfrac{1}{2} - \tfrac{1}{2*k+2} \end{aligned} \tag{3.17}$$

Mit

$$\lim_{k\to\infty}(\frac{1}{2} - \frac{1}{2*k+2}) = 0,5 \tag{3.18}$$

gilt für den Erwartungswert der Synchronisierungszeit $E(t_s(p,i))$ für jedes $k \geq 1$ daher offensichtlich $E(t_s(p,i)) \leq 0,5$. Wegen $E(t_m(p,i)) = 0,5 * (1-\frac{1}{k+1})$ gilt weiterhin für jedes $k \geq 1$:

$$E(t_m(p,i)) = E(t_s(p,i)) \tag{3.19}$$

Für jeden einzelnen, nicht ausgelasteten Prozessor $p \in P$ ist also die erwartete Berechnungszeit so groß wie die erwartete Synchronisierungszeit, die erwartete Effizienz liegt in

diesem sehr ungünstigen Fall bei $0,5$.

Die erwartete Summe der gesamten Synchronisierungszeiten aller k Prozessoren $E(t_{idle}(k))$ lässt sich ebenfalls abschätzen. Da ein Prozessor p_i mit $E(t_m(p_i,i)) = (1 - \frac{1}{k+1})$ und $t_s(p_i,i) = 0$ voll ausgelastet ist, liegen diese wegen $E(t_m(p_j,i)) = \frac{1}{2} * (1 - \frac{1}{k+1})$ für alle $p_j \neq p_i$ und $E(t_m(i)) = (1 - \frac{1}{k+1})$ für $k \geq 2$ bei

$$\begin{aligned} E(t_{idle}(k)) &= (k-1) * (E(t_m(i)) - E(t_m(p_j,i))) \\ &= (k-1) * (1 - \tfrac{1}{k+1} - \tfrac{1}{2} * (1 - \tfrac{1}{k+1})) \\ &= (k-1) * (\tfrac{1}{2} * \tfrac{k}{k+1}) \\ &= \tfrac{1}{2} * \tfrac{k^2-k}{k+1} \\ &= \tfrac{1}{2} * \tfrac{(k+1)*(k-1)+1-k}{k+1} \\ &= \tfrac{k-1}{2} - \tfrac{k-1}{2*(k+1)} \end{aligned} \tag{3.20}$$

Der Subtrahend konvergiert gegen $0,5$, also gilt $E(t_{idle}(k)) \leq \frac{k-1}{2}$.

Für die erwartete Gesamtberechnungszeit $E(t_{busy}(k))$ gilt:

$$E(t_{busy}(k)) = \frac{k}{k+1} + (k-1) * (\frac{1}{2} * \frac{k}{k+1}) > E(t_{idle}(k)) \tag{3.21}$$

Daher ist bei der gemeinsamen Betrachtung für k Prozessoren $E(t_{busy}(k)) > E(t_{idle}(k))$, die erwartete Gesamteffizienz des Verfahrens ist also in diesem schlechten Fall größer als $0,5$.

Falls die Last in einem Schritt i perfekt auf die k Prozessoren verteilt ist, sind die Werte von $t_m(p,i)$ für alle $p \in P$ gleich, es gilt wie bereits beschrieben $t_m(p,i) = t_m(i) = 1$ und $t_s(p,i) = t_m(i) - t_m(p,i) = 0$, daher liegt die Effizienz der einzelnen Prozessoren bei $1,0$. Der Erwartungswert der gesamten Berechnungszeit liegt dann bei $E(t_{busy}(k)) = k * t_m(i) = k$, die erwartete Gesamtsynchronisierungszeit bei $E(t_{idle}(k)) = (k-1) * t_s(p,i) = 0$. Die Gesamteffizienz des Verfahrens liegt unter diesen Bedingungen also bei $1,0$.

Der zweite betrachtete Fall ist der Optimalfall. Unter realen Bedingungen braucht der Lastausgleich allerdings einige Simulationsschritte als Reaktionszeit, so dass die Synchronisierungszeiten unter realistischen Annahmen größer null sind (siehe hierzu auch Abschnitt 5.2.3). Da sich das Verfahren jedoch nur an Modelle wendet, deren Lastdynamik gutartig ist, werden die im ersten Fall betrachteten, willkürlich über die Prozessoren verteilten Lasten nicht auftreten. Trotzdem wird auch hier noch eine erwartete Effizienz von mindestens $0,5$ erreicht.

4. Ein Framework zur Parallelisierung von Simulationsanwendungen

In diesem Kapitel wird die Umsetzung des vorgestellten Ansatzes als ein Framework zur Entwicklung von Simulationsanwendungen beschrieben. Wie bereits geschildert war der Wunsch nach einem parallelen Modul für die Simulation von Stadtbahnfahrplänen im Projekt *Computer Aided Tram Scheduling (CATS)* der Ausgangspunkt für die Überlegungen. Dies spiegelt sich im für das Framework gewählten Namen wider: *„cellular engine for cats"*, kurz *cellforce*.

4.1. Vorgehen und verwendete Techniken

Der beschriebene Ansatz wurde als Framework für C++-Projekte realisiert. Die Entwicklung fand unter *Microsoft Windows 7* (64 Bit) statt, für die Implementierung wurden jedoch (abgesehen von der Nachrichtenübertragung) keine über die C++-Standardbibliotheken hinaus gehenden Funktionen genutzt, so dass eine Umsetzung für Unix-Derivate mit minimalen Änderungen möglich ist.

Die nachrichtenbasierte Kommunikation zwischen den Prozessoren wurde mit dem *Message Passing Interface* (MPI, beschrieben z.B. von Gropp, Lusk und Skjellum in [22]), genauer gesagt der Bibliothek MPICH-2 in der Version 1.4.1, realisiert. Zur Entwicklung wurde die Entwicklungsumgebung *Eclipse Galileo* und der C++-Compiler der *Gnu Compiler Collection* in der Version 3.4.5 verwendet. Das Framework umfasst ca. 6.500 Zeilen C++-Code, die darauf aufbauenden Beispielapplikationen nochmals ca. 14.200 Zeilen. Die Quellen der Bibliothek und der verwendeten Beispielprojekte finden sich auf der diesem Band beiliegenden CD-ROM (siehe Anhang C).

4.2. Management der parallelen Simulation

Ein Großteil des Managements der Simulation, insbesondere die Kommunikation zwischen den Prozessoren und der Lastausgleich, ist aus Sicht einer die Bibliothek nutzenden

Anwendung transparent. Die Anwendung nutzt eine relativ schmale API (siehe Abschnitt 4.3) und ist in der sonstigen Ausgestaltung von Modell und Berechnungsweise frei. Die API ist in erster Linie mit Blick auf Modelle mit fixem Zeitinkrement gestaltet, kann aber auch für die Simulation ereignisbasierter Modelle verwendet werden.

4.2.1. Abbildung des Modellgraphen

Die Knoten des Modellgraphen werden als Objekte der Klasse *Cell* abgebildet. Die Kanten des Graphen, hier also die Verbindungen der *Cell*-Objekte untereinander, werden durch Instanzen der Klasse *Port* abgebildet, die von den *Cell*-Objekten verwaltet werden (siehe Abbildung 4.1).

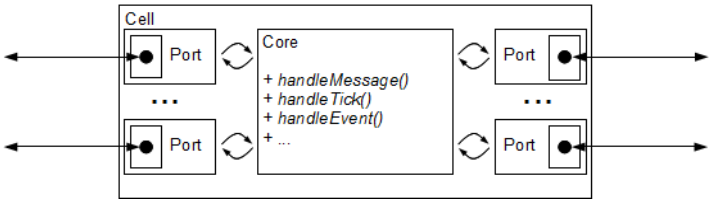


Abbildung 4.1.: Struktur der Klasse *Cell*

Der wichtigste Bestandteil eines *Cell*-Objekts ist die Instanz einer von der abstrakten Klasse *Core* abgeleiteten Klasse. Diese Objekte implementieren eine Reihe von Ereignisbehandlungsroutinen, die aufgerufen werden, wenn die Simulationszeit fortschreitet (Methode *void handleTick(Tick* p_pTick)*[1]), eine Nachricht von einem anderen *Cell*-Objekt eintrifft (Methode *void handleMessage(Port* p_pPort, Message* p_pMessage)*) oder ein terminiertes Ereignis verarbeitet werden soll (Methode *void handleEvent(Event* p_pEvent)*). Zur Realisierung dieser Funktionen bieten die Klassen *Cell* und *Core* eine Reihe von Unterstützungsmethoden. Die Klasse *Cell* übernimmt aus Sicht der Anwendung die gesamte Verwaltung des Simulationsablaufs, der Parallelisierung und des Lastausgleichs, der *Core* selbst übernimmt lediglich die Ausführung des Modellverhaltens. Abschnitt 4.3.2 beschreibt das Ineinandergreifen der Klassen *Cell* und *Core* und die Implementierung des Modellverhaltens näher.

Die in Kapitel 3 als Token bezeichneten Transienten erben von der Klasse *Message*, werden zwischen den *Cell*-Objekten versendet und interagieren mit der Simulationslogik in den von *Core* abgeleiteten Klassen. Aus Sicht der *Core*-Objekte (und damit der Anwendung) ist transparent, ob die Verbindung zu einem anderen Modellknoten desselben

[1] Signaturen von Methoden werden im Folgenden bei der ersten Erwähnung vollständig angegeben; bei weiteren Vorkommen werden - um die Geduld des Lesers zu schonen - nur noch die Bezeichner verwendet.

Prozessors führt oder ob die Zielinstanz von einem anderen Prozessor verwaltet wird. Auf welchem Prozessor der Adressat einer Nachricht zu finden ist, ist nur den jeweiligen *Port*-Objekten bekannt, die die Nachrichten selbständig versenden.

4.2.2. Die Simulationsschleife

Die Verwaltung der Simulationsanwendung geschieht zum großen Teil durch die Klassen *SimManager* und *Scheduler*. Zum Erstellen einer eigenen Anwendung muss die Klasse *SimManager* überschrieben werden (Details dazu im Abschnitt 4.3.1). Nach dem Start initialisiert das Framework zuerst die MPI-Funktionen, dann wird mit einem Aufruf der von der Anwendung zu implementierenden Methode *void initializeModel()* das Simulationsmodell eingelesen. Die Modellknoten (abgeleitet von der Klasse *Core*, siehe Abschnitt 4.3.2) werden erstellt und gemäß der Modellvorgaben durch Objekte vom Typ *Port* verbunden.

Das Verfahren basiert wie beschrieben auf einer konservativen Parallelisierungsmethode, der synchronen Ausführung, wie sie in Abschnitt 2.2.1.1 beschrieben wird. Die Details der Implementierung der Barriere werden von MPI gekapselt und sind daher auch aus Sicht der Anwendung transparent. Es ist jedoch prinzipiell möglich, das Framework gemäß des Null-Nachrichten-Verfahrens oder zu einer optimistischen Simulation umzubauen und - weiterhin aus Anwendersicht transparent - mit Hilfe des Time Warp-Verfahrens zu implementieren.

Durch *SimManager* wird auch die initiale Zuordnung der Modellknoten zu den beteiligten Prozessoren vorgenommen, die Objekte werden dann per MPI an die zugeordneten Prozessoren übergeben. Dann beginnt die Ausführung der Simulationsläufe, deren Anzahl per Aufruf der Methode *void setNumberOfRuns(int p_iNoRuns)* festgelegt wird. Vor jedem Lauf ruft das Framework die vom Anwender zu implementierende Methode *void prepareForNextRun(int p_iNumberOfCurrentRun)* auf. Dann wird ein Reset des Modells vorgenommen, daraufhin der eigentliche Simulationslauf ausgeführt. Als letzter Schritt in der Schleife wird der Anwendung über den Aufruf der Callback-Funktion *void runDone(int p_iNumberOfCurrentRun)* die Möglichkeit gegeben, die Ergebnisse des Laufs zu verarbeiten.

Die von der Bibliothek in der Methode *void run()* verwendete Implementierung der Simulationsschleife wird in Algorithmus 4.1 vereinfacht gezeigt. Zu Beginn des Simulationslaufs wird die aktuelle Simulationszeit auf die vorgegebene Startzeit gesetzt. Die Schleife selbst wird solange ausgeführt, bis entweder die maximal zulässige Simulationszeit den vorgegebenen Wert *m_iMaxtime* überschritten hat, oder alle Modellknoten im Zustand *halted* (angehalten) sind (Näheres dazu in Abschnitt 4.3.2).

Vor Beginn des eigentlichen Simulationsschritts wird zuerst der Synchronisierungsschritt ausgeführt, für die Synchronisierungsbarriere wird dabei die MPI-Funktion *MPI_Barrier()* genutzt. Die Messung der beim Aufenthalt in der Barriere vergehenden Synchronisierungszeit $t_s(p)$ wird wie in Abschnitt 3.2.3.1 beschrieben zur Berechnung der Auslastung verwendet.

Algorithmus 4.1 Simulationsschleife (vereinfacht) in C++

```
void Scheduler::run() {
    bool allCellsHalted = false;
    Tick *t = new Tick(m_iStarttime);
    while(!allCellsHalted && t->getTime()<m_iMaxtime) {
        synchronize(t->getTime());
        allCellsHalted = true;
        for(vector<Cell*>::iterator itCells=m_vpCells.begin();
            itCells!=m_vpCells.end(); itCells++) {
                if((*itCells)->handleTick(t)) allCellsHalted = false;
        }
        this->reshuffle();
        t->inc();
        if(m_iTimeToCheck>0) {
            m_iTimeToCheck--;
        } else {
            checkForFuses();
            checkForSplits();
            m_iTimeToCheck = Parameters::c_iSPLIT_CHECK_PERIOD;
        }
    }
}
```

Im folgenden Hauptteil des Simulationsschritts wird für jeden Modellknoten die Ereignisbehandlungsroutine *handleTick* aufgerufen (Details hierzu in Abschnitt 4.3.2), die einen Boole'schen Wert zurück gibt. Ist dieser gleich *true,* so ist der Knoten aktiv oder hat zumindest ein Simulationsereignis terminiert. Sind alle Knoten inaktiv und keine Ereignisse vorhanden, kann der Simulationslauf vor dem Erreichen der maximalen Simulationszeit beendet werden.

Falls sich die *Core*-Objekte durch die zur Verfügung gestellten Timer-Funktionen zur jeweils nächsten Ereigniszeit wecken lassen, so ergibt sich aus Sicht der Anwendung eine ereignisbasierte Simulation mit variablem Zeitfortschritt, wie beschrieben in Abschnitt 2.1. Insbesondere bei geringer Ereignisdichte ist dies zu bevorzugen, da so viele überflüssige Aufrufe der Ereignisbehandlungsroutinen (siehe dazu Abschnitt 4.3.2) eingespart werden. Werden keine Timer eingesetzt, werden die Objekte zu jedem Zeitschritt angesprochen, dies entspricht dann einer diskreten Simulation mit fixem Zeitfortschritt (siehe nochmals Abschnitt 2.1).

Bei der Modellbildung wird i.d.R. davon ausgegangen, dass ein Simulationsschritt in allen Modellknoten zugleich ausgeführt wird, die Reihenfolge der Ansprache darf also keinen Einfluss auf die Simulationsergebnisse haben. Durch eine in jedem Simulationsschritt identische Aufrufreihenfolge der Knoten kann es jedoch zu so einer Beeinflussung durch Details der Implementierung kommen, man spricht hier auch von Simulationsartefakten. Um diese zu vermeiden wird die Reihenfolge zwischen zwei Simulationsschritten jeweils zufällig permutiert.

Nach dem Erhöhen der Simulationszeit werden zum Abschluss des Simulationsschritts ggf. noch Knoten wie in Abschnitt 3.2.4 beschrieben aufgeteilt resp. verschmolzen. Die Ausführung dieser Berechnung kann mit Hilfe eines Parameters *c_iSPLIT_CHECK_PERIOD* so konfiguriert werden, dass sie nicht bei jedem Simulationsschritt durchgeführt wird.

4.2.3. Management der Transienten

Transienten werden auf Basis der Klasse *Message* implementiert und enthalten die für die Kooperation mit den Modellknoten nötigen Methoden und Variablen. Details zum Umgang mit *Message* finden sich in Abschnitt 4.3.3.

Für das Verschieben von Transienten zwischen *Cell*-Objekten werden zwei Fälle unterschieden, die aus Sicht der Anwendung allerdings transparent sind: Falls Sender und Empfänger vom selben Prozessor verwaltet werden, wird lediglich ein Zeiger an den Empfänger übergeben. Falls Sender und Empfänger durch zwei unterschiedliche Prozessoren verwaltet werden, werden die Objekte per MPI verschoben. Das Framework ruft vor dem Transfer die Methode *void serialize()* auf. Hier wird das Objekt serialisiert, d.h. zusammengesetzte Datenstrukturen werden in einen zusammenhängenden Bytestrom umgewandelt. Nach dem Transfer zum Empfängerprozessor ruft das Framework die Methode *void deserialize()* auf, die den Transienten rekonstruiert. Dieser wird dann der als Empfänger bestimmten *Core*-Instanz übergeben und kann von dem dort implementierten Modellverhalten verarbeitet werden.

4.2.4. Aufteilen und Verschmelzen von Knoten

Wie in Abschnitt 3.2.4 beschrieben, werden überdurchschnittlich belastete Knoten aufgeteilt um sowohl einen besseren Lastausgleich als auch eine genauere Analyse der Modellaktivität zu ermöglichen; zwei benachbarte Knoten mit Unterlast werden zu einem Knoten zusammengefasst.

Die Suche nach möglichen Kandidaten findet dabei nach den in Abschnitt 3.2.4 be-

schriebenen Kriterien im Rahmen der Simulationsschleife in den Methoden *void checkForSplits()* und *void checkForFuses()* der Klasse *Scheduler* statt.

Die eigentliche Aufteilung eines Objekts findet dann in dessen Methode *Cell* split()* statt.

Analog zum Aufteilen der Knoten werden im Rahmen der Simulationsschleife in der Klasse *Scheduler* Kandidaten a und b zum Verschmelzen ermittelt. Das eigentliche Verschmelzen wird dann von *Cell*-Objekt a in der Methode *State* fuse(Cell* p_pCellB)* durchgeführt. Ihr wird die Instanz b von *Cell* übergeben; dieses Objekt wird nach der Ausführung der Methode aus dem Modellgraphen genommen und gelöscht.

4.2.5. Lastverlagerung

Im Rahmen des dynamischen Lastausgleichs werden Instanzen von *Cell* zwischen den Prozessoren verschoben. Dazu müssen diese Objekte inklusive der Ports und der von *Core* abgeleiteten Objekte zunächst serialisiert werden. Dies geschieht für die Anwendung transparent durch einen Aufruf der Methode *void freeze(Message* p_pMessage)*. Im Rahmen dieser Methode ruft das Framework auch die von der von *Core* abgeleiteten Klasse zu implementierende Methode *int freeze(int p_iOffset, Message* p_pMessage)* auf (näheres dazu in Abschnitt 4.3.2). Die resultierenden Bytestöme werden dann mit Hilfe der MPI-Bibliothek zu den jeweiligen Empfänger-Prozessoren gesendet. Dort werden sie mit Hilfe der Methode *void thaw(Message* p_pMessage)* als *Cell*-Objekte rekonstruiert und in den Modellgraphen eingehängt.

4.3. Schnittstellen zur Anwendung

Um eine auf das *cellforce*-Framework aufsetzende Simulationsanwendung zu entwickeln, müssen lediglich die drei Klassen *SimManager*, *Core* und *Message* überschrieben werden. In den nächsten Abschnitten werden die Schnittstellen[2] der einzelnen Klassen beschrieben.

4.3.1. Schnittstelle der Klasse *SimManager*

Die Klasse *SimManager* muss von der Anwendung überschrieben werden, sie ist ein guter Ort für die Main-Methode als Einsprungpunkt der Anwendung (für eine beispielhafte

[2] Im Folgenden wird die API-Version 4 vom 20. Dezember 2012 beschrieben. Änderungen sind für spätere Versionen nicht ausgeschlossen, sie werden in der dem cellforce-Framework beiliegenden Datei *changes.txt* beschrieben.

Main-Methode siehe Algorithmus 4.2). Hier findet ein Großteil der Verwaltung der Anwendung statt, u.a. wird wie bereits beschrieben in dieser Klasse das Modell (z.B. aus einer Datenbank) eingelesen und als Graph aufgespannt, die Parallelität verwaltet, der Ablauf einer gegebenen Zahl von Simulationsläufen gesteuert und deren Ergebnisse ausgewertet.

Algorithmus 4.2 Routine *main()* einer Beispielanwendung

```
int main(int argc, char* argv[]) {
    // --- Instanz der Klasse DemoSim erzeugen
    // --- DemoSim erbt von SimManager
    DemoSim demosim(argc, argv);
    // --- SimMananger: Modell initialisieren und auf
    // --- Rechenknoten verteilen. Hier findet
    // --- der callback an initializeModel() statt
    demosim.initialize();
    // --- SimManager: Maximale Simulationszeit setzen
    demosim.setMaxTime(DemoConst::MAX_SIMTIME);
    // --- SimManager: Initiale Simulationszeit setzen
    demosim.setStartTime(DemoConst::MIN_SIMTIME);
    // --- SimManager: Anzahl der Läufe setzen
    demosim.setNumberOfRuns(DemoConst::NUMBER_OF_RUNS);
    // --- SimManager: Simulation laufen lassen
    State * result = demosim.runModel();
    // --- DemoSim: Auswerten und in DB eintragen
    demosim.evaluateResult(result);
    // --- SimManager: MPI geordnet abschalten
    demosim.shutDown();
    return 0;
}
```

Die von der Klasse *SimManager* erbende Hauptklasse der Anwendung muss vier Methoden implementieren:

- *virtual void initializeModel():* Das Framework übergibt im Laufe der Initialisierung den Kontrollfluss an diese Methode. Hier sollte die Anwendung das Simulationsmodell einlesen und aufspannen.
- *virtual Core* createCustomCore():* Das Framework erwartet von dieser Methode als Rückgabewert den Zeiger auf eine Instanz der an die Anwendung angepassten Klasse *Core* (siehe Abschnitt 4.3.2).

- *virtual void prepareForNextRun(int p_ iNoRun):* Diese Methode wird vor jedem Simulationslauf aufgerufen; ggf. nötige Vorbereitungen können hier getroffen werden.
- *virtual void runDone(int p_ iNoRun):* Diese Methode wird nach jedem ausgeführten Simulationslauf aufgerufen. Auswertungen, statistische Erhebungen, etc., sollten hier stattfinden.

Die Klasse *SimManager* stellt eine Reihe von Methoden zur Verfügung, um die Simulation zu steuern und die Entwicklung der oben beschriebenen Methoden zu vereinfachen. Die wichtigsten sind:

- *State* initialize():* Das Framework, die zugrunde liegende MPI-Bibliothek und das Simulationsmodell werden initialisiert, von hier aus wird u.a. die oben beschriebene Methode *initializeModel()* aufgerufen. Diese Methode muss vor dem Anstoßen der Simulationsläufe ausgeführt werden (siehe auch Algorithmus 4.2).
- *State* addCell(Cell* p_ pNewNode):* Fügt dem Modellgraphen einen Knoten hinzu.
- *State* runModel():* Startet die Ausführung des Simulationsmodells. Die Anzahl der Simulationsläufe, Start- und Zielwert der Simulationszeit kann vorher gesetzt werden (siehe unten).
- *void resetModel():* Der Reset-Baum wird vor jedem Simulationslauf automatisch aufgerufen, d.h. in sämtlichen Modellbestandteilen wird die Methode *reset()* ausgeführt.
- *void shutDown():* Der Aufruf dieser Methode bewirkt das geordnete Schließen der Simulationsanwendung.
- *void printModel():* Das Simulationsmodell wird in Textform (z.B. zu Testzwecken) auf der Konsole ausgegeben.

Dazu kommen noch eine Reihe von Zugriffsmethoden, die hier ebenfalls kurz aufgeführt werden sollen.

- *void setMaxTime(int p_ iMaxtime):* Setzt den Wert der Simulationszeit, nach dessen Überschreiten der Simulationslauf gestoppt werden soll. Ist kein Wert gesetzt, läuft die Simulation solange weiter, bis sämtliche Knoten im Zustand *halted* sind (siehe unten).
- *void setStartTime(int p_ iStarttime):* Setzt den Startwert der Simulationszeit. Voreingestellt ist hier der Wert 0.

- *void setNumberOfRuns(int p_iNoRuns):* Legt die Anzahl der auszuführenden Simulationsläufe fest. Voreingestellt ist ein einzelner Simulationslauf.
- *int getID():* Gibt einen identifizierenden Wert für den aktuellen Prozessor zurück. Der Prozessor mit dem Wert 0 ist als Controller zuständig für die Ausführung des sequentiellen Teils des Frameworks, wie das initiale Einlesen und Verteilen des Modells. Aus Sicht der Anwendung ist diese Sonderfunktion des Prozessors 0 transparent.
- *int getSize():* Gibt die Anzahl der beteiligten Prozessoren zurück.
- *int getSizeOfModel():* Gibt die Anzahl der Modellknoten zurück.
- *Cell* getCell(int p_iIndex):* Gibt den Modellknoten mit dem Index *p_iIndex* zurück.
- *int getNumberOfRuns():* Gibt die Gesamtzahl der auszuführenden Simulationsläufe zurück.
- *int getCurrentRunNumber():* Gibt die Nummer des aktuell ausgeführten Simulationslaufs zurück.

4.3.2. Schnittstelle der Klasse *Core*

Die von der abstrakten Klasse *Core* erbenden Klassen bildet die in Kapitel 3 beschriebene Funktionalität der Modellknoten ab. Jede Instanz vor *Core* wird vom *cellforce*-Framework in ein Objekt vom Typ *Cell* eingebettet, das die Verwaltung von Kommunikation und Nachbarschaftsbeziehungen übernimmt. Die Objekte sind über mit der Klasse *Port* realisierte Kommunikationskanäle miteinander verbunden und bilden so den in Abschnitt 3.1 beschriebenen Modellgraphen. In den unten beschriebenen Ereignisbehandlungsroutinen *handleTick, handleMessage* und *handleEvent* findet sich der größte Teil der zum Modell gehörenden Logik.

Eine von *Core* erbende Klasse muss eine Reihe von Methoden implementieren.

- *virtual void handleTick(Tick* p_pTick):* Diese Methode wird vom Framework ausgeführt, um den Knoten über den Fortschritt der Simulationszeit in Kenntnis zu setzen. Die Methode wird nicht aufgerufen, wenn sich der Knoten im Zustand *halted* (siehe unten) befindet.
- *virtual void handleMessage(Port* p_pPort, Message* p_pMessage):* Über diese Methode wird der Knoten benachrichtigt, wenn eine von einem anderen Knoten

gesandte Nachricht für ihn eingetroffen ist (siehe Abschnitt 4.3.3 zur Klasse *Message*).

- *virtual void handleEvent(Event* p_pEvent):* Das Framework benachrichtigt den Knoten über diese Ereignisbehandlungsroutine über das Eintreffen eines mit *void scheduleEvent(Event* p_pEvent)* (siehe unten) terminierten Simulationsereignisses.

- *virtual double getLoadEstimation():* Das Framework erwartet als Rückgabe eine Abschätzung des eigenen Beitrags zur Rechenlast des Simulators. Der Wert kann neben der gemessenen Rechenzeit verwendet werden, um Kandidaten für das Aufteilen und Zusammenlegen von Knoten zu finden.

- *virtual int freeze(int p_pOffset, Message* p_pMessage)*: Diese Methode wird aufgerufen, wenn der Modellknoten auf einen anderen Prozessor verschoben werden soll. Das Framework erwartet, dass der Knoten seine Aktivitäten stoppt und seinen Zustand als Bytestrom kodiert in der übergebenen Instanz der Klasse *Message* übergibt.

- *virtual void thaw(int p_pOffset, Message* p_pMessage):* Hier wird einer neu erstellten Instanz eine Nachricht übergeben, die den Zustand eines von einem anderen Prozessor zum aktuellen Prozessor verlagerten Knotens als Bytestrom kodiert. Die Bibliothek erwartet, dass der Knoten den Zustand dekodiert und übernimmt.

- *virtual bool isSplitable():* Der Knoten gibt darüber Auskunft, ob er in seinem aktuellen Zustand teilbar ist.

- *virtual bool isFuseable():* Analog erwartet das Framework hier Auskunft, ob der Knoten zur Verschmelzung mit einem Nachbarn bereit ist.

- *virtual Core* split():* Diese Methode ist maßgeblich für das in Abschnitt 3.2.4 beschriebene Aufteilen eines Knotens. Hier wird als Rückgabewert der Zeiger auf eine Instanz einer von *Core* erbenden Klasse erwartet, auf die ein Teil des Modellzustands des aktuellen Knotens übertragen wurde. Die zurück gegebene Instanz wird dann in den Modellgraphen aufgenommen und eine Verbindung über ein *Port*-Objekt zur aktuellen Instanz hergestellt.

- *virtual State* fuse(Core* p_pCoreB):* Analog erwartet das Framework hier, dass der aktuelle Knoten die Modelllogik des übergebenen Knotens *p_pCoreB* übernimmt. Dieser zweite Knoten wird im Anschluss von der Bibliothek aus dem Modellgraphen gelöscht.

- *virtual void reset():* Diese Methode wird vor jedem Simulationslauf aufgerufen, hier soll der Zustand des Knotens auf die Initialwerte zurück gesetzt werden.
- *virtual string toString():* Das Framework erwartet hier als Rückgabe eine Zeichenkette, die den Zustand des Knotens beschreibt. Dies wird im Kontext der Methode *printModel* der Klasse *SimManager* verwendet.

Zur Realisierung der Funktionalität stellt die Klasse *Core* einige Hilfsmethoden zur Verfügung:

- *State* sendMessage(Port* p_pPort, Message* p_pMessage):* Mittels dieser Methode wird ein Transient oder eine andere Nachricht über einen bestimmten Port an einen anderen Simulationsknoten gesendet. Dieser wird dann vom Framework über einen Aufruf von *handleMessage* darüber in Kenntnis gesetzt.
- *void halt():* Durch den Aufruf dieser Methode wird der Modellknoten in den Zustand *halted* versetzt. Ein angehaltener Knoten wird nicht über das Verstreichen von Simulationszeit in Kenntnis gesetzt, die Routine *handleTick* wird nie aufgerufen. Der Knoten wird beim Empfang einer Nachricht über die Methode *handleMessage* oder beim Erreichen des Zeitstempels eines terminierten Ereignisses geweckt.
- *void scheduleEvent(Event* p_pEvent):* Mit Hilfe dieser Methode wird ein Simulationsereignis erzeugt, das dem Knoten zum angegebenen Zeitpunkt zur Verarbeitung übergeben wird. Dabei wird dann der Knoten ggf. aus dem angehaltenen Zustand geweckt. Auf diese Weise lässt sich mithilfe dieser Funktion ein ereignisbasiertes Modell implementieren. Die Ereignisse werden intern in einer Prioritätswarteschlage verwaltet, es können also zu einem Zeitpunkt mehrere Ereignisse terminiert sein.

Dazu kommen noch eine Reihe von weitgehend selbst erklärenden Zugriffsmethoden. Zum Lesen der Anzahl der Ports der Zelle stehen die Methoden *int getNumberOfPorts(), int getNumberOfOutboundPorts(), int getNumberOfInboundPorts()* und *int getNumberOfBidirectionalPorts()* zur Verfügung, zum Zugriff auf den Port mit dem die Zielzelle identifizierenden Wert *p_iCellID* die Methode *Port* getPort(int p_iCellID).*

Der Bezeichner des Prozessors, der den aktuellen Knoten verwaltet, lässt sich mittels *int getProcessorID()* lesen. Schließlich kann noch auf den identifizierenden Wert des Knotens zugegriffen werden mittels *void setID(int p_iNewID)* und *int getID().*

4.3.3. Schnittstelle der Klasse *Message*

Alle Nachrichtenklassen, deren Instanzen zwischen Modellknoten verschickt werden, erben von der Klasse *Message.* Wichtigste Anwendung ist das Kapseln der Transienten des

Simulationsmodells, aber auch die Implementierung darüber hinaus gehender Nachrichten und Signale, die zwischen Teilmodellen vermittelt werden.

Von *Message* erbende Klassen müssen die beiden folgenden Methoden implementieren:

- *virtual void serialize()*: Bevor die Nachricht zwischen den Prozessoren verschickt werden kann, müssen die enthaltenen Datenstrukturen serialisiert, also in einen Bytestrom verwandelt werden.
- *virtual void deserialize():* Im Rahmen dieser Methode wird nach dem Empfang eines Bytestroms dieser in für die Anwendung sinnvolle Werte zurück konvertiert.

Um das Vorgehen zu erleichtern, stellt *Message* dazu eine Reihe von Hilfsmethoden zur Verfügung: *void code32(int p_iAddress, int p_iValue), void code8(int p_iAddress, char p_cValue), void code16(int p_iAddress, short p_sValue), void codeChar(int p_iAddress, char* p_pcValue), void codeDouble(int p_iAddress, double p_dValue)* und *void codeBoolean(int p_iAddress, bool p_bValue)* codieren Variablenwerte des jeweils angegebenen Typs an eine vorgegebene Adresse im Bytestrom.

Entsprechend stellen die Funktionen *int decode32(int p_iAddress), char decode8(int p_iAddress), short decode16(int p_iAddress), char* decodeChar(int p_iAddress), double decodeDouble(int p_iAddress)* und *bool decodeBoolean(int p_iAddress)* die beschriebenen Werte aus einem empfangenen Bytestrom wieder her.

Dazu kommen noch die Funktionen *void print()* zur Kontrollausgabe des in einer Instanz von *Message* enthaltenden Bytestroms und *void copyTo(Message* p_pTarget)* zum Kopieren einer Instanz von *Message* in eine andere.

Zum Abschluss sollen noch eine Reihe von weitestgehend selbst erklärenden Zugriffsmethoden aufgezählt werden: *int getType(), void setType(int p_iType), int getSubtype(), void setSubtype(int p_iSubtype), void setOffset(int p_iOffset), int getOffset(), int getLength()* und *int getCapacity()*.

5. Experimente mit zufällig erzeugten Graphen

Um die dynamischen Eigenschaften des vorgestellten Ansatzes zu untersuchen, wird im Folgenden eine auf dem in Kapitel 4 beschriebenen Framework aufsetzende Testanwendung entwickelt. Um nicht von den Eigenschaften eines bestimmten Systems abhängig zu sein, wird dabei analog zur in Abschnitt 3.1 geschilderten Idee des Verfahrens mit zufällig erzeugten Graphen und sich darauf bewegenden Token gearbeitet.

Vor Beginn der Simulationsläufe wird ein zufälliger Graph $G(V, E)$ mit vorgegebener Zahl an Knoten $v \in V$ und Kanten $e \in E$ generiert und mit einer Reihe von Token versehen. Die Token erzeugen in jedem Simulationsschritt Rechenlast und reisen während der Simulationszeit durch den Graphen. Sie verweilen auf einem Knoten t_{max} Simulationsschritte und reisen dann über eine zufällig ausgewählte Kante weiter. Im Laufe der mit fixem Zeitinkrement fortschreitenden Simulation verlagert sich so die Rechenlast durch die Regionen des Modells, die Belastung der Prozessoren verändert sich also dynamisch.

Die Prozessorlast wird erzeugt, indem in jedem Schritt der Simulation alle Primzahlen $q_i \leq q_{max}$ mit dem als Sieb des Eratosthenes bekannten Verfahren (beschrieben z.B. von Sedgewick in [61], S. 85) berechnet werden. Für jeden Knoten des Modells wird dabei die größte zu untersuchende Zahl q_{max} durch Addition der festgelegten Grundlast des Knotens und der Gewichte der auf dem betreffenden Knoten vorhandenen Token bestimmt.

5.1. Entwurf und Entwicklung der Anwendung

Zur Realisierung der skizzierten Beispielanwendung müssen wie in Abschnitt 4.3 beschrieben die Klassen *SimManager*, *Core* und *Message* angepasst werden. Die von *SimManager* erbende Hauptklasse der Anwendung heißt hier *DemoSim*, die von *Core* erbende, die Knoten im Graphen abbildende Klasse heißt *Node*. Die Token werden durch die gleichnamige Klasse *Token* implementiert, die von *Message* erbt.

Die Anwendung steht auf der dieser Arbeit beigelegten CD-ROM zur Verfügung (siehe Anhang C).

5.1.1. Anpassung der Klasse *SimManager*

Die Klasse *DemoSim* erbt von der Klasse *SimManager*. Sie implementiert neben der *main*-Methode die in Abschnitt 4.3.1 beschriebenen Methoden *initializeModel*, *createCustomCore*, *prepareForNextRun* und *runDone*.

In der Methode *initializeModel* wird der oben beschriebene, zufällig aufgebaute Modellgraph erzeugt. Eine vorgegebene Anzahl Objekte vom Typ *Cell* wird erzeugt, die je mit einem Objekt vom Typ *Node* versehen werden. Ein vorgegebener Anteil $0 \leq p_{token} \leq 1$ der Objekte erhält dabei eine Instanz der Klasse *Token*. Im Anschluss wird in jedem *Cell*-Objekt eine vorgegebene Anzahl von *Port*-Instanzen erzeugt, die das Objekt mit zufällig ausgewählten anderen *Cell*-Knoten verbinden. Zum Abschluss der Routine werden die erzeugten Objekte mittels *addCell* in das Simulationsmodell eingefügt.

Die Methode *createCustomCore* erzeugt lediglich eine Instanz der Klasse *Node* und gibt sie zurück. In den Methoden *prepareForNextRun* und *runDone* werden die während der einzelnen Läufe erzeugten Daten verwaltet, insbesondere die gemessenen Laufzeiten. Auch diese Methoden sind sehr einfach gehalten.

Die Methode *main* der Anwendung wurde bereits in Algorithmus 4.2 gezeigt. Hier wird zuerst das Framework initialisiert und dabei das Simulationsmodell erzeugt. Dann werden einige Werte wie die maximale Simulationszeit und die Anzahl der auszuführenden Läufe gesetzt, und im Anschluss die eigentliche Simulation gestartet.

5.1.2. Anpassung der Klasse *Core*

Die von *Core* abgeleitete Klasse *Node* beherbergt den Großteil des Modellverhaltens. Wie in Abschnitt 4.3.2 beschrieben, implementiert sie die Behandlung von Simulationsereignissen in den Methoden *handleTick* und *handleMessage*.

Die Methode *handleTick* wird im Rahmen jedes Simulationsschritts aufgerufen. Hier wird die Berechnung der Primzahlen bis q_{max} durchgeführt und damit die Rechenlast erzeugt. Das Verfahren benötigt dazu den Wert der größten zu untersuchende Zahl. Für einen Knoten i wird dieser Wert bestimmt, indem zur vorgegebenen Basislast l_{basis} die Anzahl der verwalteten Token multipliziert mit der Tokenlast l_{token} hinzu addiert wird: $q_{max} = l_{basis} + l_{token} * |T_i|$.

Die Methode prüft auch für jedes vorhandene Token, ob die vorgesehene Verweilzeit auf dem aktuellen Knoten abgelaufen ist. Falls ja, wird das Token über einen zufällig ausgewählten Port an einen Nachbarknoten versendet.

Die Methode *handleMessage* wird vom Framework aufgerufen, wenn dem Modellknoten ein Transientenobjekt übergeben wird. Sie trägt das als Parameter übergebene Objekt

in die Liste der vom Knoten verwalteten Token ein.

Über die Ereignisbehandlungsroutinen hinaus muss die Klasse noch einige Hilfsroutinen implementieren: Die Methode *getLoadEstimation* gibt als Lastabschätzung die Anzahl der verwalteten Tokens zurück. In der Methode *freeze* werden die Attribute des Objekts in einen Bytestrom übertragen, die so serialisierten Objekte können dann wie in Abschnitt 3.2.3 beschrieben an andere Prozessoren übertragen werden. Empfangene Nachrichten werden mittels der Methode *thaw* zu Instanzen von *Node* rekonstruiert und in ein *Cell*-Objekt eingebettet.

Die Methoden *split* und *fuse* dienen wie in Abschnitt 4.2.4 beschrieben zur Vorbereitung des Lastausgleichs. In *split* wird ein neuer *Node* angelegt und die Hälfte der vorhandenen Token an ihn übergeben. Die maximale Verbleibzeit für neu hinzu kommende Transienten wird halbiert, so dass die gemeinsame „Größe" der beiden Instanzen der des bisherigen Objekts entspricht. In der Methode *fuse* werden die Token des als Parameter übergebenen *Node* übernommen, das übergebene Objekt wird gelöscht.

In der Methode *reset* werden die Attribute des Objekts auf die Initialwerte zurück gesetzt und die Liste der Token geleert.

5.1.3. Anpassung der Klasse *Message*

Die Implementierung der Token ist sehr simpel gehalten. Die Klasse *Token* erbt direkt von *Message* und enthält lediglich die beiden Attribute *id* und *noTicksLeft*. Diese beiden Integer-Werte werden in der zu implementierenden Methoden *serialize* mithilfe der beschriebenen Hilfsfunktionen in einen Bytestrom konvertiert und in *deserialize* aus dem Bytestrom übernommen. Das Attribut *id* enthält einen das Token identifizierenden Wert, *noTicksLeft* enthält die Anzahl der Schritte, die ein Token auf dem aktuellen Modellknoten verbringen soll.

5.2. Experimente

Im Folgenden werden eine Reihe von Experimenten beschrieben, mit deren Hilfe das dynamische Verhalten von auf dem Framework basierenden Anwendungen betrachtet werden soll. Falls nicht anders beschrieben, wird dabei mit einem mittelgroßen Basismodell gearbeitet. Es besteht aus einem Modellgraphen mit 400 Knoten und 800 Kanten, der vor den Simulationsläufen auf die folgende Weise konstruiert wird:

1. Es werden 400 Modellknoten erstellt. Die Grundlast jedes dieser Knoten liegt bei $l_{basis} = 10.000$.

2. Alle Modellknoten werden durchgegangen, dabei werden für jeden Knoten v jeweils zwei verschiedene Knoten u_1 und u_2 zufällig ausgewählt, die bisher nicht mit v durch eine Kante verbunden sind. Dann wird v durch zwei Kanten $v \longleftrightarrow u_1$ und $v \longleftrightarrow u_2$ mit diesen verbunden.

3. Jeder fünfte Knoten v_i mit $i \bmod 5 = 0$ wird mit einem Token versehen, insgesamt enthält der Graph also 80 Token. Jedes der Token hat ein Gewicht von $l_{token} = 10.000$, eine maximale Verweildauer von $t_{max} = 100$ Simulationsschritten und eine zufällige, gleich verteilte aktuelle Verweildauer $0 < t_v \leq t_{max}$. Dieser Wert t_v wird in jedem Simulationsschritt um eins verringert. Erreicht er null, so wird eine vom Knoten abgehende Kante zufällig ausgewählt; das Token wird über diese Kante verschoben und vom Empfängerknoten mit $t_v = t_{max}$ neu initialisiert.

Der Modellgraph wird vor Beginn der Läufe auf die einzelnen Prozessoren aufgeteilt, indem jeder Knoten einem zufällig ausgewählten Prozessor zugewiesen wird. Der Lastausgleich ist im Basisfall aktiv, pro Lauf werden 500 Simulationschritte durchgeführt.

Das Framework zielt wie beschrieben auf den Einsatz auf einem oder mehreren gekoppelten Desktop-PCs oder Notebooks ab. Die meisten Experimente werden daher auf zum Zeitpunkt der Niederschrift aktuellen Notebooks durchgeführt, es werden dabei bis zu acht Prozessoren oder Prozessorkerne parallel eingesetzt. Da kein einzelner Rechner mit einer ausreichenden Zahl an Prozessor(kern)en zur Verfügung steht, werden die Versuche auf zwei bis drei sternförmig über einen 100-Megabit-Switch verbundenen Rechnern ausgeführt. Diese Topologie entspricht einem typischen Local Area Network (LAN).

Die Notebooks sind mit 6 Gigabyte Hauptspeicher und einem Prozessor vom Typ Intel Core i7-740QM (beschrieben von Intel in [27]) ausgestattet, der über vier Prozessorkerne bei einer Taktfrequenz von 1,73 Gigahertz verfügt. Die Kerne nutzen einen gemeinsamen Cache von 6 Megabyte. Sind nicht alle Kerne ausgelastet, kann der Prozessor die Taktfrequenz auf bis zu 2,93 Gigahertz erhöhen, indem die nicht verwendeten Kerne abgeschaltet und als Kühlfläche genutzt werden. Intel nennt diese Technologie TurboBoost und beschreibt sie in [25]. Um Vergleichbarkeit der mit verschiedener Prozessorzahl durchgeführten Läufe zu erreichen, wird die TurboBoost-Technik abgeschaltet.

Weiterhin verfügen die verwendeten Prozessoren über die von Marr, et al. in [43] vorgestellte Hyperthreading-Technik. Hier werden dem Betriebssystem zusätzlich zu den vier physisch vorhandenen Prozessorkernen noch vier weitere, „virtuelle" Kerne zur Verfügung gestellt. Diese virtuellen Kerne arbeiten nicht mit der vollen Leistung, sondern nutzen zeitweise brachliegende Infrastruktur wie Rechenwerke oder Busse. Die vorhandene Prozessor-Hardware kann so voll ausgenutzt werden. Um eine Vergleichbarkeit zwischen

Anz. Proz.	Basisfall		
	Laufzeit	Speedup	Zugewinn
1	2.010,0	1,00	1,00
2	1.010,3	1,99	0,99
3	681,9	2,95	0,96
4	516,5	3,89	0,94
5	421,0	4,77	0,88
6	372,1	5,40	0,63
7	340,9	5,90	0,49
8	325,8	6,17	0,27

Tabelle 5.1.: Laufzeiten (in Sekunden) und Speedup-Werte für den Basisfall

den eingesetzten Prozessorkernen zu wahren, wird Hyperthreading nicht verwendet.

Für die in Abschnitt 3.2.3 beschriebenen Parameter für den Lastausgleich haben sich die folgenden Werte als sinnvoll erwiesen: Die Messwerte werden geglättet mit $\alpha = 0,5$; die Schranke β für die Synchronisierungszeit bewegt sich mit einem Anpassungsfaktor von $\gamma = 1,1$ zwischen $\beta_{min} = 5$ und $\beta_{max} = 50$, beginnend mit $\beta_0 = 10$; der Anteil der zu verschiebenden Knoten liegt bei $\varphi = 0,02$.

Mit dem beschriebenen Basisfall wird zuerst in Abschnitt 5.2.1 das grundlegende Laufzeitverhalten ausgemessen, und dann in Abschnitt 5.2.2 die Skalierfähigkeit der Anwendung betrachtet. Im Anschluss wird in Abschnitt 5.2.3 der Einfluss des Lastausgleichs auf die Laufzeit bestimmt. Um dann den Einfluss der initialen Verteilung des Modellgraphen auf die beteiligten Prozessoren abzuschätzen, wird in Abschnitt 5.2.4 eine zufällige Zuordnung mit einer Aufteilung auf Basis der Kernighan/Lin-Methode verglichen. Zusätzlich wird eine bewusst schlechte Eingangsverteilung eingesetzt. Die Auswirkungen äußerer Störungen auf den Ablauf der Simulation werden in Abschnitt 5.2.5 betrachtet.

5.2.1. Laufzeitvergleich

Zuerst soll das Verhalten der Anwendung bei der Simulation des Basisfalls betrachtet werden. Dazu wird die Simulation auf bis zu acht Prozessoren ausgeführt. Die Prozessoren stehen der Anwendung dabei - von den vom Betriebssystem genutzten Ressourcen abgesehen - exklusiv zur Verfügung. Für jede Konfiguration werden zehn Simulationsläufe durchgeführt und jeweils Laufzeit und Speedup gemittelt. Die resultierenden Laufzeiten und Speedup-Werte sind in Abbildung 5.1 und Tabelle 5.1 dargestellt. Die Tabelle zeigt zusätzlich den Zugewinn auf, den der zuletzt hinzu geschaltete Prozessor zum Speedup einbringt.

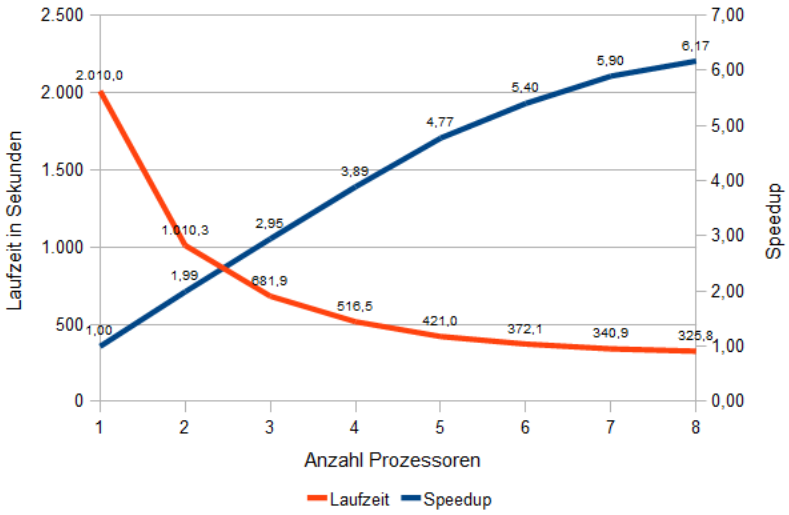


Abbildung 5.1.: Laufzeiten und Speedup-Werte für den Basisfall

Bis zum vierten eingesetzten Prozessor steigt der Speedup steil an, hier wird ein Wert von 3,89 erreicht. Die Laufzeit sinkt von 2.010,0 Sekunden bei der sequentiellen Ausführung um 74,3% auf 516,5 Sekunden bei vier eingesetzten Prozessoren. Jeder weitere genutzte Prozessor bringt deutlich weniger Zugewinn, der achte Prozessor bringt nur noch einen zusätzlichen Speedup von 0,27. Beim Einsatz zusätzlicher Prozessoren verwaltet jeder Prozessor einen immer kleiner werdenden Anteil des Modells, entsprechend steigen die Anteile der Synchronisierungs- und Kommunikationszeit im Vergleich zur reinen Berechnungszeit.

5.2.2. Skalierbarkeit

Um nach den in Abschnitt 3.3 beschriebenen Überlegungen auch praktisch zu überprüfen, ob die Anwendung linear skaliert, sollen nun einige Experimente durchgeführt werden. Die Versuche beginnen mit der Simulation eines Graphen mit 100 Knoten und 200 Kanten auf einem einzelnen Prozessor. Von diesen Werten ausgehend steigen die Größe des Graphen und die Anzahl der eingesetzten Prozessoren linear zu bis 800 Knoten und 1.600 Kanten auf acht Prozessoren.

Die Ergebnisse werden in Abbildung 5.2 und Tabelle 5.2 dargestellt. Eine lineare Regression ergibt als Annäherung für den Skalierungsfaktor die lineare Funktion $s(k) = 0,86 * k + 0,27$, für die Laufzeit der Anwendung $T(k) = 10,16 * k + 483,3$.

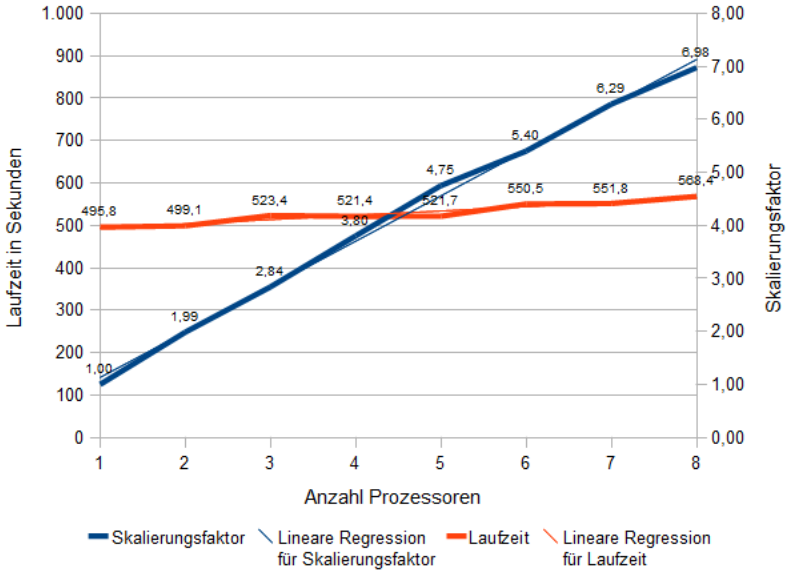


Abbildung 5.2.: Experimente zur Skalierbarkeit der Anwendung

Anz. Proz.	Graph		Skalierbarkeit		
	$\|V\|$	$\|E\|$	Laufzeit	Skalierungsfaktor	Zugewinn
1	100	200	495,8	1,00	1,00
2	200	400	499,1	1,99	0,99
3	300	600	523,4	2,84	0,86
4	400	800	521,4	3,80	0,96
5	500	1.000	521,7	4,75	0,95
6	600	1.200	550,5	5,40	0,65
7	700	1.400	551,8	6,29	0,89
8	800	1.600	568,4	6,98	0,69

Tabelle 5.2.: Experimente zur Skalierbarkeit der Anwendung

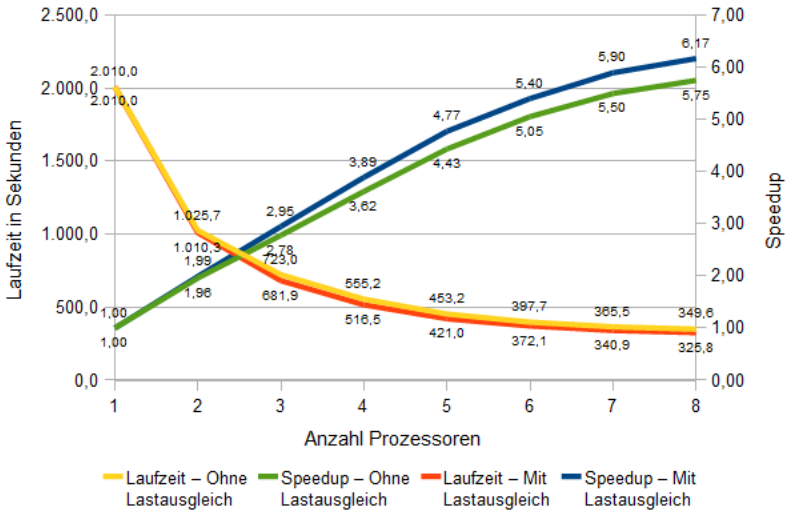


Abbildung 5.3.: Einfluss des Lastausgleichs auf Laufzeit und Speedup

5.2.3. Einfluss des Lastausgleichs

Die Durchführung des Lastausgleichs benötigt Rechenzeit und verursacht Kommunikationslast. Ohne dynamischen Lastausgleich verursachen die über die Partitionsgrenzen wandernden Tokens jedoch eine Lastverschiebung, die die Berechnungszeiten der Teilmodelle auseinander driften lässt und so die durchschnittliche Synchronisierungszeit erhöht. Daher soll geprüft werden, ob die verbesserte Laufzeit den für den Lastausgleich nötigen Aufwand rechtfertigt. Hierzu werden einige Versuche durchgeführt, deren Parameter denen des beschriebenen Basismodells entsprechen.

In Abbildung 5.3 und Tabelle 5.3 wird das Verhalten der Anwendung bei der Simulation des Basismodells mit und ohne Lastausgleich gegenüber gestellt. Es wird deutlich, dass die Differenz der Speedup-Werte mit der Anzahl der eingesetzten Prozessoren steigt. Je höher die Anzahl der Prozessoren ist, auf die das Modell aufgeteilt wird, umso kleiner sind die einzelnen Modellpartitionen. Mit sinkender Partitionsgröße steigt die durch über die Partitionsgrenzen wandernde Token verursachte dynamische Lastverschiebung, die Berechnungszeiten für die einzelnen Teilmodelle entwickeln sich auseinander. Ohne Lastausgleich kann diese nicht ausgeglichen werden, so dass die durchschnittliche Synchronisierungszeit wächst. Die Prozessoren sind daher schlechter ausgelastet, die Laufzeit steigt.

Abbildung 5.4 zeigt für die einzelnen Simulationsschritte eines typischen Laufs mit zwei Prozessoren die Entwicklung der durchschnittlichen Synchronisierungszeit. Es ist

Prozessoren	Mit Lastausgleich		Ohne Lastausgleich		Differenz
	Laufzeit	Speedup	Laufzeit	Speedup	Speedup
1	2.010,0	1,00	2.010,0	1,00	0,00
2	1.010,3	1,99	1.025,7	1,96	0,03
3	681,9	2,95	723,0	2,78	0,17
4	516,5	3,89	555,2	3,62	0,27
5	421,0	4,77	453,2	4,43	0,34
6	372,1	5,40	397,7	5,05	0,35
7	340,9	5,90	365,5	5,50	0,40
8	325,8	6,17	349,6	5,75	0,42

Tabelle 5.3.: Einfluss des Lastausgleichs auf Laufzeit (in Sekunden) und Speedup

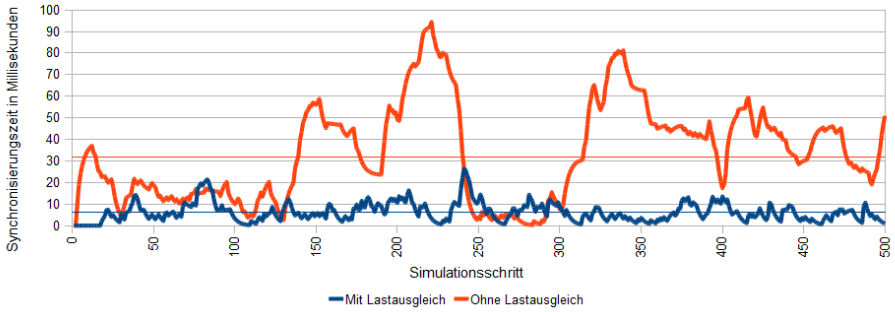


Abbildung 5.4.: Verlauf der Synchronisierungszeit mit und ohne Lastausgleich (Zwei Prozessoren)

offensichtlich, dass der Lastausgleichsmechanismus die Wartezeiten des jeweils weniger belasteten Prozessors deutlich senkt. Die durchschnittliche Synchronisierungszeit steigt ohne Ausgleich der Rechenlast von 6,2 Millisekunden um 545% auf 33,8 Millisekunden. Abbildung 5.5 zeigt den entsprechenden Verlauf für vier beteiligte Prozessoren. Die Partitionen sind hier kleiner, die Token wandern daher öfter über Partitionsgrenzen. Die durchschnittliche Synchronisierungszeit steigt von 8,6 Millisekunden um 536% auf 46,1 Millisekunden.

5.2.4. Einfluss der initialen Verteilungsmethode

Bei den bisherigen Versuchen wurde der Modellgraph vor Beginn der Simulationsläufe zufällig partitioniert. Bei vielen Modellen kann eine Graph Partition-Heuristik wie die in Abschnitt 3.2.2 beschriebene Kernighan/Lin-Methode die Anzahl der Kanten zwischen zwei Partitionen deutlich senken. Dieser potentiellen Ersparnis stehen als Kosten die Laufzeit der Heuristik gegenüber.

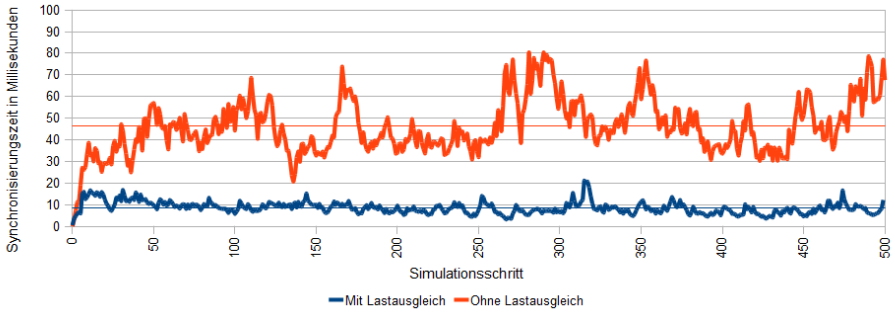


Abbildung 5.5.: Verlauf der Synchronisierungszeit mit und ohne Lastausgleich (Vier Prozessoren)

Unabhängig von der initialen Partitionierungsmethode werden beim implementierten Lastausgleichsverfahren wie in Abschnitt 4.2.5 beschrieben zuerst die Knoten übertragen, die die externen Verbindungskosten zwischen den Prozessoren verringern. Der Modellgraph wird also durch den Lastausgleich während des Laufs entzerrt.

Um den Einfluss der initialen Verteilung zu ermitteln, werden Läufe mit durch die Kernighan/Lin-Heuristik partitionierten Modellen zufälligen Partitionierungen gegenüber gestellt. Zusätzlich wird eine sehr ungleichmäßige Modellaufteilung eingesetzt: Hierbei werden 80% der Modellknoten einem einzelnen Prozessor zugeordnet, die restlichen 20% zufällig auf die anderen eingesetzten Prozessoren verteilt. Die anderen Parameter entsprechen dem beschriebenen Basismodell.

Abbildung 5.6 zeigt den Verlauf der Synchronisierungszeit für einen typischen Lauf. Der durchschnittliche Wert liegt bei diesem Lauf für die zufällige Anfangsverteilung bei 10,3 Millisekunden, für das mit der Kernighan/Lin-Heuristik partitionierten Modell bei 14,4 Millisekunden und für das ungleich verteilte Modell bei 33,0 Millisekunden. Nachdem der Lastausgleichsmechanismus die ungleiche Verteilung binnen ca. 50 Simulationsschritten kompensiert hat, werden keine bedeutenden Unterschiede im Verhalten mehr sichtbar. Werden nur die Synchronisierungszeiten ab Simulationsschritt 50 betrachtet, gleichen sich die Durchschnittswerte an auf 10,3 Millisekunden für die zufällige Verteilung, 14,5 Millisekunden für die Kernighan/Lin-Partitionierung und 9,4 Millisekunden bei der schlechten Verteilung.

In einem weiteren Experiment werden Laufzeiten und Speedup-Werte für randomisiert oder mittels Kernighan/Lin-Verteilung partitionierte Modelle verglichen. Dazu werden bis zu acht Prozessoren eingesetzt, für jeden Messpunkt werden zehn Läufe ausgeführt. Dabei verhalten sich Speedup und Laufzeit bei beiden Partionierungsmethoden ähnlich (siehe Abbildung 5.7 und Tabelle 5.4), die Differenz der Speedup-Werte fällt tendenziell

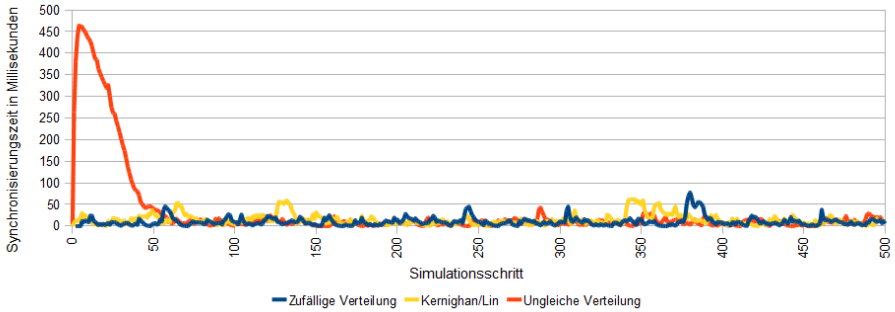


Abbildung 5.6.: Einfluss der initialen Verteilungsmethode auf die Synchronisierungszeit

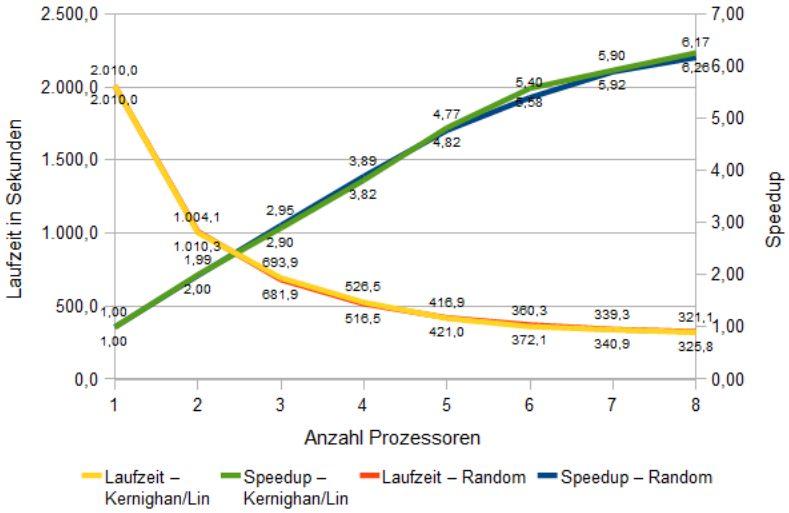


Abbildung 5.7.: Einfluss der initialen Verteilungsmethode auf Laufzeit und Speedup

zu Gunsten der Kernighan/Lin-Methode aus.

5.2.5. Einfluss äußerer Störungen

Bislang wurden lediglich die Auswirkungen aus der Modelldynamik entstehender Störungen der Lastbalance betrachtet. Um die Reaktion des Frameworks auf äußere Störungen zu prüfen, wird zunächst ein Simulationslauf des Basismodells mit den beschriebenen Standardparametern auf zwei Prozessoren ausgeführt. Ab Simulationsschritt 150 wird einer der beiden Prozessoren durch einen externen Benutzerprozess belastet, der die ihm zugewiesene Rechenkapazität komplett ausnutzt. Zum Schritt 300 wird diese externe Last wieder abgeschaltet.

Abbildung 5.8 zeigt an der primären y-Achse den Verlauf der Berechnungs- und Syn-

Prozessoren	Random		Kernighan/Lin		Differenz
	Laufzeit	Speedup	Laufzeit	Speedup	Speedup
1	2.010,0	1,00	2.010,0	1,00	0,00
2	1.010,3	1,99	1.004,1	2,00	-0,01
3	681,9	2,95	693,9	2,90	0,05
4	516,5	3,89	526,5	3,82	0,07
5	421,0	4,77	416,9	4,82	-0,05
6	372,1	5,40	360,3	5,58	-0,18
7	340,9	5,90	339,3	5,92	-0,03
8	325,8	6,17	321,1	6,26	-0,09

Tabelle 5.4.: Einfluss der initialen Verteilungsmethode auf Laufzeit (in Sekunden) und Speedup

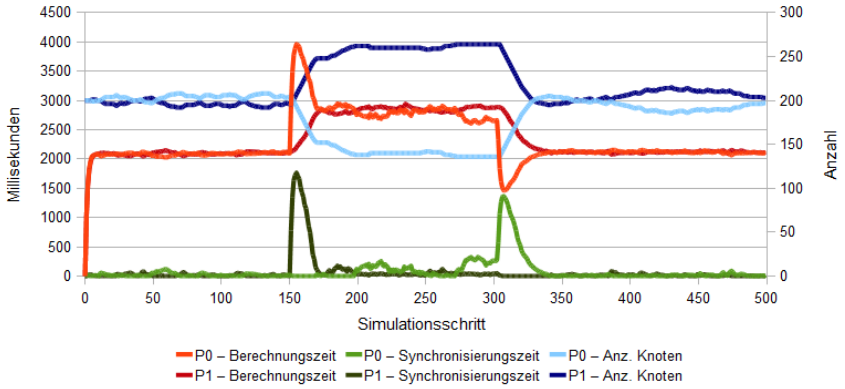


Abbildung 5.8.: Einfluss anderer Benutzerprozesse auf den Simulationsablauf

chronisierungszeit der beteiligten Prozessoren bei dem beschriebenen Experiment. Ab Simulationsschritt 150, also nach der Zuschaltung der externen Last, steigt die pro Iteration benötigte Berechnungszeit für Prozessor p_0 von ca. 2.060 Millisekunden auf etwa das Doppelte (3.970 Millisekunden) an. Offenbar teilt das Betriebssystem die zur Verfügung stehende Prozessorzeit etwa gleich auf die beiden nachfragenden Prozesse auf. Die Berechnungszeit von Prozessor p_1 bleibt unverändert, also steigt die Synchronisierungszeit für p_1 entsprechend an. Nun setzt der Lastausgleich ein und transferiert Modellknoten vom ausgelasteten Prozessor p_0 nach p_1. Nach etwa 35 Simulationsschritten ist ein Gleichgewicht erreicht: p_0 verwaltet nun um die 140 Knoten, p_1 etwa 260 Knoten (siehe nochmals Abbildung 5.8, sekundäre y-Achse). Die Berechnungszeiten für die einzelnen Simulationsschritte sind ausgeglichen, die Synchronisierungszeiten wieder auf das übliche Maß reduziert.

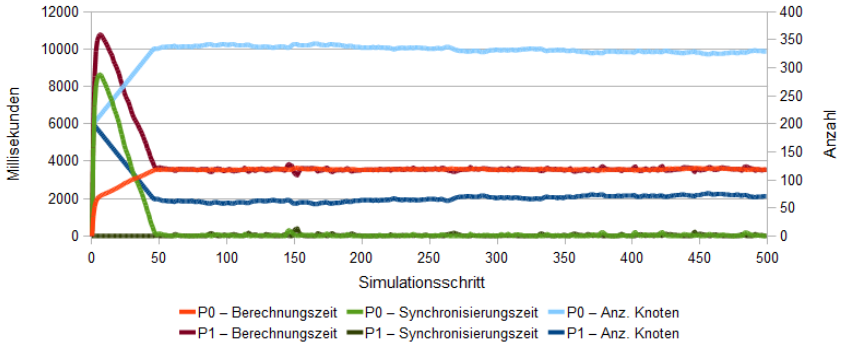


Abbildung 5.9.: Einfluss inhomogener Rechnernetze auf den Simulationsablauf

Zum Simulationsschritt 300 wird die externe Last abgeschaltet, die gesamte Kapazität von Prozessor p_0 steht nun wieder der Anwendung zur Verfügung. Die Berechnungszeit verringert sich entsprechend, dafür steigt die Synchronisierungszeit, da Prozessor p_0 nun auf die Fertigberechnung der Simulationsschritte durch den die Mehrzahl der Knoten verwaltenden Prozessor p_1 warten muss. Hier greift der Lastausgleich wieder ein und verschiebt eine Reihe von Modellknoten vom Prozessor p_1 auf den unausgelasteten Prozessor p_0. Ab ca. Simulationsschritt 340 ist ein Gleichgewicht erreicht, der Rest der Simulation verläuft unauffällig. Offenbar gleicht das dynamische Lastausgleichsverfahren also auch während des Laufs auftretende äußere Störungen effektiv aus.

Mit einem weiteren Experiment soll geprüft werden, ob das Lastausgleichsverfahren dem Framework wie gewünscht ermöglicht, auch inhomogene Gruppen von Rechnern effizient zu nutzen und die zur Verfügung stehende Rechenkapazität auszulasten.

Als weiterer Rechner wird daher gemeinsam mit einem Notebook auf Basis des i7-740QM-Prozessors ein deutlich langsameres Netbook mit einem 32-Bit-Prozessor vom Typ Intel Atom N270 (vorgestellt von Intel in [26]) mit einem einzelnen Prozessorkern bei 1,6 Gigahertz eingesetzt, das über 2 Gigabyte Hauptspeicher verfügt.

Die Abbildung 5.9 zeigt den Ablauf der Simulation einer Instanz des Basismodells: Während die inhomogene Rechenleistung zu Beginn des Experiments noch zu langen Wartezeiten für Prozessor p_0 führt (bis zu 8.594 Millisekunden), wird durch das Eingreifen des Lastausgleichs ca. ab Schritt 50 ein Gleichgewicht hergestellt. In dieser Phase verwaltet Prozessor p_1 noch 60 bis 70 Modellknoten, während Prozessor p_0 330 bis 340 Knoten betreut.

Die Experimente verdeutlichen, dass das dynamische und adaptive Lastausgleichsverfahren das effiziente Berechnen in inhomogenen Rechnernetzen ermöglicht.

6. Anwendung bei der Simulation von Stadtbahnfahrplänen

Im Rahmen des Projekts *Computer Aided Tram Scheduling (CATS)* wurde ein Satz von Software-Werkzeugen entwickelt, mit dessen Hilfe robuste Fahrpläne, die zugleich einer gegebenen Menge von verkehrsplanerischen Restriktionen genügen, generiert, simuliert und bewertet werden können. Diese Anwendungen sollen im Folgenden vorgestellt werden, beginnend mit einer Beschreibung des Projekts CATS und der dort verwendeten Begriffe. Danach wird der genutzte Optimierungsansatz erklärt, gefolgt von einer Beschreibung des auf das *cellforce*-Framework aufsetzenden Moduls zur parallelen Simulation. Im Anschluss wird dann die Software auf den Kölner Stadtbahnfahrplan angewandt, unter Berücksichtigung des im Bau befindlichen Nord-Süd-Tunnels. Um zu zeigen, dass die Werkzeuge auch zur Simulation und Optimierung anderer Netzen geeignet sind, folgt dann eine etwas kompaktere Betrachtung des Stadtbahnnetzes der südfranzösischen Stadt Montpellier. Den Abschluss bildet eine Betrachtung des Laufzeitverhaltens der Simulationssoftware, um so die Eignung des in den Kapiteln 3 und 4 vorgestellten Ansatzes für die Verkehrssimulation einzuschätzen.

6.1. Computer Aided Tram Scheduling (CATS)

In vielen Stadtbahnnetzen nutzen mehrere Linien gemeinsam Ressourcen wie Haltepunkte, Weichen und Gleisabschnitte. Die in Europa weit verbreiteten Taktfahrpläne sind daher typischerweise sehr dicht, an zentralen Haltestellen fahren in den Hauptverkehrszeiten fast minütlich Bahnen ein. Damit sich die hier unweigerlich entstehenden, lokalen Verspätungen nicht durch das Netz ausbreiten, muss ein eingesetzter Taktfahrplan robust sein. Als Robustheit wird in diesem Kontext der Grad bezeichnet, zu dem die Pünktlichkeit des Betriebs unabhängig ist von im Alltag auftretenden, kleineren Störungen, wie z.B. längere Ein- und Ausstiegszeiten, kurze Behinderungen durch PKW, oder ein durch herabfallende Blätter auf den Schienen vermindertes Beschleunigungsvermögen. In robusten Fahrplänen beschränken sich diese Verzögerungen auf die unmittelbar betroffenen Fahrzeuge, in weniger robusten Fahrplänen werden die Verspätungen an gemeinsam

genutzten Ressourcen auf andere Fahrzeuge übertragen, Verspätungswellen breiten sich durch das Netz aus.

Als Indikator für Robustheit dienen hier die im Fahrplan vorgesehenen zeitlichen Abstände zwischen zwei Fahrzeugen. Bei einem angenommenen Zehn-Minuten-Takt lassen sich zwei Linien auf einem gemeinsamen Streckenabschnitt z.B. mit jeweils fünf Minuten planmäßigem Abstand zum nachfolgenden Fahrzeug versehen. Hier kann eines der beteiligten Fahrzeuge um mehr als vier Minuten verspätet sein, ohne dass die nachfolgenden Fahrten verzögert werden. Bei einer extrem ungleichmäßigen Aufteilung in Abstände von neun Minuten und einer Minute kann das eine Fahrzeug zwar mehr als acht Minuten Verspätung haben ohne dass es zu weiteren Konsequenzen kommt, das andere Fahrzeug jedoch überträgt schon eine kleine Verspätung ab einer Minute auf die folgenden Bahnen. Da im Folgenden von typischerweise kleinen Störungen ausgegangen wird, wird die gleichmäßige Aufteilung der zur Verfügung stehenden Zeitpuffer in möglichst hohe Abstände als sehr robust, das Auftreten von sehr kleinen Abständen als nicht robust angesehen.

Da an einen Fahrplan auch eine große Zahl von sich in politischen, ökonomischen oder rechtlichen Überlegungen begründenden planerischen Ansprüchen gestellt wird, kann die Gestaltung eines Fahrplans nicht allein auf ein einzelnes Ziel wie Robustheit ausgerichtet sein. Typische Ansprüche beinhalten festgelegte Abfahrtszeiten an bestimmten Haltepunkten (z.B. an Bahnhöfen der Deutschen Bahn AG), einen höheren Takt in den Kernbereichen bestimmter Linien, garantierte Verbindungen an einigen Stationen („Rendezvous-Verkehr") und Sicherheitsabstände in Tunneln und eingleisigen Streckenabschnitten. Um anwendbare Fahrpläne erstellen zu können, müssen die verwendeten Werkzeuge diese verkehrsplanerischen Ansprüche erfassen und abbilden können.

Mit den Softwareanwendungen sollen mithilfe von Methoden aus der kombinatorischen Optimierung robuste Fahrpläne generiert werden, die zugleich definierbaren planerischen Bedingungen gehorchen. Diese Pläne sollen mit Hilfe einer Simulationsanwendung untereinander und mit gegebenen Plänen verglichen und auf ihre Einsetzbarkeit geprüft werden.

Im Folgenden werden zuerst kurz die verwendeten Begriffe und der Aufbau des Projekts CATS geschildert, insbesondere mit Hinblick auf die zentrale Datenbank zur Verwaltung von Stadtbahnnetzen und -fahrplänen.

6.1.1. Verwendete Begriffe

Die meisten der in den folgenden Ausführungen zum Thema Stadtbahnverkehr verwandten Begriffe sind selbst erklärend. Daher soll hier nur eine kurze Einführung erfolgen und dabei die Gelegenheit genutzt werden die verwendeten Symbole zu definieren.

Am *Haltepunkt* $h \in H$ halten Bahnen und warten auf ihre planmäßige Abfahrtszeit, der Passagieraustausch findet hier statt. Im Folgenden werden die Begriffe Haltepunkt, Haltestelle und Bahnsteig synonym verwendet.

Eine *Station* $s \in S$ ist eine Menge von Haltepunkten, $s_i = \{h_{i_1}, ..., h_{i_n}\}$, die geographisch beieinander liegen und einem gemeinsamen Namen zugeordnet sind. So besteht die Kölner Stadtbahnstation Neumarkt aus den Haltepunkten NEU-12, NEU-53, NEU-77 und NEU-110.

Ein *Streckenabschnitt* $g \in G$ verbindet zwei Knoten im Netz, wobei Knoten sowohl Haltepunkte als auch Weichen sein können. Streckenabschnitte sind gerichtet und verfügen über Attribute wie Länge, Höchstgeschwindigkeit und geplante Überfahrzeiten. Die Menge G der Streckenabschnitte bildet die Menge der Kanten des Netzgraphen. Strecke, Streckenabschnitt und Gleisabschnitt werden synonym benutzt.

Eine *Weiche* $w \in W$ verbindet drei Streckenabschnitte miteinander. Weichen haben eigene Höchstgeschwindigkeiten und müssen vor dem Überfahren reserviert werden. Die Menge der Haltepunkte H bildet zusammen mit der Menge W der Weichen die Menge der Knoten des Netzgraphen. Die Menge W aller Weichen lässt sich unterteilen in zusammenführende Weichen $W_{2,1}$ mit zwei eingehenden und einem ausgehenden Streckenabschnitt und teilende Weichen $W_{1,2}$ mit einem eingehenden und zwei ausgehenden Streckenabschnitten. Weichen mit mehr als zwei eingehenden und/oder ausgehenden Streckenabschnitten sind modellierbar, wurden in der Realität aber nicht beobachtet. Auf gegengerichteten Strecken liegende, zusammen gehörende Weichen mit einer gemeinsamen Bezeichnung bilden eine *Abzweigung* $a = \{w_i, w_j\}$.

Einer ungerichteten *Linie* $l \in L$ sind gerichtete *Linienvarianten* zugeordnet. Im Kölner KVB-Netz besteht die Linie 1 aus einer ganzen Reihe von Varianten, die wichtigsten davon sind die Varianten mit den Bezeichnern 1-BW02, „Linie 1 - Von Weiden nach Bensberg" und 1-FW01, „Linie 1 - Von Bensberg nach Weiden". Ein solcher, eindeutiger Bezeichner besteht dabei aus der Kennung der Linie (hier Linie 1), einer groben Richtungsangabe (BW: „Backward", FW: „Forward") und einem Index (hier 02 resp. 01). Die meisten Linien verfügen über mehr als eine Linienvariante pro Richtung. Zu den üblichen Varianten gehören neben den Hauptvarianten, die die eigentlichen Strecken abfahren, auch noch logistische Varianten, die die Anfangs- oder Endpunkte der Hauptvarianten mit den Depots verbinden. Da die eigentliche Linie außer einem Bezeichner keine zusätzlichen Informationen gegenüber den Linienvarianten hält, verwenden wir beide Begriffe in passendem Kontext synonym. Eine Linienvariante enthält als wichtigstes Attribut einen *Linienverlauf* $(h_1, h_2, ...h_n)$, eine geordnete Liste von zu bedienenden Haltepunkten. Die Begriffe Linienverlauf und Linienroute werden synonym benutzt.

Viele Eigenschaften eines physischen, eine Strecke abfahrenden *Fahrzeugs* werden bestimmt durch den *Fahrzeugtyp*. Dieser definiert Beschleunigungs-, Brems- und Fahrverhalten, Passagierkapazität und durch Anzahl, Beschaffenheit und Position der Türen die Geschwindigkeit des Passagierwechsels.

Eine *Fahrt* $f \in F$ definiert das geplante Abfahren des Verlaufs einer Linienvariante l ab einem bestimmten Startzeitpunkt t_0, also $f = (l, t_0)$. Darauf aufbauend ist der *Umlauf* $u \in \lambda$ eines Fahrzeugs mit $u = (f_1, ..., f_n)$ definiert als die ihm zugeordnete Liste von Fahrten. Der Umlauf beginnt in der Regel mit der Ausfahrt aus dem Depot und endet nach einer Reihe von Fahrten mit der Einfahrt dorthin. Da ein Fahrzeug an einem Betriebstag genau einen Umlauf bedient, werden die Begriffe in dieser Arbeit synonym verwendet.

Der Fahrplan λ entspricht der Vereinigungsmenge aller Umläufe. Aus Sicht der prospektiven Fahrgäste wird also festgelegt, welche Linien ihn wann von welchem Starthaltepunkt zu welchem Zielhaltepunkt befördern und welche Umsteigemöglichkeiten es dort gibt. Aus Sicht des Anbieters legt der Fahrplan fest, ab welcher Startzeit welche Linienverläufe zu bedienen sind.

6.1.2. Architektur des CATS-Projekts

Die Module des Projekts CATS gruppieren sich um eine Datenbank, die Orts-, Netz-, Linien- und Fahrplandaten enthält. Die Optimierungs-, Simulations- und Visualisierungsmodule kommunizieren über die Datenbank oder über XML-Konfigurationsdateien (siehe Abbildung 6.1).

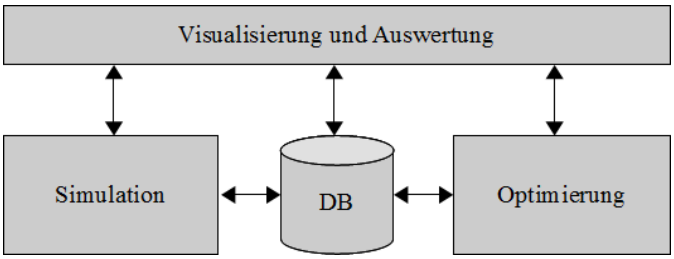


Abbildung 6.1.: Module des CATS-Projekts

Die Datenbank folgt den Vorschriften des vom *Verband Deutscher Verkehrsunternehmen e. V.* (VDV) in [79] beschriebenen ÖPNV5-Datenmodells. Sie wurde von Lückemeyer im Rahmen seiner Diplomarbeit (siehe Lückemeyer in [37]) mit einer ersten Version des Stadtbahnnetzes der Kölner Verkehrsbetriebe AG (KVB) aus dem Jahr 2001 befüllt. Im Rahmen dieser Arbeit wurde die Abbildung dieses Netzes erweitert und korrigiert

und die Datenbank um Modelle des zum Zeitpunkt der Niederschrift aktuellen Netzes von 2012 (inklusive des real eingesetzten Fahrplans) und des nach der Fertigstellung des Nord-Süd-Tunnels zu erwartenden Netzes von 2020 ergänzt. Weiterhin wurde ein Modell des im Jahr 2013 aktuellen Stadtbahnnetzes der Stadt Montpellier erfasst.

Im Rahmen des Projekts CATS wird lediglich eine Teilmenge der im ÖPNV5-Modell definierten Relationen verwendet (für eine Übersicht siehe VDV in [79], S. 54ff.). Aus dem Bereich Ortsdaten wird die Relation REC_ORT benutzt, die Haltestellen und andere Ortspunkte definiert, aus dem Bereich Netzdaten die Relationen REC_SEL und RE_FZT_FELD, die Gleisabschnitte und die hierfür geplanten Fahrzeiten enthalten, und REC_SEL_ZP zur Verwaltung von Zwischenpunkten (z.B. Verkehrsampeln) auf den Gleisabschnitten. Hinzu kommt noch die Relation ORT_HZTF, mit der vom Standard abweichende Haltezeiten verwaltet werden. Aus dem Bereich Liniendaten kommen die Relationen REC_LID und LID_VERLAUF hinzu, die die einzelnen Linienvarianten und deren Verlauf definieren, und aus dem Bereich Fahrplandaten die Relationen REC_UMLAUF, REC_FRT und REC_FRT_HZT, die die eingesetzten Fahrzeuge, deren unter dem aktuellen Fahrplan zu absolvierende Fahrten sowie vom Standard abweichende Haltezeiten beschreiben.

Um mehrere Fahr- und Umlaufpläne verwalten zu können, wurden drei weitere Relationen eingeführt (siehe Anhang A.1), die Kompatibilität zum ÖPNV5-Modell bleibt dabei gewahrt. Dazu wurden Relationen geschaffen, um optionale Daten, wie Quell-Ziel-Matrizen und gemessene Verspätungen, zu verwalten (siehe Anhang A.2), durch deren Verwendung eine deutlich genauere Simulation möglich wird. Alle beschriebenen Module arbeiten aber bereits auf der Basis der durch ÖPNV5 definierten obligatorischen Daten. Aufgrund seiner Kompatibilität zum ÖPNV5-Standard kann die Projektsoftware auf die meisten europäischen Stadtbahnnetze angewandt werden.

Die zum Projekt gehörende Anwendung zur Visualisierung und Auswertung wurde von Ullrich im Rahmen seiner Diplomarbeit [73] erstmals entwickelt und seitdem weiter ausgebaut. Sie erlaubt sowohl graphische Ansichten der erstellten Fahrpläne als auch Visualisierungen von Simulationsläufen und deren Ergebnissen.

6.2. Optimierung von Stadtbahnfahrplänen

6.2.1. Hintergrund

Eine ganze Reihe von Ansätzen zur Optimierung von Straßenbahn- oder Zugfahrplänen sind bekannt. Dazu gehören die von Bampas, et al. in [4], Cacchiana, Caprara und Fischetti in [7], Caimi, et al. in [8], Genç in [20], Schöbel in [59], Speckenmeyer, et al. in

[62] und Suhl und Mellouli in [65] beschriebenen Verfahren.

Einige der erwähnten Ansätze optimieren dabei hinsichtlich einer einzigen allgemeinen Zielgröße, wie zum Beispiel die Minimierung der Fahrzeugverspätung (siehe Schöbel in [59] und Suhl und Mellouli in [65]) oder die Maximierung der Robustheit (siehe Bampas, et al. in [4], Caimi, et al. in [8] und Genç in [20]). Andere benutzen eine Kombination solcher allgemeiner Zielgrößen, wie Robustheit und Gewinnausrichtung (siehe Cacchiana, Caprara und Fischetti in [7]) oder soziale Opportunitätskosten und operative Kosten (siehe Speckenmeyer, et al. in [62]). Die in Abschnitt 6.1 beschriebenen, für die Einsetzbarkeit der Fahrpläne wichtigen verkehrsplanerischen Vorgaben, werden in der besprochenen Literatur jedoch nicht berücksichtigt. Ohne die Beachtung dieser Vorgaben können jedoch keine in der Praxis anwendbaren Fahrpläne erstellt werden.

Da das Problem sehr komplex ist, verwenden viele der Autoren heuristische Ansätze, wie Lagrange-Heuristiken von Cacchiana, Caprara und Fischetti in [7] oder *Simulated Annealing* von Speckenmeyer, et al. in [62]. Andere beschreiben exakte Algorithmen für eingeschränkte Problemklassen, wie von Bampas, et al. in [4] für *Chain Networks* und *Spider Networks*.

6.2.2. Vorgehen

Im Folgenden wird ein Ansatz beschrieben, der heuristische und exakte Methoden kombiniert um optimale Taktfahrpläne zu generieren, die sowohl auf maximale Robustheit zielen als auch einer gegebenen Menge von verkehrsplanerischen Vorgaben entsprechen. Durch die Möglichkeit zur freien Wahl des gemeinsamen Linientakts können, obwohl bei der Anwendung typischerweise die Hauptverkehrsperioden mit den kürzesten vorkommenden Takten betrachtet werden, auch Nebenverkehrs- und Nachtverkehrsperioden untersucht werden. Sind die gewünschten Taktintervalle der einzelnen Linien voneinander verschieden, kann ein globales Taktintervall durch die Verwendung des kleinsten gemeinsamen Vielfachen der Linientakte gebildet werden. Die Taktfrequenz der einzelnen Linien muss dann mit Hilfe von Verstärkerfahrten (siehe Abschnitt 6.2.3) auf das gewünschte Maß erhöht werden.

Hierzu wird ein Branch-and-Bound-Algorithmus (beschrieben z.B. von Dakin in [11]) adaptiert, dem ein Genetischer Algorithmus (beschrieben z.B. von Dreo, et al. in [14]) voran gestellt wird. Dem Branch-and-Bound-Algorithmus wird dabei die beste von der Heuristik gefundene Lösung als initiale obere Schranke übergeben, so dass ein Kaltstart verhindert wird und große Teile des Lösungsbaums schnell ausgeschlossen werden können. Die verbliebene Laufzeit wird gemäß der von Knuth in [34] vorgestellten Methode abgeschätzt.

Das im Folgenden vorgestellte Modell wurde in einer ersten Version von Franz in seiner vom Autor betreuten Diplomarbeit [15] implementiert und wie von Ullrich, et al. in [74] geschildert, erweitert und präzisiert.

6.2.3. Modellbildung

Zur Berechnung der Robustheit eines Fahrplans λ wird für jede Haltestelle h der im Fahrplan vorgesehene Abstand $\delta_{f,pred(f)}(h,\lambda)$ zwischen jeder Fahrt f und deren unmittelbarer Vorgängerin $pred(f)$ untersucht. $\delta_{f,pred(f)}(h,\lambda)$ ist also die Zeitdauer, die am untersuchten Haltepunkt h zwischen der Abfahrt von $pred(f)$ und der Abfahrt von f vergeht.

Um die Komplexität dieser immer wieder vorzunehmenden Berechnung zu reduzieren, wird wie von Genc in [20] beschrieben aus aufeinander folgenden, ausschließlich von den selben Linien befahrenen Haltepunkten, ein maximaler Haltepunkttyp h' erstellt, der durch die Anzahl der einbezogenen Haltepunkte $\varphi_{h'}$ gewichtet wird (s. Abbildung 6.2). Die so reduzierte Menge der Haltepunkte wird durch H' beschrieben.

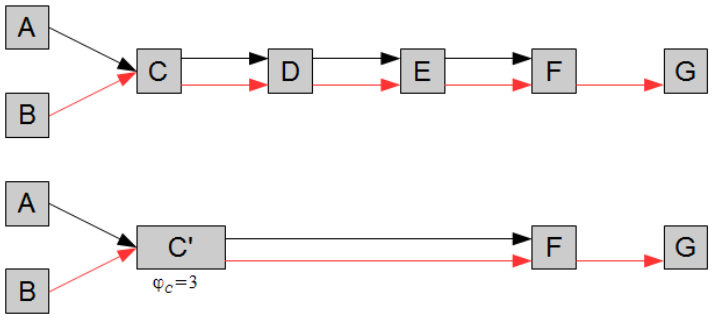


Abbildung 6.2.: Beispiel für die Reduktion der Haltepunkte

Um nun die Robustheit Φ_a eines Fahrplans λ zu berechnen, werden für jeden Haltepunkttyp $h' \in H$' und alle ihn innerhalb des gewählten Taktintervalls (typisch sind Takte von 10 Minuten) bedienenden Fahrten $f \in F_{h'}$ die Inversen der Sicherheitsabstände $\delta_{f,pred(f)}(h',\lambda)$ addiert, und so Strafen für kleine Sicherheitsabstände eingeführt (siehe Formel 6.1). Für jeden Haltepunkttyp h' muss dabei der resultierende Wert mit dem Gewicht $\varphi_{h'}$ des Haltepunkttyps multipliziert werden.

$$\Phi_a(\lambda) = \sum_{h' \in H'} \sum_{f \in F_{h'}} \frac{1}{\delta_{f,pred(f)}(h', \lambda)} * \varphi_{h'} \tag{6.1}$$

Die verkehrsplanerischen Vorgaben sind gegeben als eine Menge R. Um den Lösungskandidaten in Hinblick auf eine planerische Vorgabe $r \in R$ bewerten zu können, wird ein von der Ausgestaltung des Fahrplans λ abhängiger Erfülltheitsgrad $\rho_r(\lambda) \in \{1, 2, 3, \infty\}$ eingeführt. Die Werte bilden dabei absteigende Erfüllungsgrade ab, von 1 (komplett erfüllt) über zwei (gut) und drei (akzeptabel) bis zu ∞ (nicht erfüllt). Eine einzelne nicht erfüllte planerische Anforderung führt dazu, dass der Kandidat als ungültig zurück gewiesen wird. Zur Berechnung der Zielfunktionskomponente Φ_b eines Fahrplans λ werden für alle $r \in R$ die Erfüllungsgrade addiert (s. Formel 6.2).

$$\Phi_b(\lambda) = \sum_{r \in R} \rho_r(\lambda) \tag{6.2}$$

Abhängig von der Größe des betrachteten Netzes und der Anzahl der zu beachtenden Vorgaben sind die beiden Bestandteile der Zielfunktion typischerweise nicht direkt miteinander vergleichbar. Daher wird ein Normalisierungsfaktor σ definiert, der das Verhältnis zwischen theoretisch optimalen Sicherheitsabständen und bestmöglicher Erfüllung aller $r \in R$ abbildet. Die optimalen Sicherheitsabstände $\delta^{opt}_{f,pred(f)}(h')$ zweier Fahrten $pred(f)$ und f an Haltepunkttyp $h' \in H'$ ergeben sich durch äquidistantes Aufteilen des zur Verfügung stehenden Zeitintervalls auf die den Haltepunkttyp bedienenden Linien. Der bestmögliche Erfüllungsgrad ρ^{min}_r einer Anforderung $r \in R$ entspricht unabhängig vom konkreten Fahrplan dem minimalen vom Planer vorgegebenen Wert, typischerweise ist das $\rho^{min}_r = 1$. Hierdurch ergibt sich also ein Normalisierungsfaktor σ wie in Formel 6.3 beschrieben.

$$\sigma = \left(\sum_{h' \in H'} \sum_{f \in F_{h'}} \frac{1}{\delta^{opt}_{f,pred(f)}(h')} \right) / \sum_{r \in R} \rho^{min}_r \tag{6.3}$$

Durch Kombination von $\Phi_a(\lambda)$ und $\Phi_b(\lambda)$ entsteht die durch den Faktor σ normalisierte Zielfunktion $\Phi(\lambda)$ (s. Formel 6.4). Um eine Gewichtung der Bestandteile zu ermöglichen wird der Faktor $0 \leq \alpha \leq 1$ als relatives Gewicht der Erfüllung der planerischen Bedingungen eingeführt.

$$\Phi(\lambda) = (1 - \alpha) * \sum_{h' \in H'} \sum_{f \in F_{h'}} \frac{1}{\delta_{f,pred(f)}(h', \lambda)} * \varphi_{h'} + \alpha * \sigma * \sum_{r \in R} \rho_r(\lambda) \tag{6.4}$$

Eine gültige Lösung hat außerdem einigen Nebenbedingungen zu gehorchen. Restrik-

tion (1) (siehe Formel 6.5) bestimmt, dass jede erste Startzeit μ_i innerhalb des Taktintervalls, von 0 bis $takt - 1$ liegt.

$$\forall i \leq |\lambda| : \ 0 \leq \mu_i < takt \tag{6.5}$$

Restriktion (2) (siehe Formel 6.6) gibt an, dass zwischen zwei Abfahrten f und $pred(f)$ an jedem Haltepunkttyp $h' \in H'$ mindestens eine Minute Zeit vergehen muss. Eine Haltestelle darf also nicht gleichzeitig von zwei Fahrzeugen belegt werden, der Fahrplan muss kollisionsfrei sein.

$$\forall h' \in H' : \ \forall f \in F : \ \delta_{f,pred(f)}(h', \lambda) > 0 \tag{6.6}$$

Zielfunktion und Restriktionen werden in Abbildung 6.3 zusammen gefasst.

Minimiere $\underbrace{(1-\alpha) * \sum_{h' \in H'} \sum_{f \in F_{h'}} \frac{1}{\delta_{f,pred(f)}(h', \lambda)} * \varphi_{h'}}_{Robustheit} + \underbrace{\alpha * \ \sigma * \sum_{r \in R} \rho_r(\lambda)}_{Planerische\ Vorgaben}$

mit $\sigma = \left(\sum_{h' \in H'} \sum_{f \in F_{h'}} \frac{1}{\delta^{opt}_{f,pred(f)}(h')} \right) / \sum_{r \in R} \rho_r^{min}$

und $0 \leq \alpha \leq 1$

Unter

(1) $0 \leq \mu_i < takt$ $\quad \forall i \leq |\lambda|$

(2) $\delta_{f,pred(f)}(h') > 0$ $\quad \forall h' \in H' : \forall f \in F_{h'}$

Figure 6.3.: Übersicht über das Optimierungsmodell

Um von Verkehrsplanern vorgetragene Ansprüche an einen Fahrplan für die Optimierung berücksichtigen zu können, muss eine gewisse Formalisierung stattfinden. Von Walter wurden in Zusammenarbeit mit Planern der KVB, wie in ihrer vom Autor betreuten Diplomarbeit [81] beschrieben, die folgenden sieben Typen von Ansprüchen an einen Fahrplan identifiziert:

- *Startzeiten:* Für eine Linie l werden für einen Haltepunkt h die gewünschten Startzeiten festgelegt. Hierzu wird vom Verkehrsplaner für jede mögliche Startminute i

mit $0 \leq i < takt$ ein Wert aus $\{1, 2, 3, \infty\}$ vergeben.

- *Abstände:* Die Abfahrten einer Linie l_1 an einem Haltepunkt h_1 dürfen ausschließlich in einem gewissen zeitlichen Abstand zu den Abfahrten der Linie l_2 an Haltepunkt h_2 abfahren. Hier werden ebenfalls für jeden mögliche Abstand i in Minuten mit $0 \leq i < takt$ Werte aus $\{1, 2, 3, \infty\}$ vergeben.

- *Verstärkerfahrten:* Die Fahrten der Linie l_1 verstärken die Linie l_2 in Abschnitten mit hoher Nachfrage. Um beide Linien gleichmäßig auszulasten, sollten ihre Fahrten daher das Intervall zwischen zwei Fahrten von l_2 halbieren.

- *Eingleisige Strecken:* Um Kollisionen zu vermeiden, muss zwischen zwei gegengerichtet in einen eingleisigen Streckenabschnitt einfahrenden Fahrzeugen mindestens ein zeitlicher Abstand von t Minuten vorgesehen sein (siehe Abbildung 6.4).

- *Wendezeiten:* Nach der Abfahrt vom letzten Haltepunkt der Linienvariante l_2 muss das Fahrzeug wenden. Es müssen daher mindestens t Minuten vergehen, bis das Fahrzeug den Betrieb am ersten Haltepunkt der l_2 entgegen gerichteten Linienvariante l_1 aufnehmen kann.

- *Umsteigeverbindungen:* Fahrten der Linie l_1 dürfen einen Haltepunkt h erst eine festgelegte Anzahl von Minuten nach der Abfahrt von l_2 bedienen. So soll den Passagieren von l_2 das zeitnahe Umsteigen nach l_1 ermöglicht werden.

- *Rendezvous-Verbindungen:* Fahrten mehrerer Linien warten mit ihren Abfahrten von mehreren, meist zur selben Station gehörenden Haltepunkten aufeinander. Diese Vorgaben gelten typischerweise für einige Anschlüsse an zentrale Stationen in den Nachtstunden, so sollen lange Wartezeiten für die Fahrgäste vermieden werden. Da im Projekt CATS vorrangig die Hauptverkehrszeiten betrachtet werden, wurden Rendezvous-Verbindungen bislang nicht modelliert.

Im nächsten Schritt soll demonstriert werden, dass sich alle Vorgabetypen in Abstands- und Startzeitvorgaben überführen lassen, die Betrachtung dieser beiden einfachen Typen also ausreicht.

- *Verstärkerfahrten:* Hier unterstützen Fahrzeuge der Linie l_2 eine Linie l_1, indem sie eine gemeinsame Teilstrecke ab einem bestimmten Haltepunkt im zeitlichen Abstand von $\frac{takt}{2}$ zu l_1 befahren. Eine Verstärkerfahrtvorgabe kann daher durch eine Abstandsvorgabe abgebildet werden, die einen Abstand von $\frac{takt}{2}$ von l_1 zu l_2 festlegt.

- *Eingleisige Strecke:* Es sei ein eingleisiger Streckenabschnitt zwischen zwei Haltepunktpaaren (h_1, h_4) und (h_2, h_3) mit einer Überfahrzeit von t Minuten gegeben (siehe Abbildung 6.4). Eine von h_3 abfahrende Fahrt v_2 der Linie l_1 muss dann mit der Einfahrt in den Streckenabschnitt mindestens t Minuten warten, nachdem eine Fahrt der Linie l_2 an Haltepunkt h_1 abgefahren ist. Dies kann durch eine Abstandsvorgabe erreicht werden. Um einen eingleisigen Streckenabschnitt komplett abzudecken, müssen zwischen dort fahrenden Linien jeweils entsprechende Abstandsvorgaben modelliert werden.

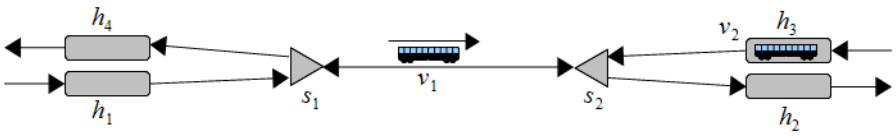


Abbildung 6.4.: Situation an eingleisigen Streckenabschnitten

- *Wendezeiten:* Eine Wendezeitvorgabe kann direkt in eine Abstandsvorgabe überführt werden. Diese legt die Mindestzeit fest, die zwischen der Abfahrt am letzten Haltepunkt h_2 der betroffenen Linie l_2 vergehen muss, bevor die Fahrt der entgegen gerichteten Linie l_1 am Haltepunkt h_1 beginnen darf.

- *Umsteigeverbindungen:* Hier wird durch eine Abstandsvorgabe sicher gestellt, dass Linie l_1 eine angemessene Zeit nach der Linie l_2 von örtlich benachbarten Haltepunkten abfährt. Die Passagiere von l_2 können so nach l_1 umsteigen.

- *Rendezvous-Verbindungen:* Zur Abbildung von Rendezvous-Verbindungen muss ein Ring aus Abstandsvorgaben über alle beteiligten Linien definiert werden. Sei angenommen, dass bei drei beteiligten Linien Linie l_1 einige Minuten nach l_2 abfährt, l_2 einige Minuten nach l_3, l_3 wiederum nach l_1. Wie man leicht sieht, führt dieses Vorgehen unter Annahme der in den Hauptverkehrszeiten üblichen kurzen Haltezeiten zu einem Deadlock. Um einen gültigen Fahrplan unter Einhaltung von Rendezvous-Vorgaben zu generieren, wird mit der durch ÖPNV5 definierten Relation ORT_FRT_HZT gearbeitet, die notwendige längere Standzeiten für individuelle Fahrten an den beteiligten Haltepunkten vorschreibt. Der Deadlock kann damit z.B. gelöst werden, indem l_1 vor l_3 in die Station *ein*fährt, aber nach l_3 *ab*fährt, so dass ein Passagierwechsel in beide Richtungen möglich ist.

Zur Vereinfachung werden im Rest dieses Kapitels nur Abstands- und Startzeitvorgaben betrachtet. Diese können in Tabellenform erfasst (siehe als Beispiel hierzu die Tabellen 6.2 und 6.3 in Abschnitt 6.4.1.1) und bei der Optimierung berücksichtigt werden.

6.2.4. Konfiguration des Genetischen Algorithmus

Genetische Algorithmen sind Metaheuristiken aus dem Bereich der Evolutionären Algorithmen zur Lösung kombinatorischer Optimierungsprobleme. Hier wird versucht, durch die algorithmische Nachbildung der natürlichen Mechanismen der Evolution (wie z.B. Selektion, Rekombination und Mutation) eine Verbesserung der Angepasstheit eines Lösungskandidaten an die jeweilige Situation (hier definiert durch die Zielfunktion und die Nebenbedingungen) zu erreichen. Die am besten angepassten Individuen überleben in einer zunächst meist zufällig erzeugten Population, pflanzen sich fort und geben so die günstigen Ausprägungen an folgende Generationen weiter. Durch geeignet gestaltete Rekombination und Mutation werden neue Ausprägungen in den Genpool eingebracht. Im Laufe der Generationen finden sich immer besser an das gegebene Szenario angepasste Individuen, es werden also bessere Lösungskandidaten für das formulierte Optimierungsproblem erzeugt. Für weitere Einführungen in die Verfahren und Begrifflichkeiten Evolutionärer und Genetischer Algorithmen siehe z.B. Dreo, et al. in [14], S. 75ff. oder Schöneburg, Heinzmann und Feddersen in [60].

Seinem Wesen als Metaheuristik gemäß muss das Verfahren für konkrete Anwendungsfälle entsprechend angepasst werden. Hierzu werden die Startzeiten μ_i der jeweils ersten Fahrt jeder Linie i zu einem Individuum $\lambda = (\mu_1, ..., \mu_n)$ kombiniert. Alle anderen Fahrten folgen dann entsprechend des Taktintervalls. Zur Messung der Fitness eines Individuums kann direkt die oben beschriebene Zielfunktion $\Phi(\lambda)$ eingesetzt werden.

Die Anwendung generiert eine Startpopulation aus zufällig erzeugten Individuen, die hinsichtlich Kollisionsfreiheit und Einhaltung der planerischen Vorgaben valide sind. Um die Komplexität zu reduzieren, wird im Algorithmus eine einfache Turnierselektion (siehe Dreo, et al. in [14], S. 88ff.) und als Kombinationsverfahren ein Zwei-Punkt-Crossover (siehe Schöneburg, Heinzmann und Feddersen in [60], S. 198f.) angewandt. Als Methode zum Einbringen der Kind-Individuen in die Population wird ein *steady state*-Verfahren (siehe Dreo, et al. in [14], S. 90f.) verwendet.

Die Software stellt mehrere Mutationsverfahren zur Verfügung (siehe Schöneburg, Heinzmann und Feddersen in [60], S. 200ff.). Für die unten beschriebenen Experimente wird eine zufällige Minimalmutation benutzt, bei der Startzeiten lediglich um eine Minute verschoben werden. Dieses Verfahren hat sich bei Evaluierungen als am besten geeignet erwiesen (für ausführlichere Beschreibungen von Details der Implementierung

siehe nochmals Franz in [15]).

Nach dem Ende jedes Optimierungslaufs wird ein Hillclimber-Algorithmus auf den besten gefundenen Lösungskandidaten angewandt, um so ggf. seinen Zielfunktionswert weiter zu verbessern.

6.2.5. Konfiguration des Branch-and-Bound-Solvers

Die Branch-and-Bound-Methode ist eine allgemeine Strategie zur Lösung kombinatorischer Optimierungsmethoden mit ganzzahligen Variablen. Der Lösungsraum wird hier als Baum angesehen, mit den inneren Knoten als Teillösungen des gegebenen Problems. Die Baumwurzel wird als Teillösung angesehen, bei der keine der Lösungsvariablen gesetzt ist. Mit dem Aufstieg durch die Ebenen des Baums („Branching") wird jeweils eine zusätzliche Lösungsvariable mit einem zulässigen Wert belegt. Beim Durchsuchen des Baumes wird für den aktuell betrachteten Teilbaum unter Berücksichtigung der bereits gesetzten Lösungsvariablen eine lokale untere Schranke für die beste noch erreichbare Lösung berechnet. Diese wird mit der durch die beste bereits gefundene Lösung definierte obere Schranke verglichen. Bietet der untersuchte Teilbaum nicht die Möglichkeit, eine bessere Lösung zu finden, so wird er verworfen („Bounding"). Durch die möglichst präzise Abschätzung der Schranken können große Teile des Suchbaums früh ausgeschlossen, und die Laufzeit so massiv beeinflusst werden. Für eine weiter gehende Einführung in die Branch-and-Bound-Strategie siehe z.B. Dakin in [11].

Wie beschrieben wird im geschilderten Fall die beste vom Genetischen Algorithmus gefundene Lösung dem Branch-and-Bound-Löser als globale obere Schranke übergeben, um einen Kaltstart zu vermeiden und so die nötige Rechenzeit zu verkürzen.

Um möglichst früh Teilbäume ausschließen zu können, muss die Zielfunktion angepasst werden. Dazu wird für jeden zu untersuchenden inneren Knoten die Menge der Linien L geteilt in die Menge $\hat{L}$ der für diesen Knoten bereits gesetzten Linien und die Menge der noch ungesetzten Linien $\tilde{L}$. Die Menge der Haltepunkttypen H' wird in die disjunkten Mengen $\hat{H}$ und $\tilde{H}$ aufgeteilt. Die Menge $\hat{H}$ enthält dabei die Haltepunkttypen, die ausschließlich von bereits gesetzten Linien aus $\hat{L}$ bedient werden, die Menge $\tilde{H}$ die Haltepunkttypen, die auch oder ausschließlich von noch nicht gesetzten Linien aus $\hat{\tilde{L}}$ angefahren werden. Die Menge der Planungsvorgaben R wird in die disjunkten Mengen $\hat{R}$ und $\tilde{R}$ aufgeteilt. Die Menge $\hat{R}$ enthält dabei die Vorgaben, die ausschließlich von bereits gesetzten Linien abhängig sind, die Vorgaben in Menge $\tilde{R}$ sind von noch nicht gesetzten Linien abhängig. In der für die Anwendung im Branch-and-Bound-Algorithmus modifizierten Zielfunktion $\Phi'(\lambda)$ (siehe Formel 6.7) repräsentiert $\tilde{\delta}_{f,pred(f)}(h', \lambda)$ den bestmöglichen Sicherheitsabstand unter Berücksichtigung der bereits gesetzten Linien $\hat{L}$. Der

Wert ρ_r^{min} steht für den wie bereits beschrieben berechneten bestmöglichen Erfüllungsgrad der planerischen Vorgabe $r \in R$.

$$\begin{aligned}\Phi'(\lambda) &= (1-\alpha) * \left(\sum_{h' \in \hat{H}} \sum_{f \in F_h} \frac{1}{\delta_{f,pred(f)}(h',\lambda)} * \varphi_{h'} + \sum_{h' \in \tilde{H}} \sum_{f \in F_h} \frac{1}{\tilde{\delta}_{f,pred(f)}(h',\lambda)} * \varphi_{h'} \right) \\ &+ \alpha * \left(\sum_{r \in \hat{R}} \rho_r(\lambda) + \sum_{r \in \tilde{R}} \rho_r^{min} \right) * \sigma \end{aligned} \tag{6.7}$$

Weitere Details zur Implementierung der adaptierten Optimierungsverfahren finden sich bei Franz in [15].

6.3. Parallele Simulation von Stadtbahnfahrplänen

6.3.1. Hintergrund und Vorgehen

Viele Simulationsmodelle für schienengebundene Verkehrssysteme bilden Eisenbahnnetze für Fern- und Regionalverkehr ab (siehe z.B. Middelkoop und Bouwman in [49] oder Nash und Huerlimann in [52]). Diese Systeme haben Gemeinsamkeiten mit Stadtbahnnetzen, wie z.B. beim Passagieraustausch oder dem Manövrierverhalten der Fahrzeuge, unterscheiden sich aber in wichtigen Aspekten. So sind Stadtbahnnetze oft gemischt, d.h. die Fahrzeuge bewegen sich sowohl über exklusiv genutzte Untergrundstrecken als auch zusammen mit anderen Verkehrsteilnehmern auf Straßenniveau, müssen also deren Verhalten beachten und sich gemeinsamen Verkehrsregelungsstrategien unterwerfen. Daraus folgend ist das Verhalten von Stadtbahnfahrzeugen eine Kombination aus den Eigenschaften von Eisenbahnen und Individualverkehr. Eine einfache Übernahme von Methoden der Eisenbahnsimulation ist daher nicht zielführend.

Unter Berücksichtigung der Gemeinsamkeiten mit Individualverkehrssystemen hat Joisten in ihrer Diplomarbeit [29] ein auf dem von Nagel und Schreckenberg in [51] verwendeten Zellularautomatenansatz basierendes Modell für Stadtbahnnetze entwickelt, das wie von Lückemeyer und Speckenmeyer in [39] beschrieben an dem für Zellularautomaten typischen hohen Aggregationsgrad leidet. Lückemeyer entwickelte darauf in seiner Dissertation [38] ein ereignisorientiertes Simulationsmodell, das einige dieser Nachteile wie von Lückemeyer und Speckenmeyer in [39] beschrieben beseitigt.

Um verbleibende Ungenauigkeiten zu beseitigen, wurde von Lückerath in seiner vom Autor betreuten Diplomarbeit [40] eine erste Version eines zunächst sequentiellen Si-

mulationsmodells vorgestellt, das durch Lückerath, Ullrich und Speckenmeyer in [41] und von Ullrich, et al. in [74] erweitert und verfeinert wurde. Dieses Modell wurde im Rahmen dieser Arbeit mithilfe des *cellforce*-Frameworks parallelisiert und soll im Folgenden kurz beschrieben werden. Dazu wird zuerst auf die Modellbildung eingegangen, wobei die einzelnen Teilmodelle für das physische Stadtbahnnetz, die Fahrzeuge und die Linien- und Fahrpläne vorgestellt werden. Im Anschluss werden die im Modell genutzten stochastischen Elemente dargestellt. Darauf folgt eine kurze Beschreibung der für die Implementierung des Modells notwendigen Arbeiten.

6.3.2. Modellbildung

6.3.2.1. Das physische Netz

Das physische Stadtbahnnetz wird modelliert als gerichteter Graph mit Haltepunkten, Gleisabschnitten und Weichen als Knoten. Jeder dieser Knoten verwaltet dabei die aktuell auf ihm befindlichen Fahrzeuge. Verbindungen zwischen den Bestandteilen des Netzes werden als gerichtete Kanten abgebildet. Die Abbildungen 6.5 und 6.6 zeigen beispielhaft einen Ausschnitt aus einem Stadtbahnnetz und dessen Übertragung in das Modell.

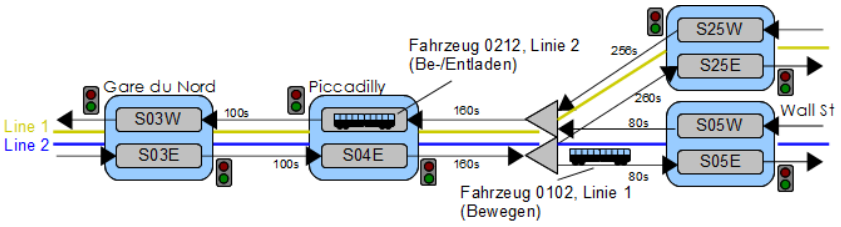


Abbildung 6.5.: Beispielhafter Ausschnitt eines Stadtbahnnetzes

Ein Haltepunkt verwaltet zu jedem Zeitpunkt maximal ein Fahrzeug. Er hält als Attribute u.a. eine Bezeichnung, und Verweise auf die übergeordnete Station und die ein- und ausgehenden Streckenabschnitte.

Streckenabschnitte sind die einzigen Knotentypen, die - eine hinreichende Länge vorausgesetzt - gleichzeitig mehrere Fahrzeuge verwalten können. Im Kölner KVB-Netz können einige Bereiche bidirektional genutzt werden. Hier muss Vorsorge getroffen werden, dass sie nicht gleichzeitig in verschiedene Richtungen befahren werden. Da das ÖPNV5-Modell keine bidirektionalen Abschnitte vorsieht, werden diese als zwei gegenläufige, gekoppelte Strecken abgebildet. Die Reservierung des einen Abschnitts führt zur Blockierung des anderen.

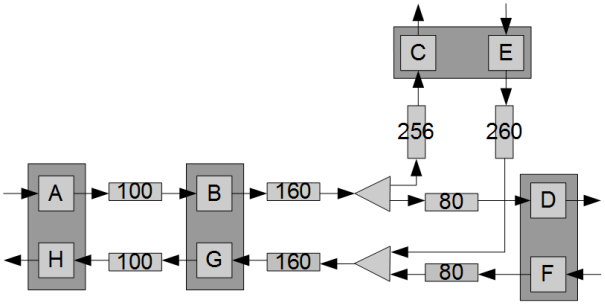


Abbildung 6.6.: Beispielgraph zur Repräsentation eines Stadtbahnnetzes

Strecken verfügen über Attribute wie Länge, geplante Überfahrzeiten, individuelle Höchstgeschwindigkeiten und eine Liste verwalteter Lichtsignalanlagen. Die Positionen der Lichtsignalanlagen auf dem Gleis sind in der Datenbank durch einen Offset zu dessen Start definiert. Ihr Zustand a_i zum Zeitpunkt t wird, wie in Formel 6.8 beschrieben, bestimmt durch die Dauer der Ampelphasen t_{red} und t_{green} (mit $t_{cycle} = t_{red} + t_{green}$) und den zufällig gesetzten Zeitpunkt des ersten Phasenübergangs t_{init} von grün nach rot (siehe hierzu Abschnitt 6.3.2.4).

$$a_i(t) = \begin{cases} red & falls \quad (t - t_{init}(i)) \mod t_{cycle} < t_{red} \\ green & sonst \end{cases} \tag{6.8}$$

Weichen sind als Transferpunkte modelliert, haben also keine Ausdehnung, sondern werden in Nullzeit überfahren. Für die Überfahrt muss die Bahn allerdings eine vorgeschriebene Höchstgeschwindigkeit einhalten. Weichen müssen vor der Überfahrt erfolgreich reserviert und nach dem Erreichen eines Sicherheitsabstands wieder freigegeben werden.

6.3.2.2. Fahrzeuge

Das Teilmodell zur Abbildung der Fahrzeuge ist gemäß dem ereignisbasierten Paradigma aufgebaut, wie es von Banks, et al. in [5], S. 106ff. beschrieben wird. Dabei ändern die Fahrzeuge ihren Zustand wie bereits in Abschnitt 2.1 beschrieben durch Simulationsereignisse bestimmten Typs, die an diskreten Simulationszeitpunkten stattfinden. Die Ereignisse werden einer nach Zeitstempel geordneten Prioritätswarteschlange entnommen und verarbeitet. Bei der Verarbeitung kann sich der Systemzustand ändern, es können

Folgeereignisse erzeugt werden. Das Modell berücksichtigt 14 Typen von Simulationsereignissen (siehe Tabelle 6.1).

Nr.	Ereignistyp	Beschreibung
01	TRIP_START	Das Fahrzeug beginnt eine neue Fahrt
02	TRIP_END	Das Fahrzeug beendet die aktuelle Fahrt
03	TRAM_STAND	Das Fahrzeug beginnt eine Standphase
04	TRAM_MOVE	Das Fahrzeug beginnt eine Bewegungsphase
05	TRAM_ACCELERATE	Das Fahrzeug beginnt eine Beschleunigungsphase
06	TRAM_BRAKE	Das Fahrzeug beginnt zu bremsen
07	TRAM_EMERG_BRAKE	Das Fahrzeug führt eine Notbremsung aus
08	TRAM_CRASH	Das Fahrzeug verunfallt, die Simulation meldet eine Ausnahme und bricht ab
09	TRAM_PASS_EXCHANGE	Das Fahrzeug beginnt mit dem Passagierwechsel
10	SWITCH_LOCK	Eine Weiche wird zur Überfahrt reserviert
11	SWITCH_FREE	Die Weichenreservierung wird aufgehoben
12	BIDIR_TRACK_LOCK	Ein eingleisiger Streckenabschnitt wird reserviert
13	BIDIR_TRACK_FREE	Die Streckenreservierung wird aufgehoben
14	TRAM_TRANSFER	Das Fahrzeug wird zum nächsten Simulationsknoten transferiert

Tabelle 6.1.: Simulationsereignistypen zur Simulation von Stadtbahnnetzen

Das Fahrverhalten der einzelnen Bahnen wird durch den Fahrzeugtyp bestimmt. Exemplarisch soll hier das Fahrverhalten des im Kölner Stadtbahnnetz eingesetzten Fahrzeugtyps Vossloh-Kiepe K4000 gezeigt werden, wie es von Lückerath in [40] bestimmt wurde (siehe Formel 6.9).

$$v(t) = \begin{cases} 0 & falls\ 0 \leq t < 1 \\ \frac{14*t}{3} - \frac{10}{3} & falls\ 1 \leq t < 8 \\ 35,3322 * \sqrt[3]{t} - 36,6645 & falls\ 8 \leq t \leq 36 \\ 80 & falls\ 36 < t \end{cases} \tag{6.9}$$

Als Grundlage hierfür wird das von Vossloh-Kiepe heraus gegebene Datenblatt [80] genutzt, das von mit Passagieren beladenen Fahrzeugen ausgeht. Für eine Übersicht über den Funktionsverlauf siehe Abbildung 6.7. Der Beschleunigungsverlauf kann im Wesentlichen in die beiden gezeigten Bereiche von null bis acht Sekunden und von neun bis 36 Sekunden aufgeteilt werden, die nominelle Höchstgeschwindigkeit von $80\,km/h$ ist bei durchgängiger Beschleunigung nach 36 Sekunden erreicht. Im Datenblatt wird ein

durchschnittliches Beschleunigungsvermögen von $1,3\,m/sec^2$ und ein Bremsvermögen von $1,4\,m/sec^2$ angegeben. Für Bremsen in Notsituationen gilt ein Bremsvermögen von $3,0\,m/sec^2$.

Um kleinere Störungen wie z.B. durch laubbedeckte Gleise oder verkehrsbehindernd haltende PKW abzubilden, werden eine stochastische Trödelwahrscheinlichkeit $0 \leq p_d \leq 1$ und ein Trödelfaktor $d \geq 1$ modelliert. Der Wert p_d gibt an, mit welchen Wahrscheinlichkeit ein Beschleunigungsvorgang „vertrödelt" werden soll, d.h. sich zeitlich um den angegeben Faktor d verlängert. Genaueres hierzu findet sich in Abschnitt 6.3.2.4.

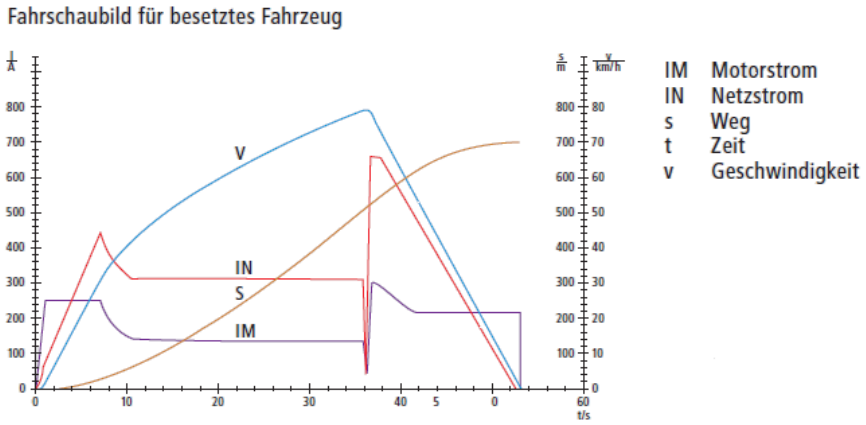


Abbildung 6.7.: Fahrverhalten der Vossloh-Kiepe K4000 (Quelle: Vossloh Kiepe in [80])

6.3.2.3. Linien- und Fahrpläne

Die von den Fahrzeugen zu bedienenden Linienvarianten beinhalten, wie in Abschnitt 6.1.1 beschrieben, eine geordnete Liste von Haltepunkten, die in entsprechender Reihenfolge anzufahren sind. Da das ÖPNV5-Modell keine Informationen zu zwischen den abzufahrenden Haltepunkten liegenden Weichen enthält, muss für jede Linienvariante die Richtungswahl an auftretenden teilenden Weichen entweder vor oder während der Durchführung des Simulationslaufs dynamisch berechnet werden.

Wie bereits in 6.1.1 beschrieben, ist jedem zu simulierenden Fahrzeug eine der Datenbank entnommene Folge von Fahrten zugeordnet, sein Umlauf $u = (f_1, ..., f_n)$. Jede Fahrt wiederum ordnet eine Linienvariante einer geplanten Startzeit zu. Die Vereinigungsmenge aller Umläufe des betrachteten Betriebstags bildet den Fahrplan λ.

Durch die Kombination von durch Weichen erweiterten Linienverläufen und den durch den Fahrplan festgelegten Fahrten ist jedem eingesetzten Fahrzeug für jeden Zeitpunkt im Betriebstag eine geplante Position zugeordnet. Durch Vergleich dieser geplanten Posi-

tionen mit den in den Simulationsläufen beobachteten kann die Verspätung der Fahrzeuge gemessen werden.

6.3.2.4. Randomisierung

Das Simulationsmodell enthält einige randomisierte Elemente: Die Phasenübergänge der Lichtsignalanlagen, die Passagierwechselzeiten an Haltepunkten und das „Trödeln" beim Beschleunigen der Bahnen.

Die real verwendeten Umschaltstrategien der *Lichtsignalanlagen* stehen in der Datenbasis nicht zur Verfügung, sind aber sehr wahrscheinlich vom dynamisch auftretenden Verkehr abhängig. Im Modell werden Startzustände und Umschaltzeitpunkte daher als zufällig angenommen und mittels eines Zufallszahlengenerators erzeugt.

Die Phasendauern t_{red} und t_{green} werden in der Standardeinstellung mit je 30 Sekunden als konstant angenommen und bilden zusammen die Zykluszeit t_{cycle} (siehe Formel 6.10).

$$t_{cycle} = t_{red} + t_{green} \tag{6.10}$$

Um hier für jede Lichtsignalanlage i mit einem einzelnen zufällig erzeugten Wert auszukommen, wird der Zeitpunkt des ersten Übergangs $t_{init}(i)$ von grün nach rot mit Formel 6.11 auf einen einer Gleichverteilung entnommenen, künstlichen Zeitpunkt vor Beginn des Simulationslaufs gelegt. Somit ist für jede Lichtsignalanlage für jeden Zeitpunkt der Simulation die Ampelphase festgelegt.

$$t_{init}(i) = -1 * rand(0, t_{cycle}) \tag{6.11}$$

Die *Verteilung der Passagierwechseldauern* bildet ab, dass unabhängig von der Anzahl der ein- und aussteigenden Fahrgäste eine vom Fahrzeugtyp abhängige Mindestzeit zum Öffnen und Schließen der Türen benötigt wird. In der Realität ist die Zahl der Passagiere abhängig vom betrachteten Haltepunkt, Wochentag und Uhrzeit. Der Durchsatz der zu- und aussteigenden Passagiere ist wiederum vom Fahrzeugtyp und seinen Charakteristika wie Türbreite, vorhandener Standraum nahe der Türen, Gangbreite, etc., abhängig. Zur genaueren Abbildung können ebenso die Anzahl der bereits im Fahrzeug vorhandenen Passagiere verwertet werden.

Solche Ergänzungen sind im bestehenden Modell sehr einfach vorzunehmen, überschreiten aber die Grenzen der bislang verfügbaren Datenbasis. Daher arbeitet das beschriebene Modell mit einer Funktion (zuerst von Lückemeyer in [38] beschrieben), die dem Maximum der vom aktuellen Fahrzeugtyp v abhängigen minimalen Ladezeit m_v und dem Ergebnis einer vom Typ des Fahrzeugs v und dem betrachteten Haltepunkt h abhängigen

Dreiecksverteilung (siehe Banks, et al in [5], S. 182ff) mit den Parametern $a_{h,v}$, $b_{h,v}$ und $c_{h,v}$ entspricht (siehe Formel 6.12 und Abbildung 6.8).

$$l_{h,v} = max(m_v, tri(a_{h,v}, b_{h,v}, c_{h,v})) \tag{6.12}$$

Mangels genauerer Daten berechnet das Modell die Ladezeit $l_{h,v}$ in Sekunden mit dem Parametersatz $\forall h \in H : \forall v \in V : a_{h,v} = 0$, $b_{h,v} = 30$, $c_{h,v} = 15$, $m_v = 12$.

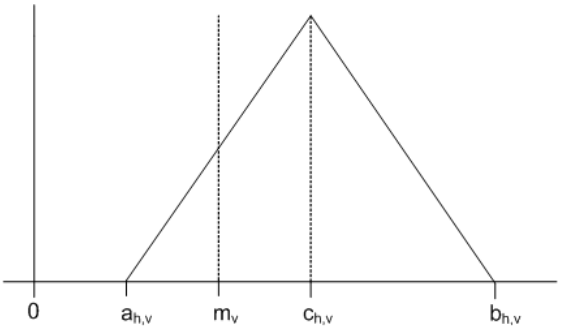


Abbildung 6.8.: Dichte der zugrunde liegenden Dreiecksverteilung

Trödelwahrscheinlichkeit und -faktor können abhängig von Gleis und Fahrtzeitgruppe (Tagesfahrten und Nachtfahrten, vorgegeben durch ÖPNV5) individuell modelliert werden. Als Standardeinstellung wird für die simulierten Tagesfahrten eine Trödelwahrscheinlichkeit von $p_d = 0,3$ und ein Trödelfaktor von $d = 1,3$ angenommen, für den wenig dichten Nachtverkehr wird $p_d = 0,0$ gesetzt.

6.3.3. Implementierung der Simulationsanwendung

Das beschriebene Modell ist auf das Framework *cellforce* abgestimmt und kann daher relativ leicht auf dessen Strukturen abgebildet werden. Eine ausführbare Simulationsanwendung wird erstellt, indem wie in Abschnitt 4.3 beschrieben die Klassen *SimManager*, *Core* und *Message* angepasst werden.

Der Ablauf der Simulationsanwendung beginnt mit einer Instanzierung der von *SimManager* erbenden Klasse *ParSiVaL* (ParSiVaL steht hier für „Parallele Simulation von Verkehr auf Liniennetzen"). Diese Klasse liest u.a. die angegebenen Modelldaten zu Stadtbahnnetz und Fahrplan aus der Datenbank, erstellt daraus Objekte vom Typ *ParSiVaL-Core* und führt im Anschluss die gewünschte Zahl von Simulationsläufen aus. Die Anwendung erhebt während der Läufe die Auslastungs- und Verspätungsdaten und schreibt sie in Textdateien.

Die Klasse *ParSiVaLCore* erbt von *Core* und implementiert das Verhalten des Modells. Um die in Abschnitt 4.2.4 beschriebene Möglichkeit des Teilens und Verschmelzens von Knoten zu nutzen, kann eine Instanz von *Core* mehrere, auch unterschiedlich geartete Modellabschnitte aufnehmen.

Die zu simulierenden Fahrzeuge werden als Instanzen der auf *Message* aufbauenden Klasse *Tram* realisiert. Umgesetzt ist bislang nur das Fahrverhalten des Typs K4000 (siehe Abschnitt 6.3.2.2), da nur für dieses Modell eine ausreichende Datengrundlage zur Verfügung steht. Weitere Fahrzeugtypen können aber durch das Erweitern der abstrakten Basisklassen auf einfache Weise implementiert werden. Für eine detailliertere Beschreibung von Modell und Implementierung sei nochmals auf Lückerath in [40], Lückerath, Ullrich und Speckenmeyer in [41] und Ullrich, et al. in [74] verwiesen.

6.4. Optimierung und Simulation des Kölner Stadtbahnfahrplans

Um das plausible Verhalten der beschriebenen Software-Werkzeuge zu demonstrieren, werden sie im Folgenden auf die Kölner Stadtbahnnetze der Jahre 2001, 2012 und 2020 angewandt. Dazu werden mit dem Optimierer Fahrpläne erzeugt, im Anschluss simuliert und mit Hilfe der Visualisierungswerkzeuge untersucht und verglichen.

Dies kann als Beitrag zur funktions- und theoriebezogenen Validierung des Simulationsmodells angesehen werden. Eine ergebnisorientierte Validierung ist mangels Vergleichsdaten zur real auftretenden Verspätung im Rahmen dieser Arbeit nicht möglich. (Für eine Übersicht über die verschiedenen Validierungsverfahren siehe Liebl in [35], S. 206ff.)

Obwohl die Untersuchungen im Rahmen dieser Arbeit notgedrungen oberflächlich bleiben müssen, sind einige allgemeine Aussagen über die zugrunde liegenden Verkehrsnetze möglich. Inbesondere für das geplante Netz von 2020 ist interessant, ob sich die Entlastungen der Innenstadtstrecken durch die zu erwartende Inbetriebnahme des Nord-Süd-Tunnels auf die beobachteten Verspätungen auswirken. Die Betrachtungen gehen jedoch sicherlich nicht weit genug, um konkrete Handlungsempfehlungen für Verkehrsplaner abzuleiten.

6.4.1. Das Kölner Stadtbahnnetz von 2001

In der CATS-Datenbank liegt eine Repräsentation des Stadtbahnnetzes der Kölner KVB von 2001 vor (Abbildung 6.9 gibt eine Übersicht). Das Netz besteht aus 528 Haltepunkten und 58 Weichen, die durch 584 Gleisabschnitte verbunden sind. Die Gleise haben eine

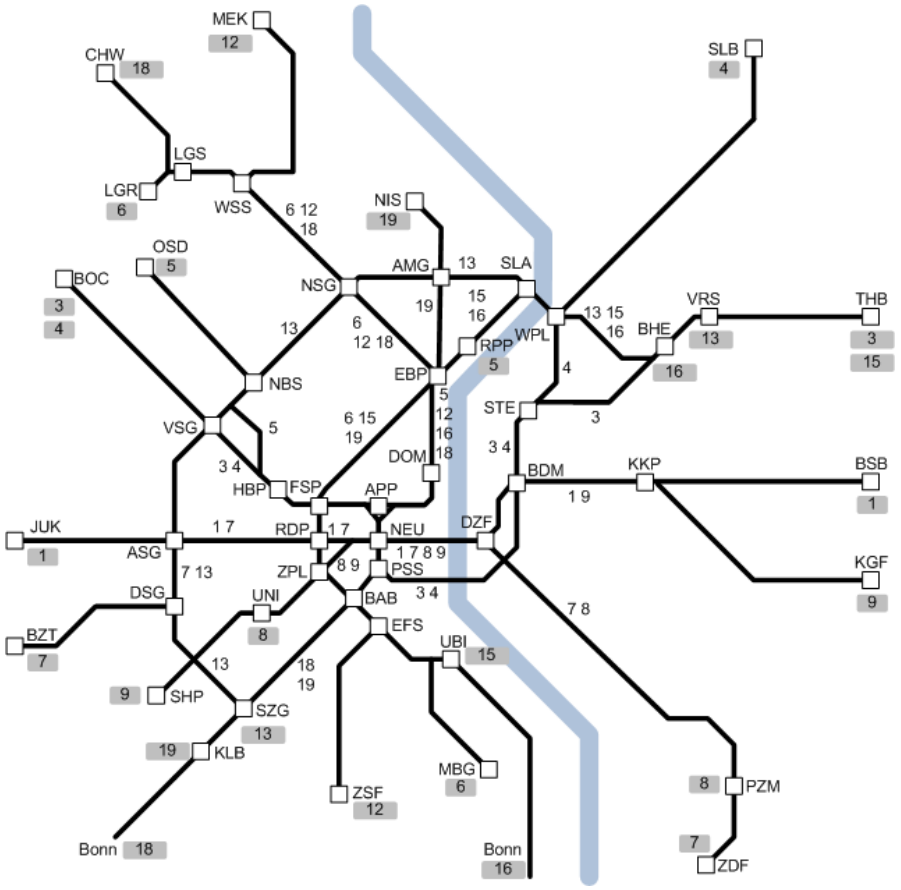


Abbildung 6.9.: Übersicht zum Kölner Stadtbahnnetz im Jahr 2001

Gesamtlänge von 407,4 Kilometern und somit eine durchschnittliche Länge von 697,6 Metern. Die 15 Linien mit 182 Linienrouten des Netzes werden von 178 Fahrzeugen bedient, die pro Betriebstag 2.814 Fahrten ausführen.

Dabei durchqueren alle Linien mit Ausnahme der weite Teile des Gürtels befahrenden Linie 13 das Stadtzentrum. Die Ost-West-Achse wird von Junkersdorf im Westen bis Bensberg im Osten durch die Linie 1 bedient, in der Innenstadt verstärkt durch die Linien 7, 8 und 9. Die Linien 6, 16 und 18 bilden wesentliche Bestandteile der Nord-Süd-Achse. Die Linien 16 und 18 verlassen das Kölner Stadtgebiet nach Süden und binden so die Orte im Vorgebirge und Teile des Bonner Stadtgebiets (hier insbesondere die Innenstadt inklusive des Bonner Hauptbahnhofs) an.

Die Daten zu den beschriebenen Stadtbahnnetzen und Fahrplänen finden sich in der Datenbank auf der beiliegenden CD-ROM (siehe Anhang C).

6.4.1.1. Generieren von Fahrplänen

Im betrachteten Szenario werden beispielhaft die folgenden planerischen Bedingungen an den zu generierenden Fahrplan gestellt:

1. Fahrten der Linie 15 in Richtung Norden sollen an ihrer Startstation Ubierring zwei bis drei Minuten vor der aus Bonn kommenden Linie 16 abfahren. Damit soll erreicht werden, dass an den Haltestellen der Südstadt wartende Passagiere verstärkt die Linie 15 benutzen und so die bereits mit Passagieren aus dem Umland besetzten Fahrten der Linie 16 entlastet werden.

2. Die Linie 1 soll auf der Strecke zwischen Junkersdorf und Brück, Mauspfad im doppelten Takt fahren. Um auf dem dicht befahrenen zentralen Abschnitt fünf Abfahrten pro zehnminütigem Taktzeitraum mit Abständen von zwei Minuten zu erreichen, werden vier- bis sechsminütige Abstände zugelassen. Die Verstärkerlinie wird im Folgenden auch mit 1A bezeichnet.

3. Analog zu 2. sollen sich im Innenstadtbereich Linie 19 und Linie 18 verstärken, ebenso die Linien 3 und 4.

4. Für die Abfahrtszeiten der Linie 18 werden Fenster sowohl am Bonner als auch am Kölner Hauptbahnhof fest gesetzt. Auf diese Art können Anschlüsse zur Deutschen Bahn AG und zu den Stadtbahnlinien der Bonner Verkehrsbetriebe realisiert werden.

5. Für die Endstationen der Linien 1, 3, 4, 5, 6, 7, 8, 9, 12, 13, 15 und 19 werden Mindestzeiten von zwei Minuten für den Richtungswechsel angenommen. Für eine genauere Abbildung muss hier sicherlich zwischen Wendeanlagen wie z.B. westlich der Station Moltkestraße und einfachen Kopfbahnhöfen wie z.B. in Niehl, Sebastianstraße unterschieden werden. Genaue Daten stehen hierfür aber nicht zur Verfügung.

6. Die im Stadtgebiet im Zehn-Minuten-Takt verkehrenden Varianten der Linien 7 und 18 erhalten jeweils Verlängerungen in die Außenbereiche mit einem Takt von 20 Minuten.

Die planerischen Bedingungen 1 bis 5 lassen sich für die Optimierungsläufe in zwei Startzeitvorgaben und 37 Abstandsvorgaben umsetzen (siehe Tabellen 6.2 und 6.3, die Nummerierung der Haltepunkte entspricht der beiliegenden Datenbank). Für die Bedingung 6 werden die äußeren Abschnitte der Varianten 7-FW01, 7-BW02, 18-FW01 und 18-BW02

Nr.	Variante	Haltepunkt	Startzeitprioritäten									
			0	1	2	3	4	5	6	7	8	9
1	18-FW01	606	0	0	0	0	0	0	0	0	1	1
2	18-BW02	203	0	0	0	0	0	0	0	0	1	1

Table 6.2.: Verwendete Startzeitvorgaben für das KVB-Netz 2001

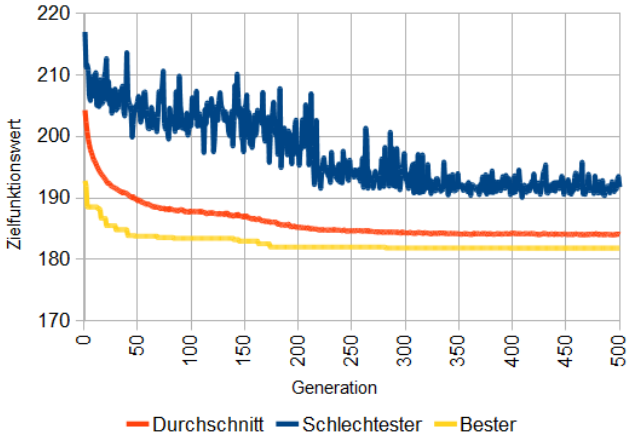


Abbildung 6.10.: Verlauf der Zielfunktionswerte bei der Anwendung des Genetischen Algorithmus (2001)

als Verlängerungen der jeweiligen Kernvarianten mit zwanzigminütigem Takt definiert. Die bei der Optimierung dieses Szenarios berücksichtigten Linienvarianten sind in Tabelle 6.4 angegeben.

Für den Optimierungslauf wird das relative Gewicht der Erfüllung der planerischen Bedingungen α auf 0,5 gesetzt, so dass Robustheit und Erfüllung der planerischen Vorgaben gleich gewichtet werden.

Der Genetische Algorithmus wird mit einer Population von 450 zufällig erzeugten Individuen initialisiert. Der beste Fitnesswert dieser Generation liegt bei 192,789 (Durchschnitt: 204,266, schlechtestes Individuum: 217,005). Der Algorithmus senkt den besten Zielfunktionswert im Laufe von 500 Generationen in ca. 19,5 Minuten Rechenzeit auf 181,467 (Durchschnitt: 181,890, schlechtestes Individuum: 184,399, siehe Abbildung 6.10). Der Branch-and-Bound-Algorithmus verbessert diesen Wert im Laufe von ca. 8,5 Stunden nochmals auf 180,696 und findet 60 optimale Lösungen.

Nr.	Bed.	Variante 1	Variante 2	Hp. 1	Hp. 2	Abstandsprioritäten									
						0	1	2	3	4	5	6	7	8	9
1	1	15-FW01	16-FW07	627	-	0	0	1	1	0	0	0	0	0	0
2	2	1-FW05	1-FW01	1	-	0	0	0	0	1	1	1	0	0	0
3	2	1-BW06	1-BW02	41	-	0	0	0	0	1	1	1	0	0	0
4	3	18-FW01	19-FW03	751	-	0	0	0	0	1	1	1	0	0	0
5	3	19-BW02	18-BW04	795	-	0	0	0	0	1	1	1	0	0	0
6	3	4-FW01	3-FW01	65	-	0	0	0	0	1	1	1	0	0	0
7	3	4-BW02	3-BW02	104	-	0	0	0	0	1	1	1	0	0	0
8	5	1-FW01	1-BW02	32	33	0	0	1	1	1	1	1	1	1	1
9	5	1-BW02	1-FW01	64	1	0	0	1	1	1	1	1	1	1	1
10	5	1-FW05	1-BW06	24	41	0	0	1	1	1	1	1	1	1	1
11	5	1-BW06	1-FW05	64	1	0	0	1	1	1	1	1	1	1	1
12	5	3-FW01	3-BW02	93	94	0	0	1	1	1	1	1	1	1	1
13	5	3-BW02	3-FW01	122	65	0	0	1	1	1	1	1	1	1	1
14	5	4-FW01	4-BW02	152	153	0	0	1	1	1	1	1	1	1	1
15	5	4-BW02	4-FW01	122	65	0	0	1	1	1	1	1	1	1	1
16	5	5-FW01	5-BW02	554	581	0	0	1	1	1	1	1	1	1	1
17	5	5-BW02	5-FW01	216	183	0	0	1	1	1	1	1	1	1	1
18	5	6-FW01	6-BW02	244	257	0	0	1	1	1	1	1	1	1	1
19	5	6-BW02	6-FW01	284	217	0	0	1	1	1	1	1	1	1	1
20	5	7-FW11	7-BW10	317	318	0	0	1	1	1	1	1	1	1	1
21	5	7-BW10	7-FW11	58	7	0	0	1	1	1	1	1	1	1	1
22	5	8-FW57	8-BW56	13	52	0	0	1	1	1	1	1	1	1	1
23	5	8-BW56	8-FW57	435	398	0	0	1	1	1	1	1	1	1	1
24	5	9-FW01	9-BW02	416	417	0	0	1	1	1	1	1	1	1	1
25	5	9-BW02	9-FW01	440	393	0	0	1	1	1	1	1	1	1	1
26	5	12-FW01	12-BW02	468	469	0	0	1	1	1	1	1	1	1	1
27	5	12-BW02	12-FW01	494	441	0	0	1	1	1	1	1	1	1	1
28	5	13-FW01	13-BW02	88	99	0	0	1	1	1	1	1	1	1	1
29	5	13-BW02	13-FW01	542	497	0	0	1	1	1	1	1	1	1	1
30	5	15-FW01	15-BW02	93	94	0	0	1	1	1	1	1	1	1	1
31	5	15-BW02	15-FW01	662	627	0	0	1	1	1	1	1	1	1	1
32	5	16-FW07	16-BW08	86	101	0	0	1	1	1	1	1	1	1	1
33	5	16-BW08	16-FW07	669	620	0	0	1	1	1	1	1	1	1	1
34	5	18-FW03	18-BW04	775	776	0	0	1	1	1	1	1	1	1	1
35	5	18-BW04	18-FW03	807	744	0	0	1	1	1	1	1	1	1	1
36	5	19-FW01	19-BW02	712	713	0	0	1	1	1	1	1	1	1	1
37	5	19-BW02	19-FW01	800	751	0	0	1	1	1	1	1	1	1	1

Table 6.3.: Verwendete Abstandsvorgaben für das KVB-Netz 2001

Linie	Hauptvarianten		Verstärkervarianten	
Linie 1	FW01	BW02	FW05	FW06
Linie 3	FW01	BW02		
Linie 4	FW01	BW02		
Linie 5	FW01	BW02		
Linie 6	FW01	BW02		
Linie 7	FW01*	BW02*	FW11	BW10
Linie 8	FW57	BW56		
Linie 9	FW01	BW02		
Linie 12	FW01	BW02		
Linie 13	FW01	BW02		
Linie 15	FW01	BW02		
Linie 16	FW07	BW08		
Linie 18	FW01*	BW02*	FW03	BW04
Linie 19	FW01	BW02		

* Verlängerung in Außenbereichen, Takt 20 Minuten

Tabelle 6.4.: Verwendete Linienvarianten für das KVB-Netz 2001

6.4.1.2. Vergleich der Fahrpläne

Um den Erfolg des Optimierungsverfahrens zu validieren, werden aus dem Pool der Initialfahrpläne und aus dem Pool der Optimalfahrpläne jeweils zehn Pläne ausgewählt. Mit jedem dieser Fahrpläne werden zehn Simulationsläufe durchgeführt und ausgewertet.

Da die Schaltstrategien für die Signale und Verkehrsampeln nicht bekannt sind, geht das Simulationsmodell wie in Abschnitt 6.3.2.4 beschrieben von Ampelphasen von 30 Sekunden Dauer aus, die zu zufälligen Zeitpunkten erstmals umgeschaltet werden. Damit signifikante Verspätungen auftreten (die sich dann abhängig vom Fahrplan unterschiedlich auswirken), wird eine relativ hohe Trödelwahrscheinlichkeit $p_d = 0,3$ gesetzt. Als Trödelfaktor wird $d = 1,3$ gewählt.

Unter den simulierten Fahrplänen starten die ersten Fahrzeuge bei einer Simulationszeit von 7.00 Uhr. Die Datenerhebung beginnt nach einer Stunde Einpendelzeit um 08.00 Uhr und wird um 16.00 Uhr beendet. Von explizit beschriebenen Ausnahmen abgesehen verkehren alle Linien in diesem Zeitraum im Zehn-Minuten-Takt.

Im Kölner Netz wurden 2001 hauptsächlich Fahrzeuge des Typs K4000 des Herstellers Vossloh Kiepe eingesetzt, für die wie beschrieben einige Daten zum Fahrverhalten zur Verfügung stehen.

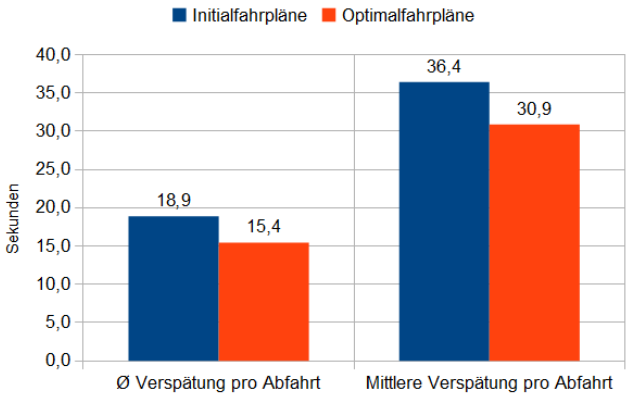


Abbildung 6.11.: Durchschnittliche und mittlere Verspätung pro Abfahrt (2001)

Vergleich allgemeiner Merkmale Unter den Initialfahrplänen liegt die durchschnittliche Verspätung pro Abfahrt bei 18,9 Sekunden, unter den Optimalfahrplänen wurden 15,4 Sekunden gemessen (siehe Abbildung 6.11). Die durchschnittliche Verspätung konnte also um 3,5 Sekunden oder ca. 18,6% gesenkt werden. Die mittlere Verspätung (pünktliche Abfahrten wurden bei der Berechnung außer Acht gelassen) reduziert sich von den initialen zu den optimalen Fahrplänen von 36,4 auf 30,9 Sekunden, also um 5,5 Sekunden oder ca. 15,1%.

Ebenfalls erhoben wurden die Häufigkeitsverteilung der Abfahrtsverspätungen (siehe Abbildung 6.12). Hier zeigt sich, dass die gemäß der Zielfunktion optimalen Fahrpläne im Bereich kleinerer Verspätungen bis zu 40 Sekunden mehr verspätete Abfahrten aufweisen, jedoch deutlich weniger größere Verspätungen (ab 60 Sekunden) beobachtet werden (siehe Abbildung 6.13).

Da auch die initial erzeugten Fahrpläne am selben Haltepunkt einen Mindestabstand von einer Minute für zwei Abfahrten vorsehen, kann sich bessere Robustheit erst bei Verspätungen ab 60 Sekunden auswirken. Erst bei diesen größeren Störungen kann nämlich bei nicht robusten Fahrplänen die Verspätung auf ein nachfolgendes Fahrzeug übertragen werden. Da dies bei der Simulation robuster Fahrpläne wegen der größeren Abstände der Fahrten seltener vorkommt, ist die Zahl der größeren Verspätungen entsprechend geringer. Leicht verspätete Fahrzeuge werden also seltener behindert, einmal vorhandene leichte Verspätungen werden seltener über die Minutenschwelle vergrößert. Dies erklärt die etwas höhere Anzahl von Verspätungen bis zu einer Minute bei den robusten Fahrplänen.

Die Gesamtzahl der um mehr als 60 Sekunden verspäteten Abfahrten pro Simulationstag sinkt von 3.095,6 für die Initialfahrpläne um 987,6 gemessene Verspätungen oder ca.

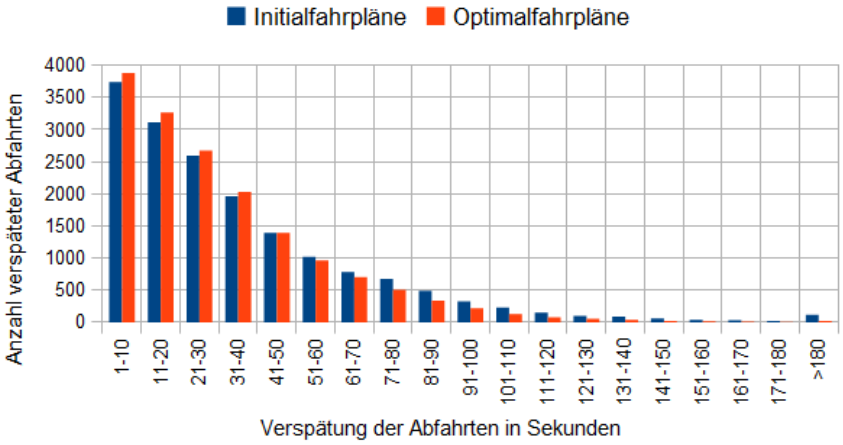


Abbildung 6.12.: Häufigkeitsverteilung der Verspätungen pro Abfahrt (2001)

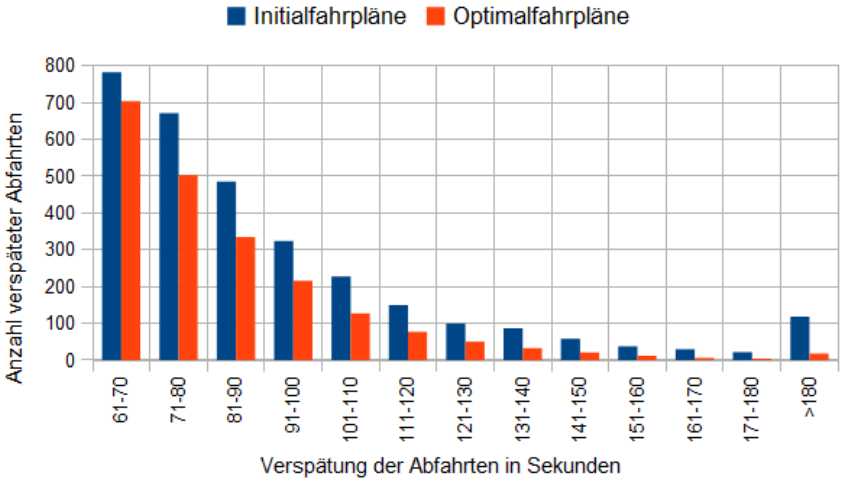


Abbildung 6.13.: Häufigkeitsverteilung der größeren Verspätungen pro Abfahrt (2001)

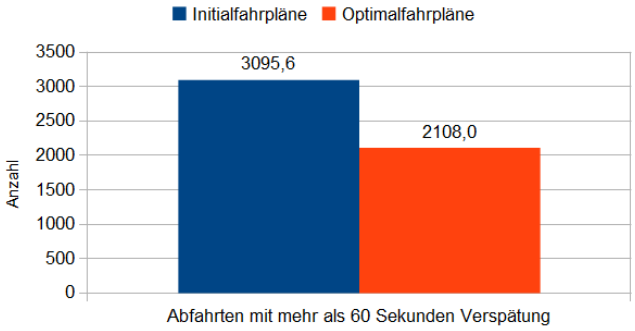


Abbildung 6.14.: Anzahl Abfahrten mit mehr als 60 Sekunden Verspätung (2001)

Richtung	Linie														
	1	1A	3	4	5	6	7	8	9	12	13	15	16	18	19
FW	7	1	4	6	9	7	7	1	3	4	0	6	3	6	7
BW	6	0	0	7	6	6	0	7	7	2	4	9	5	5	9

Tabelle 6.5.: Fahrplan 73 - Optimalfahrplan

31,9% auf 2.108,0 bei den Optimalfahrplänen (siehe Abbildung 6.14). Die Gesamtzahl aller verspäteten Abfahrten sinkt von 16.923,6 um durchschnittlich 602,0 Abfahrten oder ca. 3,6% auf 16.321,6. Die Anzahl der Abfahrten mit mehr als 180 Sekunden Verspätung sinkt sogar um 84,7% von 118,0 auf 18,0.

Die Auswertung bestätigt also, dass unter vom Optimierer als robust bewerteten Fahrplänen in der Simulation durchschnittlich deutlich weniger (und auch geringere) Verspätungen auftreten als bei zufällig erzeugten, typischerweise weniger robusten.

Vergleich zweier Fahrpläne Die bisher beschriebenen Werte sind Durchschnittsberechnungen aus der Simulation von jeweils zehn Optimal- und Initialfahrplänen. Um einzelne Linien näher zu betrachten oder bestimmten Fahrzeugen durch den Simulationstag zu folgen, ist dieses Vorgehen zu grob. Anstatt weiterhin mit dem Durchschnitt über mehrere Fahrpläne zu arbeiten, wird daher aus jedem der beiden Bereiche ein Fahrplan ausgewählt und näher untersucht. Aus der Menge der optimalen Fahrpläne wurde als typischer Vertreter Fahrplan 73 (siehe Tabelle 6.5, Zielfunktionswert: 180,696) ausgewählt, dem Fahrplan 24 (siehe Tabelle 6.6, Zielfunktionswert: 214,714) aus der Menge der recht unterschiedlichen Initialfahrpläne gegenüber gestellt wird.

Für jeden der ausgewählten Fahrpläne werden 100 Simulationsläufe mit den bereits beschriebenen Parametern ausgeführt. Mit den so generierten Daten können die Fahrpläne hinsichtlich des Verhaltens einzelner Linien und Haltepunkte untersucht werden.

Richtung	Linie														
	1	1A	3	4	5	6	7	8	9	12	13	15	16	18	19
FW	1	6	3	7	5	1	0	6	5	6	7	4	6	9	9
BW	7	4	3	0	2	7	9	7	4	7	6	6	0	7	0

Tabelle 6.6.: Fahrplan 24 - Zufälliger Initialfahrplan

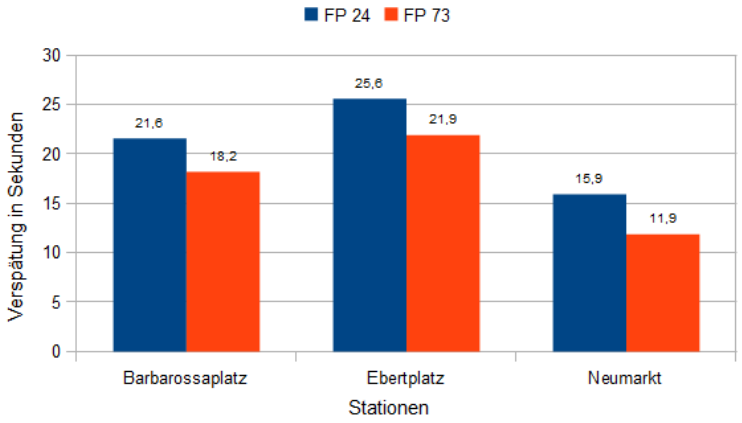


Abbildung 6.15.: Vergleich der Verspätungen an ausgewählten Stationen (Fahrpläne 24 und 73)

Weiterhin kann die Entwicklung der Verspätung einzelner Fahrzeuge im Laufe des Simulationstags verfolgt werden.

Verspätung an ausgewählten Haltepunkten Die Stationen Barbarossaplatz, Ebertplatz und Neumarkt sind wichtige Knotenpunkte des Kölner Stadtbahnnetzes; jede Linie (mit Ausnahme der die Vororte bedienende Linie 13) fährt mindestens eine dieser drei Stationen an. Abbildung 6.15 zeigt die Veränderung der Pünktlichkeit an diesen Stationen, Abbildung 6.16 und Tabelle 6.7 stellen die Verbesserungen aufgeschlüsselt nach den einzelnen Haltepunkten dar. Die Abbildungen 6.17, 6.18 und 6.19 zeigen die geplanten Abfahrtszeiten an den betrachteten Haltepunkten unter den Fahrplänen 24 und 73.

Es zeigt sich, dass sowohl auf Ebene der Stationen als auch auf Ebene der Haltepunkte jeder einzelne Verspätungswert unter dem Optimalfahrplan 73 geringer ausfällt als unter dem Initialfahrplan 24; im Durchschnitt sinkt die Verspätung an den Haltepunkten von 21,0 auf 17,4 Sekunden, also um 3,7 Sekunden oder 18%.

Offensichtlich fallen die Verspätungen an einigen Haltepunkten (wie BAB-227, BAB-449, EBP-201, EBP-268 und NEU-53) unabhängig vom verwendeten Fahrplan deutlich höher aus als an anderen. Die Abfahrtszeiten an diesen Haltepunkten sind im Vergleich zu

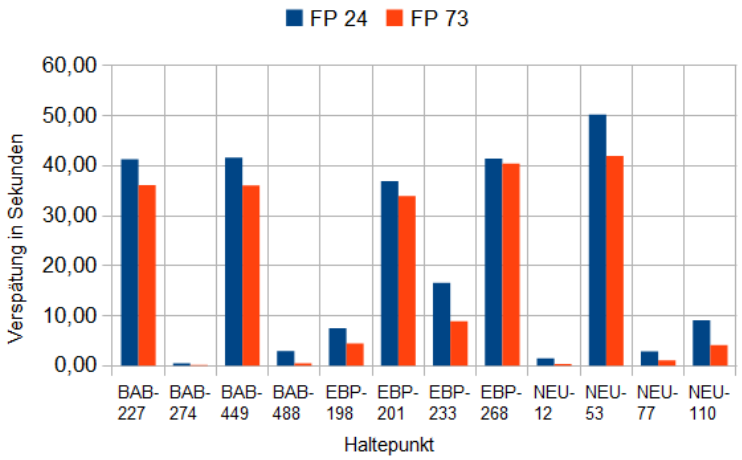


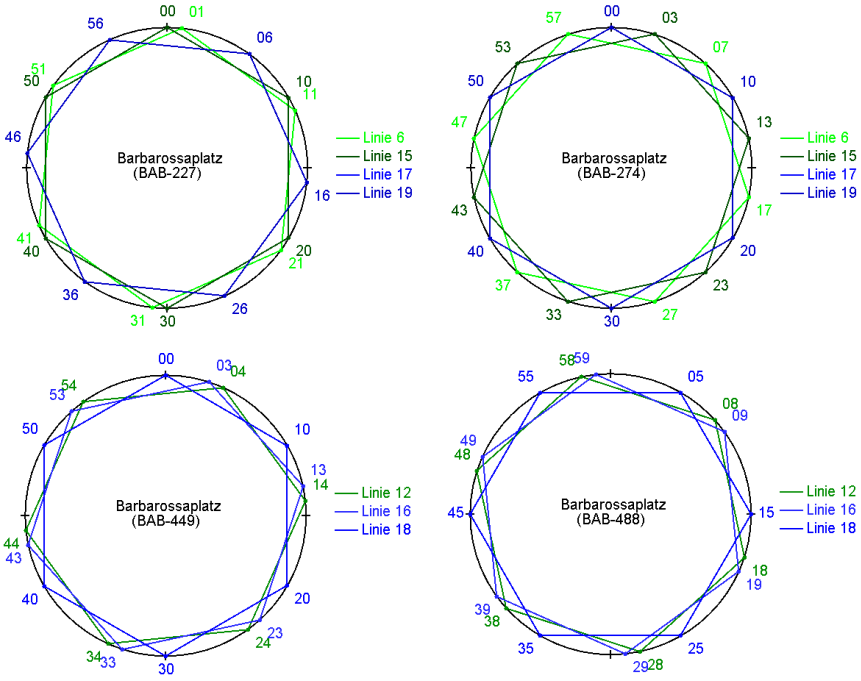
Abbildung 6.16.: Vergleich der Verspätungen an ausgewählten Haltepunkten (Fahrpläne 24 und 73)

Haltepunkten mit geringerer Verspätung innerhalb eines Fahrplans weder deutlich besser noch schlechter verteilt (siehe z.B. das Paar NEU-12 und NEU-53 in Abbildung 6.19). Die Verteilung der Abfahrten allein reicht also als Erklärung nicht, auch die Eigenschaften der den Haltepunkten voran gehenden Strecken müssen betrachtet werden.

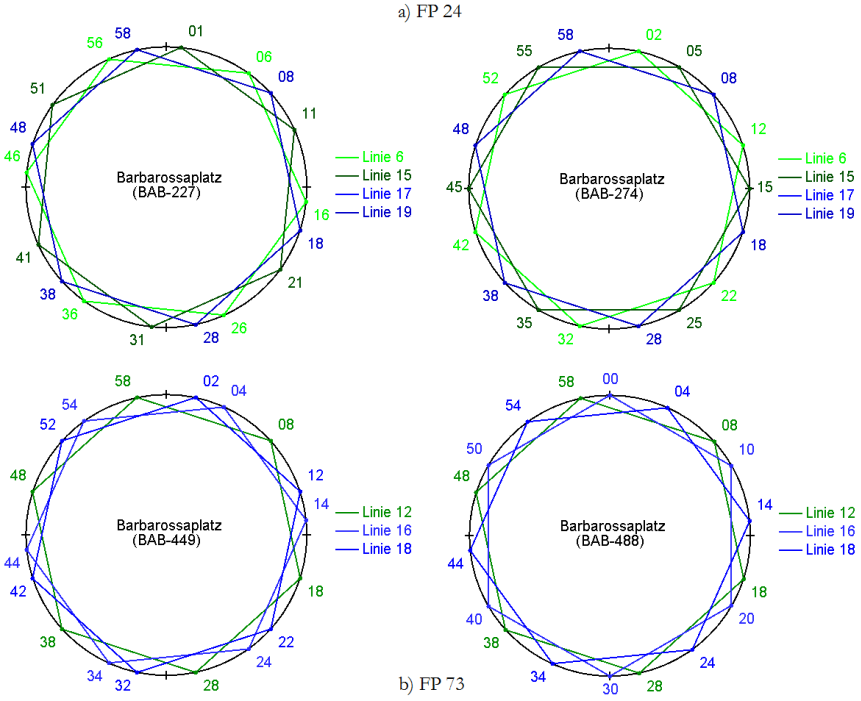
Von den betrachteten Haltepunkten sind NEU-53, BAB-449 und EBP-268 die drei mit der höchsten durchschnittlichen Verspätung.

Der Haltepunkt NEU-53 wird oberirdisch von den Linien 1, 7 und 9 befahren, davor liegt der Haltepunkt HMG-52. Für die dort beginnende, 880 Meter lange Strecke ist eine Fahrzeit von zwei Minuten vorgesehen. Wegen der die Schienen kreuzenden Fahrbahnen und Fußgängerüberwege wird die Strecke von sechs Lichtsignalanlagen in Unterabschnitte aufgeteilt. Wegen der (wie in Abschnitt 6.3.2.4 beschrieben) zufällig erzeugten Ampelphasen von 30 Sekunden Länge beträgt die durchschnittliche Wartezeit pro Lichtsignalanlage 7,5 Sekunden. Bei sechs Ampeln ergeben sich so 45 Sekunden Wartezeit, so dass für das eigentliche Befahren der Strecke (inklusive Anfahren und Abbremsen vor den Ampeln) und den Passagierwechsel am Neumarkt lediglich 75 Sekunden blieben. Das ist offensichtlich zu kurz, so dass die Verspätung auf dieser Strecke von HMG-52 (unter dem optimalen Fahrplan 73: 0,3 Sekunden) zum Neumarkt um im Schnitt 41,6 Sekunden zunimmt.

Der gegenüber von NEU-53 liegende Haltepunkt NEU-12 hat eine deutlich geringere Verspätung von 1,5 und 0,3 Sekunden. Hier fahren die Linien 1, 7 und 9 oberirdisch in Richtung Heumarkt. Einfahrende Bahnen kommen von den Stationen Rudolfplatz (Haltepunkt RDP-54, Linien 1 und 7) oder Mauritiuskirche (Haltepunkt MAK-354, Linie



a) FP 24



b) FP 73

Abbildung 6.17.: Fahrpläne 24 und 73: Abfahrten an der Station Barbarossaplatz

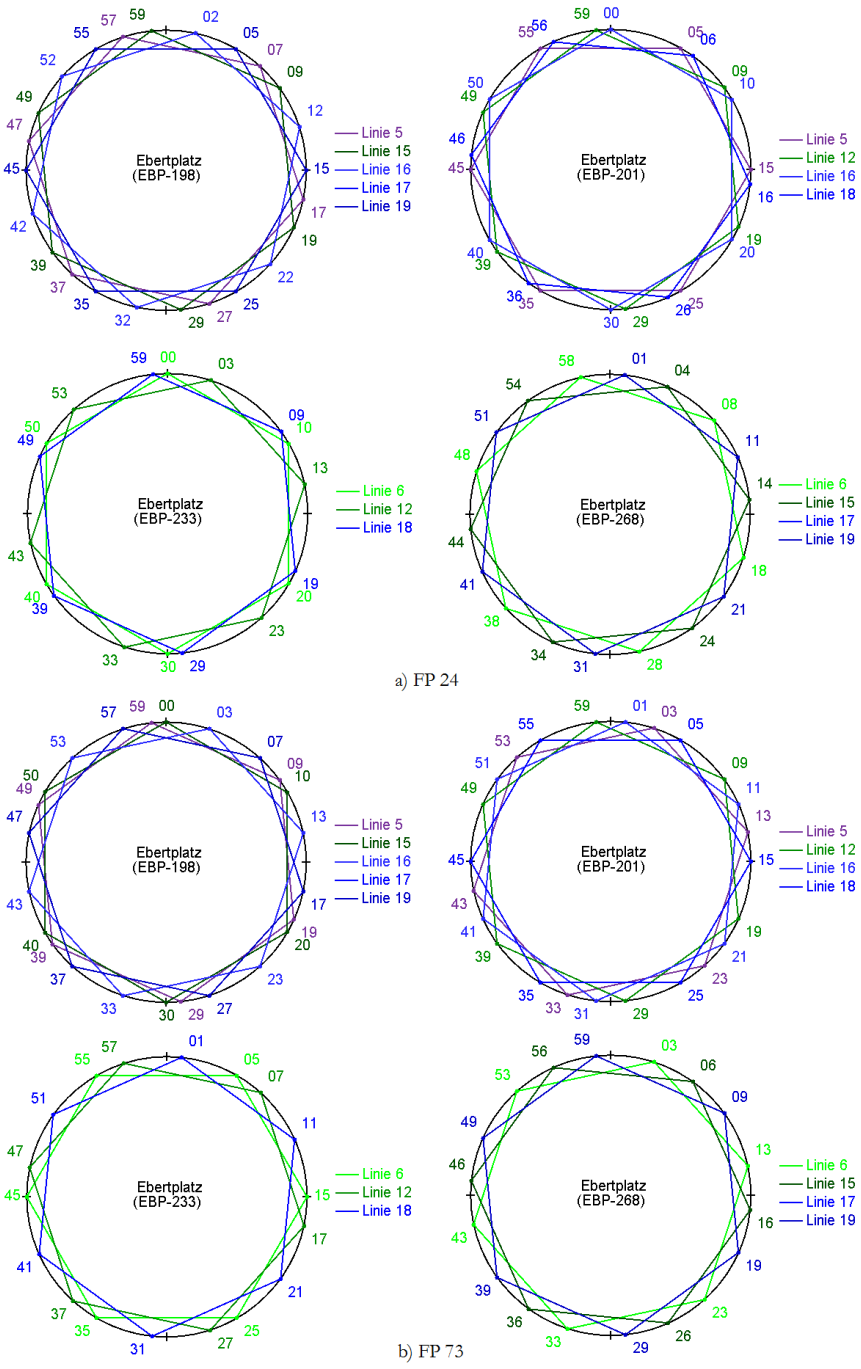


Abbildung 6.18.: Fahrpläne 24 und 73: Abfahrten an der Station Ebertplatz

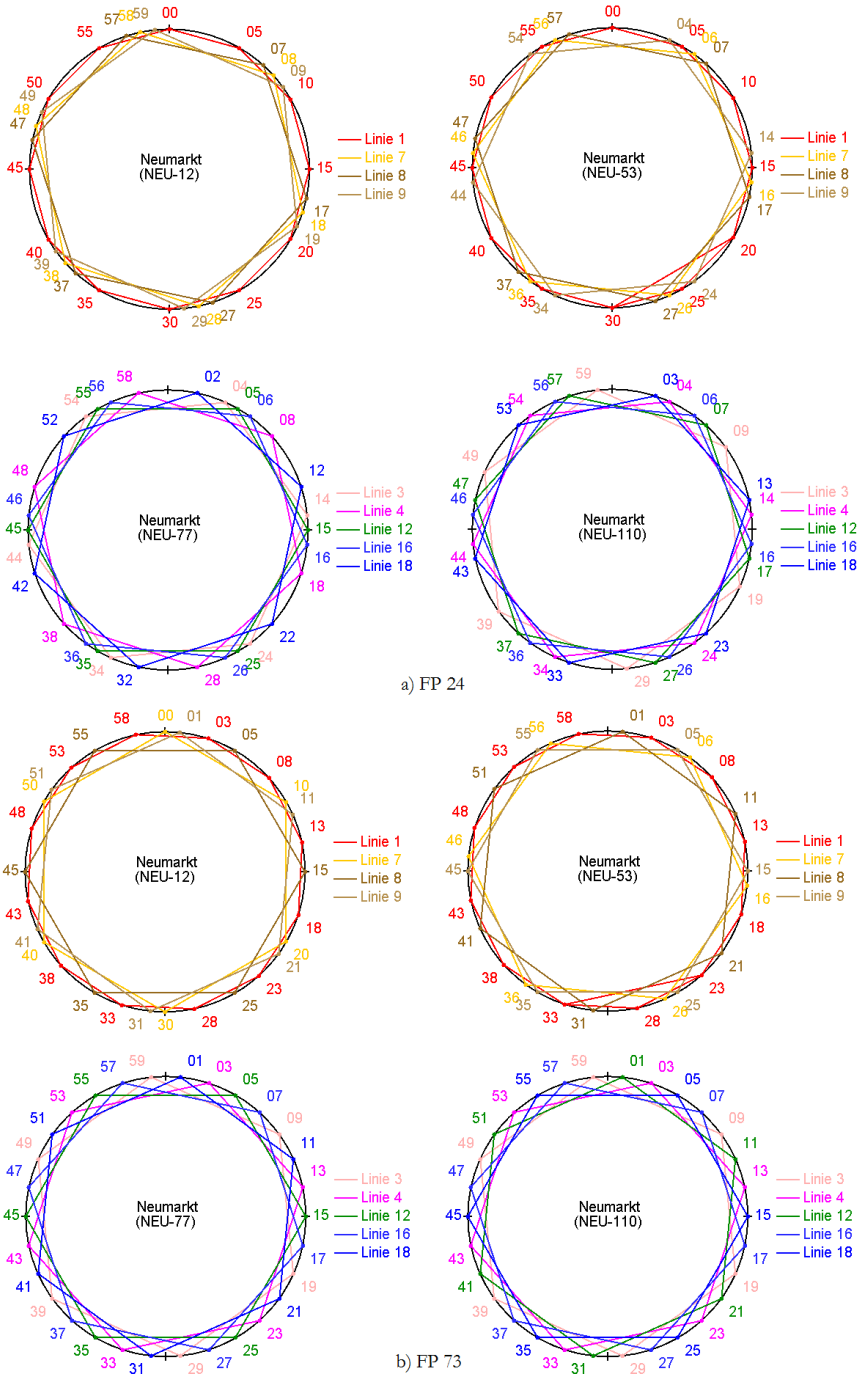


Abbildung 6.19.: Fahrpläne 24 und 73: Abfahrten an der Station Neumarkt

Station	Haltepunkt	ØVerspätung		Abs. Verb.	Rel. Verb.
		FP 24	FP 73		
Barbarossaplatz	BAB-227	41,3	36,1	5,2	0,13
	BAB-274	0,5	0,1	0,3	0,72
	BAB-449	41,6	36,1	5,6	0,13
	BAB-488	3,0	0,5	2,5	0,83
Ebertplatz	EBP-198	7,5	4,4	3,0	0,41
	EBP-201	36,9	33,9	3,0	0,08
	EBP-233	16,6	8,9	7,7	0,46
	EBP-268	41,4	40,4	1,0	0,02
Neumarkt	NEU-12	1,5	0,3	1,1	0,77
	NEU-53	50,3	41,9	8,3	0,17
	NEU-77	2,8	1,1	1,7	0,61
	NEU-110	9,1	4,2	5,0	0,54
Durchschnitt		21,0	17,4	3,7	0,18

Tabelle 6.7.: Verspätung an ausgewählten Haltepunkten in Sekunden (Fahrpläne 24 und 73)

9) über eine zusammen führende Weiche. Die Strecke vom Rudolfplatz ist dabei 590 Meter lang und wird von zwei Lichtsignalanlagen unterteilt, die geplante Fahrzeit ist hier drei Minuten. Bei durchschnittlicher Verzögerung von 15 Sekunden an den Ampeln bleibt Zeit, von RDP-54 (Durchschnittsverspätung unter FP 73: 68,7 Sekunden) im Schnitt 68,4 Sekunden aufzuholen. Auch auf der von MAK-354 kommenden, 420 Meter langen Strecke gelingt es im Vergleich zur geplanten Fahrzeit von drei Minuten einen großen Teil der vorhandenen Verspätung aufzuholen.

Mit 36,1 Sekunden ist der zur Station Barbarossaplatz gehörende Haltepunkt BAB-449 der am zweithöchsten verspätete der betrachteten Auswahl. Fahrzeuge der Linie 16 und 18 fahren von hier aus in Richtung Poststraße. Die Fahrzeuge kommen von den Stationen Eifelwall (Haltepunkt EFW-756, Linie 18) oder Eifelstraße (Haltepunkt EFS-226, Linie 16) über zwei Weichen. Von EFW-756 führt eine 720 Meter lange Strecke mit drei Lichtsignalanlagen nach BAB-449, die in zwei Minuten überfahren werden soll. An den Lichtsignalanlagen müssen die Fahrzeuge im Schnitt 22,5 Sekunden warten, für die zurück zu legenden Strecke bleiben also inklusive Passagierwechsel 97,5 Sekunden. Dazu müssen sich die Bahnen an der Weiche zur Ringstrecke zwischen den hinzu kommenden Fahrzeugen der Linien 6, 12, 15 und 16 einordnen. Die durchschnittliche Verspätung steigt zwischen EFW-756 und BAB-449 im Schnitt von 12,5 um 23,6 Sekunden. Am Haltepunkt EFS-226, der mit BAB-449 durch eine 330 Meter lange, von zwei Lichtsignalanlagen unterteilten Strecke mit einer geplanten Überfahrtzeit von zwei Minuten verbunden ist,

beträgt die durchschnittliche Verspätung 2,4 Sekunden. Hier halten allerdings auch Bahnen der Linien 6, 12 und 15, so dass kein direkter Vergleich zwischen den Verspätungen hier und an BAB-449 möglich ist.

Gegenüber von BAB-449 liegt Haltepunkt BAB-488, der durch von der Station Poststraße kommende Bahnen befahren wird. Vorgesehen sind hier zwei Minuten Fahrzeit für die 650 Meter lange, von einer Weiche geteilten Strecke. Lichtsignalanlagen gibt es an der größtenteils unterirdisch verlaufenden Strecke nicht. Die durchschnittliche Verspätung beträgt am Haltepunkt PSS-78 lediglich 1,2 Sekunden und nimmt unter dem optimalen Fahrplan bis BAB-488 um durchschnittlich 0,7 auf 0,5 Sekunden ab.

Zum Schluss soll noch der Haltepunkt EBP-268 betrachtet werden. Hier fahren Bahnen der Linien 6, 15 und 19, die von Lohsestraße (LOH-267) oder Reichenspergerplatz (RPP-581) kommen, weiter in Richtung Hansaring. Von RPP-581 führt die 465 Meter lange Strecke über zwei Weichen, sie soll inklusive Passagierwechsel am Ebertplatz in 60 Sekunden befahren werden. Die Durchschnittsverspätung in RPP-581 beträgt 4,3 Sekunden, steigt also unter dem Optimalfahrplan zu EBP-268 um 36,1 Sekunden. Von LOH-267 führt eine 800 Meter lange, von einer Weiche unterteilte Strecke nach EBP-268, für die zwei Minuten Fahrzeit geplant sind. Die Verspätung steigt um 5,8 Sekunden von 34,6 Sekunden an LOH-267 auf 40,4 Sekunden an RPP-581.

Verspätung der Linien Einen Überblick über die durchschnittliche Verspätung der Stadtbahnlinien unter den Fahrplänen 24 und 73 geben Abbildung 6.20 und Tabelle 6.8. Zur Berechnung der Werte werden zuerst für jede Fahrt die Fahrtverspätung als Durchschnittsverspätung über die Abfahrten gebildet. Um die Linienverspätung zu erhalten wird dann der Durchschnitt der Fahrtverspätungen über alle der Linie zugeordneten Fahrten berechnet.

Eine große Mehrheit der Linien erreicht unter dem Optimalfahrplan eine erhöhte Pünktlichkeit im Vergleich zum Initialfahrplan. Lediglich die Linien 3, 4 und 5 verbessern sich nicht oder werden sogar minimal schlechter. Am meisten profitieren die Linien 8, 13 und 18, die um 23, 32 und 25% pünktlicher werden. Unter Fahrplan 24 koordiniert sich die nordwärts fahrende Variante der Linie 13 in der Simulation besonders schlecht mit Fahrzeugen der Linie 7. Deren Fahrzeuge, die die gemeinsamen Gleisabschnitte planmäßig nach den Fahrzeugen der Linie 13 befahren sollen, ordnen sich an einer zusammenführenden Weiche vor Linie 13 ein und bremsen diese so an den folgenden Haltepunkten aus. Unter Fahrplan 73 treten diese Probleme nicht auf, Linie 13 ist deutlich pünktlicher. Durchschnittlich sinkt die Linienverspätung von 16,5 auf 13,8 Sekunden, also um 2,6 Sekunden oder 16%.

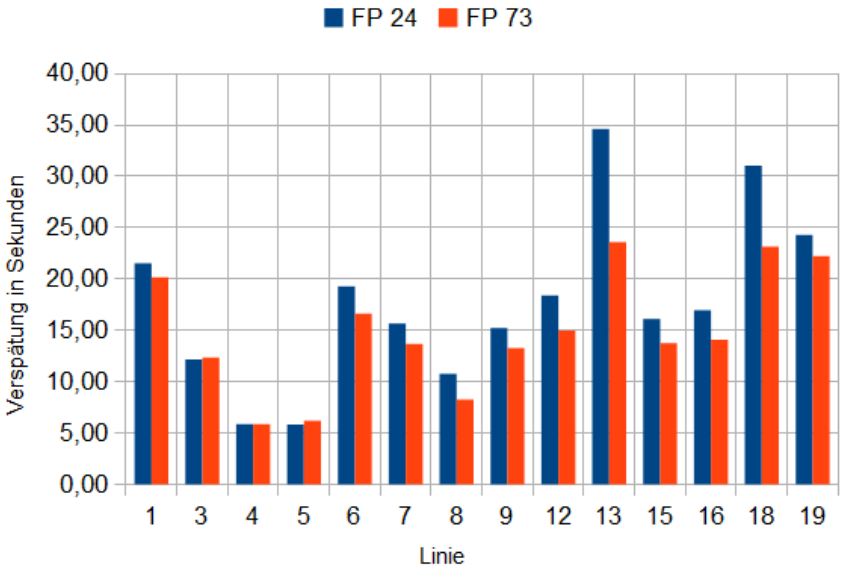


Abbildung 6.20.: Verspätung der Linien (Fahrpläne 24 und 73)

Die Linien 12, 16 und 18 befahren alle drei genannten Stationen mit im Stadtzentrum identischen Linienverläufen. Von diesen Linien hat Linie 12 den kürzesten Verlauf (Linie 16 und 18 bedienen die südlich des Stadtgebiets liegenden Ortschaften) und eignet sich daher am besten zur näheren Betrachtung. Sie liegt mit einem Pünktlichkeitsgewinn von 3,4 Sekunden oder 19% im mittleren Bereich.

Verspätungsentwicklung der Linie 12 Die Linie 12 durchquert die Stadt in Nord-Süd-Richtung, im Süden beginnend in Zollstock (siehe nochmals Abbildung 6.9).Auf dem Weg nach Norden trifft die Linie vor der Station Eifelstraße auf die von den Linien 6, 15 und 16 befahrene Ringstrecke, die sie am Barbarossaplatz stadteinwärts in den Neumarkttunnel einbiegend wieder verlässt. Diesen Streckenabschnitt bedient sie gemeinsam mit den Linien 3, 4, 16 und 18. Bei der Station Appellhofplatz biegen die Linien 3 und 4 nach Westen ab, die Linie 5 kommt hinzu und begleitet die Linie 12 zusammen mit den Linien 16 und 18 bis zum Ebertplatz. Von dort aus bedient die Linie 12 - zuerst gemeinsam mit den Linien 6 und 12 - die Stationen in Nippes, Weidenpesch und Merkenich, wo sie endet. Die Linie hat nur je eine Variante für beide Richtungen, sie bedient 28 Stationen.

In der Simulation verbessert sich die Durchschnittsverspätung der Linie 12 um ca. 19% von 18,4 auf 15,0 Sekunden. Unter den betrachteten Fahrplänen 24 und 73 werden die Fahrten der Linie 12 von zehn Umläufen bedient, deren Durchschnittsverspätung von Fahrplan 24 zu Fahrplan 73 gleichmäßig sinkt (siehe Abbildung 6.21).

Linie	ØVerspätung		Abs. Verb.	Rel. Verb.
	FP 24	FP 73		
1	21,5	20,2	1,4	0,06
3	12,2	12,4	-0,2	-0,01
4	5,9	5,9	0,0	0,00
5	5,8	6,2	-0,4	-0,06
6	19,3	16,6	2,7	0,14
7	15,7	13,7	2,0	0,13
8	10,8	8,3	2,5	0,23
9	15,2	13,3	1,9	0,13
12	18,4	15,0	3,4	0,19
13	34,6	23,6	11,0	0,32
15	16,1	13,7	2,4	0,15
16	17,0	14,1	2,9	0,17
18	31,1	23,2	7,9	0,25
19	24,3	22,2	2,1	0,09
Durchschnitt	16,5	13,8	2,6	0,16

Tabelle 6.8.: Verspätung der Linien in Sekunden (Fahrpläne 24 und 73)

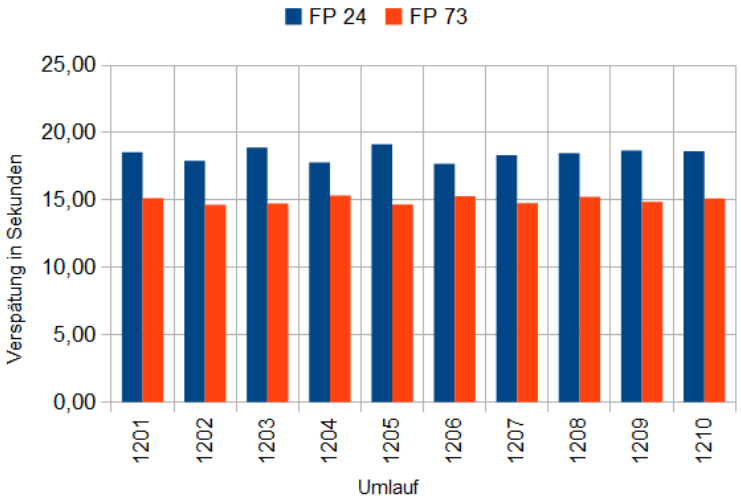


Abbildung 6.21.: Fahrpläne 24 und 73: Durchschnittsverspätung der Umläufe der Linie 12

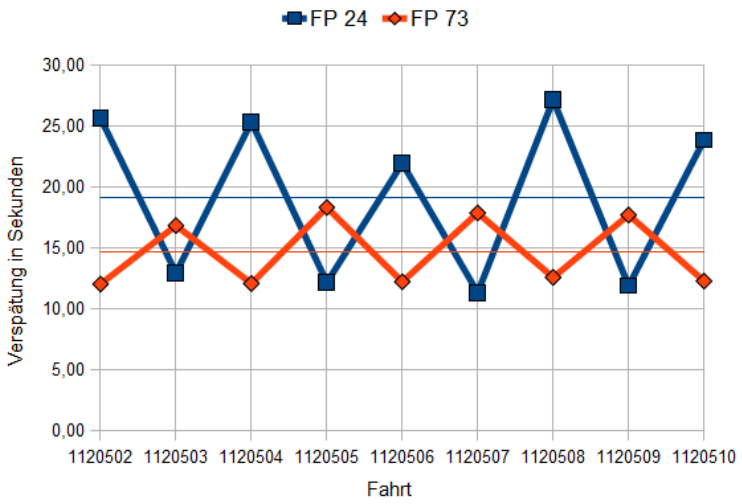


Abbildung 6.22.: Fahrpläne 24 und 73: Durchschnittsverspätung der Fahrten des Fahrzeugs 1205

Ein Blick auf die Durchschnittsverspätungen der einzelnen Fahrten, die ein Fahrzeug im Laufe des Betriebstags ausführt (beispielhaft für Fahrzeug 1205 siehe Abbildung 6.22), zeigt, dass die Fahrten in Richtung Zollstock (unter Fahrplan 24 sind das Fahrten mit geraden Bezeichnern, unter Fahrplan 73 die mit ungeraden) zum einen unter beiden Fahrplänen eine deutlich höhere Durchschnittsverspätung aufweisen als die Fahrten in Richtung Merkenich, zum anderen durch die Optimierung stark an Pünktlichkeit gewinnen. Die Phasenverschiebung des beobachteten Musters erklärt sich dadurch, dass die Optimierungs-Software die abzuarbeitenden Fahrten unter verschiedenen Fahrplänen unterschiedlichen Umläufen zuweisen kann.

Abbildung 6.24 zeigt die unter beiden Fahrplänen sehr ähnlichen Verspätungsverläufe der ersten Fahrten des Fahrzeugs 1205 in Richtung Merkenich (unter Fahrplan 24 ist das Fahrt 1120503, unter Fahrplan 73 Fahrt 1120502). Die meisten Abfahrten sind unter beiden Fahrplänen deutlich unter 30 Sekunden verspätet. Die durchschnittliche Verspätung liegt für diese Fahrt unter Fahrplan 24 bei 13,0 und unter Fahrplan 73 bei 12,0 Sekunden. Die höchsten hier beobachteten Verspätungen treten unter Fahrplan 24 mit 53,6 Sekunden an der Station Niehl (in der Abbildung bezeichnet als NIL) auf, unter Fahrplan 74 mit 51,9 Sekunden an der vorherigen Station Wilhelm-Sollmann-Straße (WSS). Weiterhin auffallend unpünktlich sind Abfahrten an der Station Barbarossaplatz (Haltepunkt BAB-449) mit 36,9 (Fahrplan 24) und 31,9 (Fahrplan 73) Sekunden Verspätung, gefolgt von der Abfahrt an der Station Dom/Hauptbahnhof (DOM-196) mit 28,3 resp.

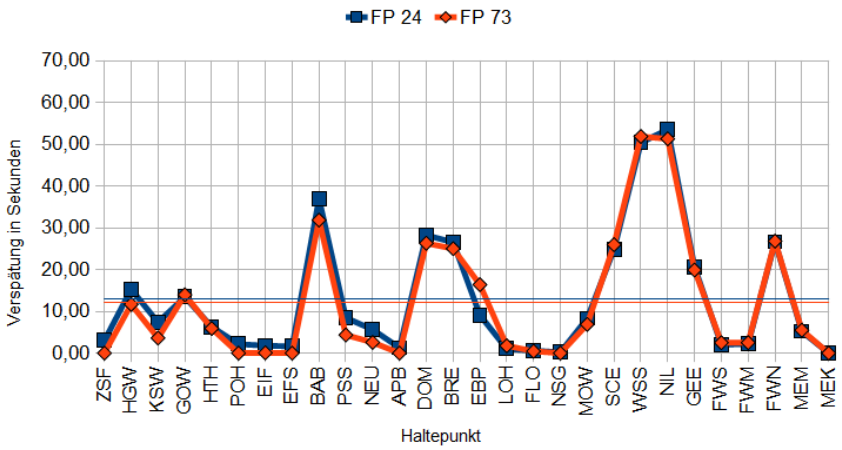


Abbildung 6.23.: Fahrzeug 1205, Fahrt Richtung Merkenich

26,3 Sekunden Verspätung.

Die Erhöhung der Verspätung vor BAB-449 (Zuwachs von 35,3 resp. 31,9 Sekunden) liegt im Durchschnitt, ihr Zustandekommen wurde bereits beschrieben. Ebenso das Sinken der Verspätung zwischen Barbarossaplatz und Poststraße, wo zwei Minuten für eine ampellose Strecke der Länge 560 Meter zur Verfügung stehen. Auf die Linien 6, 15 und 19 muss keine Rücksicht mehr genommen werden, es können etwa 28 Sekunden gut gemacht werden.

Zwischen den Haltepunkten APB-111 und DOM-196 kommt es zu einer zusätzlichen Verspätung von 27,0 resp. 26,3 Sekunden. Hier ist die geplante Fahrzeit für eine Strecke von 520 Metern inklusive einer Weiche mit einer Minute knapp bemessen.

Die höchste Unpünktlichkeit erreicht das Fahrzeug 1205 nach der Passage von Mollwitzstraße (Haltepunkt MOW-237), Scheibenstraße (SCE-238) und Wilhelm-Sollmann-Straße (WSS-239). Hier stehen lediglich zwei Minuten für die erste Strecke von 850 Metern zur Verfügung, die von vier Lichtsignalanlagen unterteilt wird. Die Strecke zwischen Scheibenstraße und Wilhelm-Sollmann-Straße ist 800 Meter lang und wird durch fünf Ampeln überwacht. Hierfür stehen ebenfalls nur zwei Minuten zur Verfügung. Die Verspätung sinkt wieder zwischen Niehl (NIL-246), Geestemünder Straße (GEE-247) und Fordwerke Süd (FWS-248). Hier stehen jeweils zwei Minuten für 810 resp. 430 Meter Strecke ohne Lichtsignalanlagen zur Verfügung, die die Linie 12 allein befährt.

Abbildung 6.23 zeigt die Verspätungsentwicklung des Fahrzeugs 1205 während seiner ersten Fahrt in Richtung Zollstock (unter Fahrplan 24 ist das Fahrt 1120502, unter Fahrplan 73 Fahrt 1120503). Die Verspätung entwickelt sich bis zum achten Halt fast

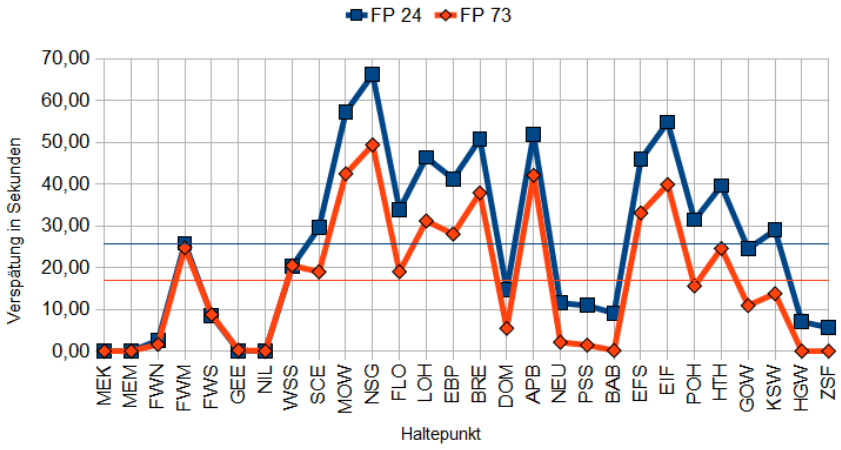


Abbildung 6.24.: Fahrzeug 1205, Fahrt Richtung Zollstock

identisch. Ab Scheibenstraße (Haltepunkt SCE-263) kommt dann unter Fahrplan 24 eine zusätzliche Verspätung von 10,6 Sekunden dazu, die bis kurz vor Ende der Fahrt nicht mehr signifikant abgebaut werden kann.

Vor dem Haltepunkt SCE-263 liegt die erste längere Strecke (770 Meter, drei Lichtsignalanlagen, geplante Fahrzeit von zwei Minuten), die die Linie 12 mit den Linien 12 und 18 teilt. Während die Abfahrten unter dem Optimalfahrplan 73 an SCE-263 gut verteilt sind, fährt unter Fahrplan 24 die Linie 12 unmittelbar den Fahrzeugen der Linie 6 hinterher (siehe Abbildung 6.25). Zwei Minuten davor sind wiederum die Abfahrten der Linie 18 eingeplant. Bereits kleine Verspätungen dieser Linien wirken sich so auf die hinterher fahrende Linie 12 aus, dass deren Verspätung zunimmt.

Der Rest der Fahrt verläuft unter beiden Fahrplänen analog. Bis Neusser Straße/Gürtel (NSG) baut sich eine Verspätung von 66,2 resp. 49,4 Sekunden auf, der Unterschied zwischen den Fahrplänen beträgt hier 9,4 Sekunden.

Für die Strecke zwischen Breslauer Platz (BRE) und Dom/HBf (DOM), auf der sich die Verspätung wieder stark abbaut, ist von der KVB eine Fahrtzeit von zwei Minuten für 500 Meter eingeplant, in Gegenrichtung jedoch nur eine Fahrzeit von einer Minute. Die geplanten Fahrtzeiten beziehen sich immer auf den Zeitraum zwischen den Abfahrten der Fahrzeuge an den Haltepunkten. Die KVB hat an Dom/HBf offensichtlich eine um eine Minute verlängerte Standzeit eingeplant, um dem häufig sehr hohen Fahrgästeaufkommen am Kölner Hauptbahnhof Rechnung zu tragen.

Vor dem Appellhofplatz (APB) trifft die Bahn auf Fahrzeuge der Linien 3, 4, 16 und 18 und verliert beim Einordnen die vorher gewonnene Zeit wieder. Die Steigerung der

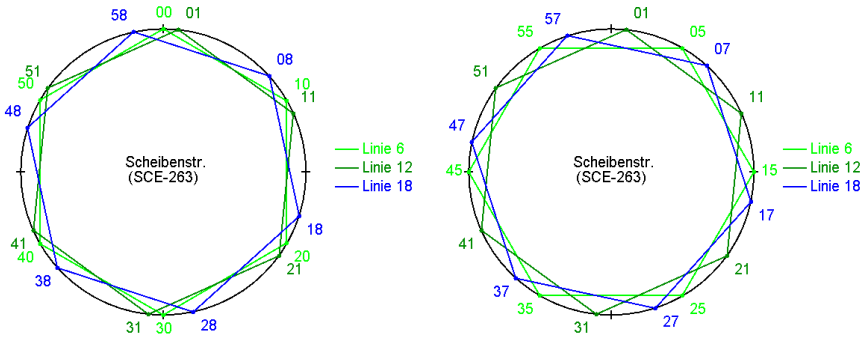


Abbildung 6.25.: Geplante Abfahrten an Wilhelm-Sollmann-Straße (SCE-263) unter Fahrplan 24 (links) und 73 (rechts)

Verspätung um 36,8 resp. 32,9 Sekunden nach BAB-488 entspricht dem Durchschnitt, es steht hier für das Überfahren einer Strecke von 330 Metern, inklusive zweier Weichen und einer Lichtsignalanlage nur eine Minute geplante Zeit zur Verfügung.

Vergleich zum real eingesetzten Fahrplan Der von der KVB im Jahr 2001 eingesetzte Fahrplan liegt in der CATS-Datenbank vor, wurde bisher allerdings nicht betrachtet. Da ihm andere, nicht vorliegende planerische Ansprüche zugrunde liegen, sind aussagekräftige Vergleiche mit den im Projektkontext generierten Fahrplänen nicht möglich.

Trotzdem soll untersucht werden, ob die Simulation des Realfahrplans zumindest Ergebnisse der gleichen Größenordnung liefert, wie die der generierten Fahrpläne. Dazu werden 100 Simulationsläufe des von der KVB verwendeten Fahrplans mit den bereits beschriebenen Parametern durchgeführt und ausgewertet. Die durchschnittliche Verspätung der einzelnen Abfahrten liegt bei 18,7 Sekunden, die schlechtesten generierten Fahrpläne liegen hier bei 18,9 Sekunden, die besten bei 15,4 Sekunden. Die mittlere Verspätung der Abfahrten liegt für den Realfahrplan bei 35,9 Sekunden, bei den schlechtesten generierten Fahrplänen bei 36,6 und den besten bei 30,6 Sekunden.

Bei den Verspätungen der Abfahrten verhält sich der Realfahrplan trotz anderer Anforderungen also ähnlich wie die generierten Pläne. Das bestätigt auch ein Blick auf die über die Linien gemessenen durchschnittlichen Verspätungen (siehe Tabelle 6.9 und Abbildung 6.26). Hier liegt die bei der Anwendung des Realfahrplans gemessene Verspätung bei allen Linien im Bereich der generierten Fahrpläne 24 und 73, mit Ausreißern nach oben bei den Linien 7 und 19.

Ähnliche Ergebnisse zeigt auch eine Übersicht über die an den Haltepunkten der Stationen Barbarossaplatz, Ebertplatz und Neumarkt gemessenen Verspätungswerte (siehe Tabelle 6.10 und Abbildung 6.27). Hier sind ebenfalls keine gravierenden Unterschiede

Linie	ØVerspätung		
	FP 24	FP 73	Realfahrplan
1	21,5	20,2	21,21
3	12,2	12,4	12,7
4	5,9	5,9	5,9
5	5,8	6,2	6,0
6	19,3	16,6	17,7
7	15,7	13,7	22,0
8	10,8	8,3	13,2
9	15,2	13,3	15,3
12	18,4	15,0	14,0
13	34,6	23,6	32,4
15	16,1	13,7	13,7
16	17,0	14,1	19,2
18	31,1	23,2	30,1
19	24,3	22,2	36,1

Tabelle 6.9.: Realfahrplan: Verspätung der Linien in Sekunden

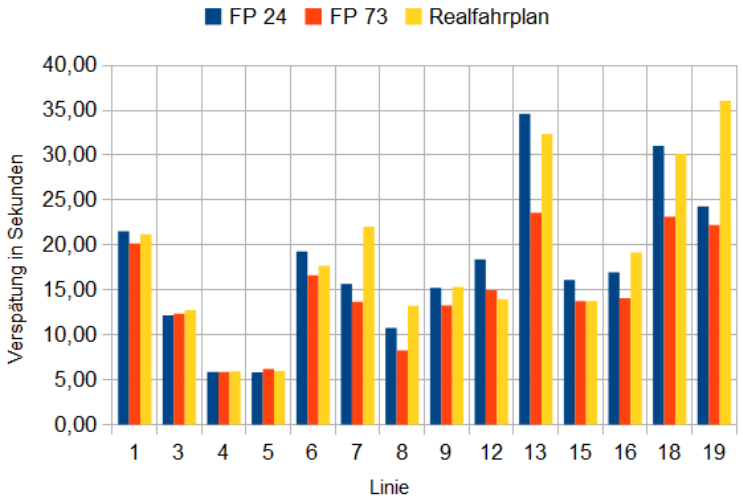


Abbildung 6.26.: Vergleich der Verspätung der Linien (Fahrpläne 24, 73 und Realfahrplan)

Station	Haltepunkt	Verspätung		
		FP 24	FP 73	Realfahrplan
Barbarossaplatz	BAB-227	41,3	36,1	42,0
	BAB-274	0,5	0,1	9,0
	BAB-449	41,6	36,1	42,5
	BAB-488	3,0	0,5	2,1
Ebertplatz	EBP-198	7,5	4,4	3,9
	EBP-201	36,9	33,9	33,0
	EBP-233	16,6	8,9	14,3
	EBP-268	41,4	40,4	42,0
Neumarkt	NEU-12	1,5	0,3	6,7
	NEU-53	50,3	41,9	45,9
	NEU-77	2,8	1,1	5,7
	NEU-110	9,1	4,2	8,7

Tabelle 6.10.: Realfahrplan: Vergleich der Verspätung an ausgewählten Haltepunkten (Fahrpläne 24, 73 und Realfahrplan)

zwischen real verwendetem und generierten Fahrplänen auszumachen.

Zusammenfassung Der Verlauf des Optimierungsprozesses zeigt, dass bereits bei der heuristischen Optimierung sehr gute Lösungen gefunden werden, der deutlich länger rechnende exakte Algorithmus findet nur noch geringe Verbesserungen. Die lange Rechenzeit wird jedoch durch den Vorteil aufgewogen, dass wegen der erschöpfenden Durchsuchung des Lösungsraums nach der Ausführung des Branch-And-Bound-Algorithmus alle besten Lösungen zur Verfügung stehen.

Die Ergebnisse der Simulation bestätigen, dass größere Abstände zwischen den Abfahrten den Fahrplan robuster gegen Verspätungen werden lassen: Die durchschnittliche Verspätung der Abfahrten wird vom initialen zum optimalen Fahrplan um 18,6% oder 3,5 Sekunden gesenkt, die Anzahl der Verspätungen über 60 Sekunden sinkt um 31,9%. Die Häufigkeitsverteilung der Abfahrtsverspätungen zeigt, dass unter den robusten Fahrplänen weniger größere Verspätungen auftreten, dafür aber mehr Verspätungen unter 60 Sekunden. Robuste Fahrpläne verhindern offenbar, dass kleine Verspätungen einzelner Bahnen durch Ressourcenkonflikte zu größeren Störungen führen. Bei den untersuchten Fahrplänen 24 und 73 zeigt sich eine Verbesserung der Pünktlichkeit sowohl bei Betrachtung einzelner Haltepunkte als auch der Linien. Die Entwicklung der simulierten Verspätung ist auch bei genauerer Betrachtung durch die vorliegenden Daten begründbar, die Ergebnisse sind plausibel.

Die Simulation des real verwendeten Fahrplans zeigt Verspätungswerte, die im Rahmen

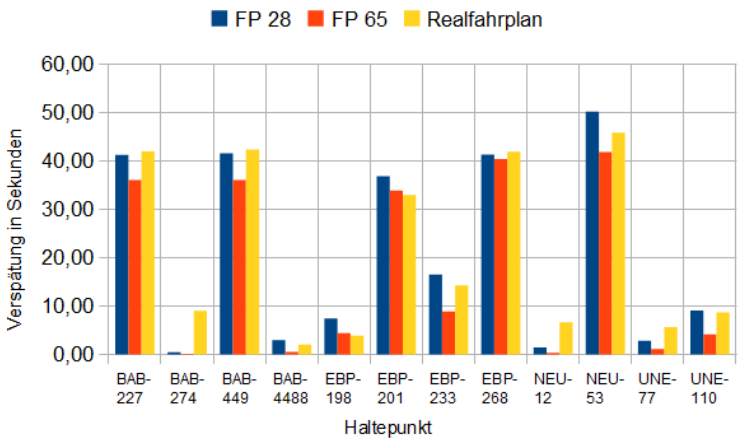


Abbildung 6.27.: Vergleich der Verspätung an ausgewählten Haltepunkten (Fahrpläne 24, 73 und Realfahrplan)

der bei den generierten Fahrplänen beobachteten Verspätungen liegen.

6.4.2. Das Kölner Stadtbahnnetz von 2012

Im Vergleich zum beschriebenen Stadtbahnnetz von 2001 sind im Netz des Jahres 2012 einige Strecken in die Außenbereiche hinein verlängert. Zusätzlich sind einige Linienführungen verändert und einige Linien mit ähnlichen Verläufen zusammen gefasst (für den Netzplan 2012 siehe Kölner Verkehrsbetriebe AG, Deutsche Bahn AG und VRS GmbH in [33]). Einen Überblick über das veränderte Netz gibt Abbildung 6.28.

Im Nordwesten der Stadt wurde die Strecke der Linie 5 von der ehemaligen Endstation Ossendorf um die Stationen Alter Flughafen am Butzweilerhof und Ikea am Butzweiler Hof zur neuen Endstation Sparkasse am Butzweiler Hof (SBH) verlängert. Die Station Takuplatz wurde im Zuge der Veränderungen aufgegeben. Ebenfalls im Nordwesten wurde die Strecke der Linie 3 von der ehemaligen Endstation Bocklemünd um die Stationen Schaffrathsgasse und Ollenhauerring (OHR) erweitert. Die Strecke der Linie 1 wurde von der ehemaligen Endstation Junkersdorf in Richtung Westen über Mohnweg, Bahnstraße, Weiden Zentrum und Weiden Schulstraße nach Köln-Weiden West (WWS) weitergeführt. Dort haben die Fahrgäste Anschluss an das S-Bahn-Netz. Teile der Strecke der bisherigen Linie 6 im Süden der Stadt wurden außer Betrieb genommen, die Stationen Rolandstraße, Bonntor, Koblenzer Straße, Tacitusstraße, Goltsteinstraße/Gürtel, Marienburger Straße und Marienburg, Südpark werden nicht mehr angefahren.

Dazu werden einige Linienverläufe verändert. Die Linien 1, 3 und 5 werden wie oben beschrieben verlängert. Der nördliche Teil der Linie 16 (bislang von Reichenspergerplatz

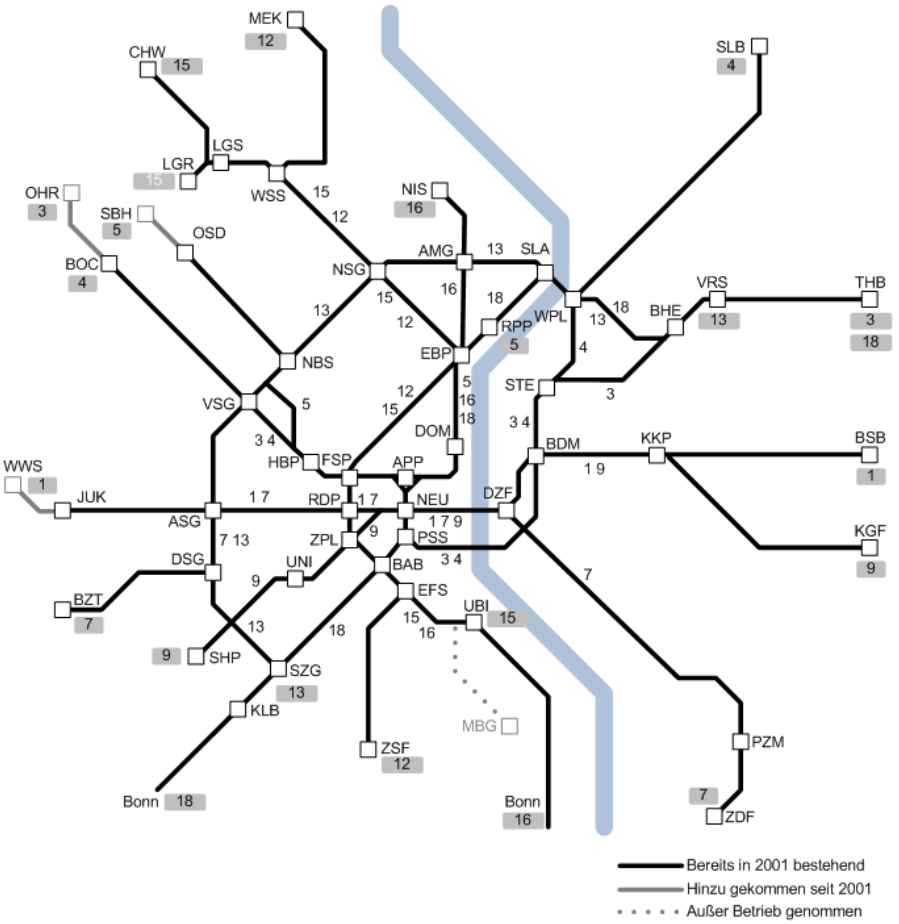


Abbildung 6.28.: Übersicht zum Kölner Stadtbahnnetz im Jahr 2012

über Wiener Platz nach Buchheim Herler Straße) übernimmt den bisherigen Verlauf der Linie 19 (von Reichenspergerplatz über Amsterdamer Straße/Gürtel nach Niehl Sebastianstraße). Die Linie 18 tauscht ihren Verlauf im Kölner Norden (bisher von Ebertplatz über Neusser Straße Gürtel nach Chorweiler) mit dem der Linie 15 (bisher von Ebertplatz über Wiener Platz nach Thielenbruch). Linie 12 bedient nicht mehr den Neumarkt, sondern befährt die Ringstrecke. Die Linien 6, 8 und 19 werden gestrichen und zumindest zum Teil durch Verstärkerfahrten auf den Innenstadtstrecken der Linien 15, 9 und 18 ersetzt. Die Linien 4, 7, 9 und 13 bleiben unverändert.

6.4.2.1. Generieren von Fahrplänen

Da sich im Vergleich zum Netz von 2001 einige Strecken- und Linienverläufe verändert haben, müssen die planerischen Vorgaben angepasst werden. Für das Szenario 2012 werden

Nr.	Variante	Haltepunkt	Startzeitprioritäten									
			0	1	2	3	4	5	6	7	8	9
3	18-FW05	606	0	0	0	0	0	0	0	0	1	1
4	18-BW06	203	0	0	0	0	0	0	0	0	1	1

Table 6.11.: Verwendete Startzeitvorgaben für das KVB-Netz 2012

daher die folgenden Vorgaben verwendet:

1. Wie in Abschnitt 6.4.1.1 beschrieben sollen Fahrten der Linie 15 in Richtung Norden an ihrer Starthaltestelle Ubierring zwei bis drei Minuten vor der aus Bonn kommenden Linie 16 abfahren.

2. Die Linie 1 soll auf der Strecke zwischen Junkersdorf und Brück, Mauspfad (hier verkehren die Varianten 1-FW05 und 1-BW06) im doppelten Takt fahren. Die Verstärkerlinie wird wieder mit 1A bezeichnet. Ebenso sollen die Linien 9, 15 und 18 in ihren Kernbereichen durch die Linien 9A (Varianten 9-FW03 und 9-BW04), 15A (15-FW03 und 15-BW04) und 18A (18-FW03 und 18-BW04) verstärkt werden.

3. Die Linien 3 und 4 sollen wie schon im Fahrplan für 2001 in vier bis sechs Minuten Abstand aufeinander folgen, so dass auf den gemeinsamen Abschnitten in der Innenstadt etwa ein Fünf-Minuten-Takt erreicht wird.

4. Wie bereits in Abschnitt 6.4.1.1 beschrieben, werden beliebig gewählte Fenster für die Abfahrtszeiten der Linie 18 an den Hauptbahnhöfen von Köln und Bonn festgelegt.

5. Für die Endstationen der Linien 1, 3, 4, 5, 7, 9, 12, 13 und 15 werden Mindestzeiten von zwei Minuten für den Richtungswechsel angenommen.

6. Die im Stadtgebiet im Zehn-Minuten-Takt verkehrenden Varianten der Linien 7 und 18 erhalten wieder Verlängerungen in die Außenbereiche mit einem Takt von 20 Minuten.

Die planerischen Bedingungen 1 bis 5 werden in zwei Startzeitvorgaben und 41 Abstandsvorgaben umgesetzt (siehe Tabellen 6.11 und 6.12). Für die Bedingung 6 werden die äußeren Abschnitte der Varianten 7-FW01, 7-BW02, 18-FW05 und 18-BW06 als Verlängerungen der jeweiligen Kernvarianten mit zwanzigminütigem Takt definiert. Die bei der Optimierung dieses Szenarios berücksichtigten Linienvarianten sind in Tabelle 6.13 angegeben.

Nr.	Bed.	Variante 1	Variante 2	Hp. 1	Hp. 2	Abstandsprioritäten									
						0	1	2	3	4	5	6	7	8	9
1	1	15-FW01	16-FW07	627	-	0	0	1	1	0	0	0	0	0	0
2	2	1-FW05	1-FW01	1	-	0	0	0	0	1	1	1	0	0	0
3	2	1-BW06	1-BW02	41	-	0	0	0	0	1	1	1	0	0	0
4	2	9-FW03	9-FW01	398	-	0	0	0	0	1	1	1	0	0	0
5	2	9-BW04	9-BW02	50	-	0	0	0	0	1	1	1	0	0	0
6	2	15-FW03	15-FW01	627	-	0	0	0	0	1	1	1	0	0	0
7	2	15-BW04	15-BW02	258	-	0	0	0	0	1	1	1	0	0	0
8	2	18-FW03	18-FW01	751	-	0	0	0	0	1	1	1	0	0	0
9	2	18-BW04	18-BW02	101	-	0	0	0	0	1	1	1	0	0	0
10	3	4-FW01	3-FW01	65	-	0	0	0	0	1	1	1	0	0	0
11	3	4-BW02	3-BW02	104	-	0	0	0	0	1	1	1	0	0	0
12	5	1-FW01	1-BW02	32	33	0	0	1	1	1	1	1	1	1	1
13	5	1-BW02	1-FW01	889	880	0	0	1	1	1	1	1	1	1	1
14	5	1-FW05	1-BW06	24	41	0	0	1	1	1	1	1	1	1	1
15	5	1-BW06	1-FW05	64	1	0	0	1	1	1	1	1	1	1	1
16	5	3-FW01	3-BW02	93	94	0	0	1	1	1	1	1	1	1	1
17	5	3-BW02	3-FW01	133	130	0	0	1	1	1	1	1	1	1	1
18	5	4-FW01	4-BW02	152	153	0	0	1	1	1	1	1	1	1	1
19	5	4-BW02	4-FW01	122	65	0	0	1	1	1	1	1	1	1	1
20	5	5-FW01	5-BW02	554	581	0	0	1	1	1	1	1	1	1	1
21	5	5-BW02	5-FW01	218	181	0	0	1	1	1	1	1	1	1	1
22	5	7-FW11	7-BW10	317	318	0	0	1	1	1	1	1	1	1	1
23	5	7-BW10	7-FW11	58	7	0	0	1	1	1	1	1	1	1	1
24	5	9-FW01	9-BW02	416	417	0	0	1	1	1	1	1	1	1	1
25	5	9-BW02	9-FW01	440	393	0	0	1	1	1	1	1	1	1	1
26	5	9-FW03	9-BW04	15	50	0	0	1	1	1	1	1	1	1	1
27	5	9-BW04	9-FW03	435	398	0	0	1	1	1	1	1	1	1	1
28	5	12-FW01	12-BW02	468	469	0	0	1	1	1	1	1	1	1	1
29	5	12-BW02	12-FW01	496	441	0	0	1	1	1	1	1	1	1	1
30	5	13-FW01	13-BW02	88	99	0	0	1	1	1	1	1	1	1	1
31	5	13-BW02	13-FW01	542	497	0	0	1	1	1	1	1	1	1	1
32	5	15-FW01	15-BW02	775	776	0	0	1	1	1	1	1	1	1	1
33	5	15-BW02	15-FW01	662	627	0	0	1	1	1	1	1	1	1	1
34	5	15-FW03	15-BW04	244	257	0	0	1	1	1	1	1	1	1	1
35	5	15-BW04	15-FW03	662	627	0	0	1	1	1	1	1	1	1	1
36	5	16-FW03	16-BW04	712	713	0	0	1	1	1	1	1	1	1	1
37	5	16-BW04	16-FW03	669	620	0	0	1	1	1	1	1	1	1	1
38	5	18-FW01	18-BW02	93	94	0	0	1	1	1	1	1	1	1	1
39	5	18-BW02	18-FW01	86	751	0	0	1	1	1	1	1	1	1	1
40	5	18-FW03	18-BW04	86	101	0	0	1	1	1	1	1	1	1	1
41	5	18-BW04	18-FW03	807	744	0	0	1	1	1	1	1	1	1	1

Table 6.12.: Verwendete Abstandsvorgaben für das KVB-Netz 2012

Linie	Hauptvarianten		Verstärkervarianten	
Linie 1	FW01	BW02	FW05	FW06
Linie 3	FW01	BW02		
Linie 4	FW01	BW02		
Linie 5	FW01	BW02		
Linie 7	FW01*	BW02*	FW11	BW10
Linie 9	FW01	BW02	FW03	BW04
Linie 12	FW01	BW02		
Linie 13	FW01	BW02		
Linie 15	FW01	BW02	FW03	FW04
Linie 16	FW03	BW04		
Linie 18	FW01	BW02	FW03, FW05*	BW04, BW06*

* Verlängerung in Außenbereichen, Takt 20 Minuten

Tabelle 6.13.: Für die Optimierung verwendete Linienvarianten im KVB-Netz 2012

Die anderen Parameter der Optimierung werden aus dem vorher gehenden Szenario übernommen. Der Optimierungslauf beginnt wieder mit dem zufälligen Erzeugen von 450 Individuen durch den Genetischen Algorithmus. Der beste Fitnesswert dieser initialen Generation liegt bei 181,483 (Durchschnitt: 192,654, schlechtester Wert: 200,919), in ca. 23 Minuten Rechenzeit verbessert sich dieser Wert im Laufe von 500 Generationen zu 176,090 (Durchschnitt: 176,354, schlechtester Wert: 178,858, siehe auch Abbildung 6.29). Der mit diesem Wert als globale obere Schranke angestoßene Branch-and-Bound-Algorithmus findet nach etwa 23 Stunden Rechenzeit 1.281 optimale Fahrpläne mit dem Zielfunktionswert 175,860.

6.4.2.2. Vergleich der Fahrpläne

Wie bereits in Abschnitt 6.4.1.2 für das KVB-Netz 2001 beschrieben, werden wieder je zehn Fahrpläne aus den Pools von Initial- und Optimallösungen entnommen und jeweils zehn Mal simuliert. Die beschriebenen Simulationsparameter und Erhebungsmethoden werden dazu beibehalten.

Allgemeine Merkmale Der Vergleich der durchschnittlichen Verspätung pro Abfahrt (siehe Abbildung 6.30) zeigt für 2012 ein ähnliches Bild wie schon beim 2001er Netz: Die gemessene Verspätung fällt von den initialen zu den optimalen Fahrplänen von 19,4 um 3,4 Sekunden oder ca. 17,5% auf 16,0 Sekunden, die mittlere Verspätung von 36,8 um 5,3 Sekunden oder 14,4% auf 31,5 Sekunden.

Bei den Häufigkeitsverteilungen der Verspätungen zeigt sich wieder, dass die Zahl der

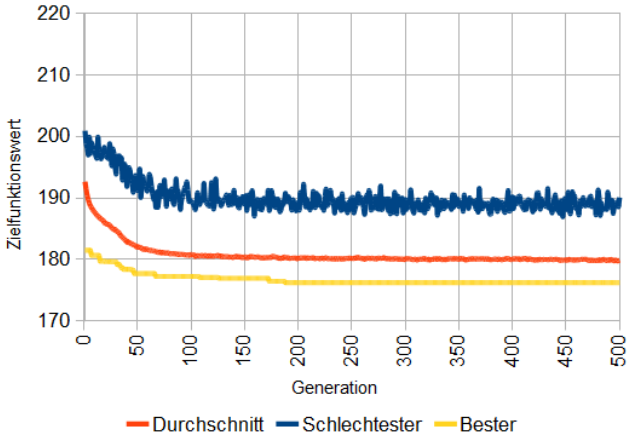


Abbildung 6.29.: Verlauf der Zielfunktion bei der Anwendung des Genetischen Algorithmus auf das KVB-Netz 2012

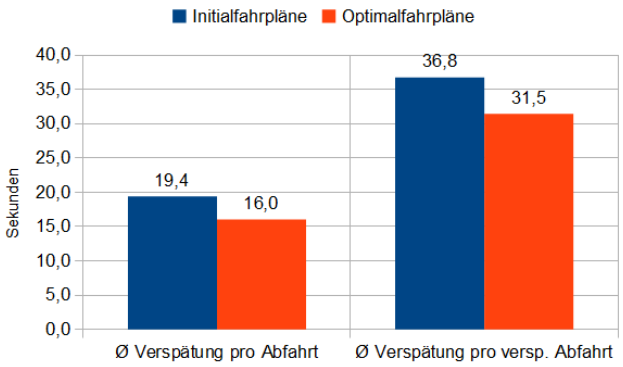


Abbildung 6.30.: Durchschnittliche und mittlere Verspätung pro Abfahrt (2012)

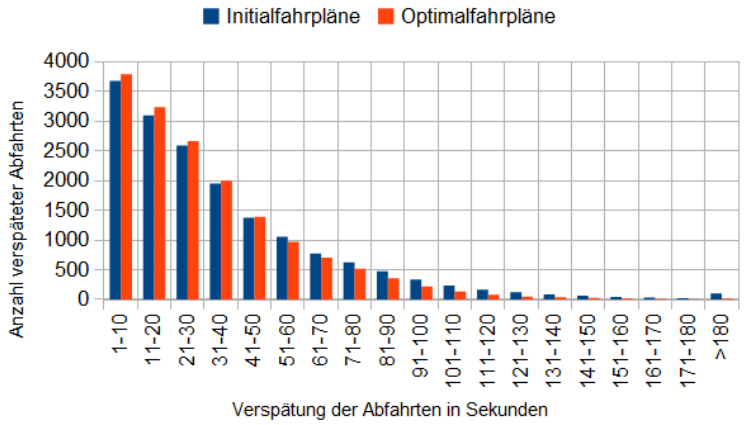


Abbildung 6.31.: Häufigkeitsverteilung der Verspätungen pro Abfahrt (2012)

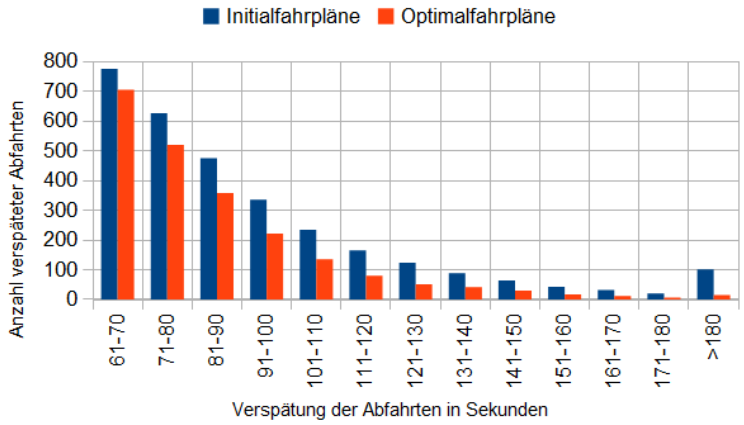


Abbildung 6.32.: Häufigkeitsverteilung größerer Verspätungen pro Abfahrt (2012)

Abfahrten unter 60 Sekunden Verspätung unter den als optimal erkannten Fahrplänen leicht zunimmt (siehe Abbildung 6.31), während die Zahl größerer Verspätungen von durchschnittlich 3.091,2 um 892,7 Abfahrten oder 28,9% auf 2.198,5 zurück geht (siehe Abbildungen 6.32 und 6.33).

Wieder ist also der Effekt zu erkennen, dass durch größere Sicherheitsabstände kleinere Verspätungen seltener zu Staus und damit größeren Verspätungen führen, sich die als optimal bezeichneten Fahrpläne also auch in der Simulation als robuster erweisen.

Vergleich zweier Fahrpläne Auch hier wird wie in Abschnitt 6.4.2.2 beschrieben jeweils ein Fahrplan aus dem Pool der initialen (Fahrplan 133, siehe Tabelle 6.14) respektive der optimalen Fahrpläne (Fahrplan 154, siehe Tabelle 6.15) ausgewählt und als Grundlage

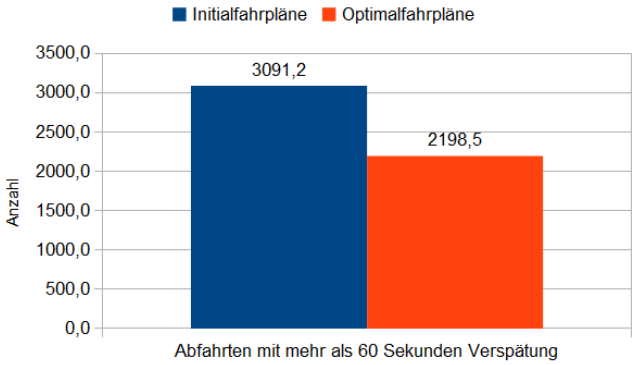


Abbildung 6.33.: Anzahl Abfahrten mit mehr als 60 Sekunden Verspätung (2012)

Richtung	Linie														
	1	1A	3	4	5	7	9	9A	12	13	15	15A	16	18	18A
FW	1	6	3	1	9	2	2	5	7	6	0	5	1	6	8
BW	9	5	9	8	6	2	6	8	7	8	1	7	2	8	9

Tabelle 6.14.: Fahrplan 133 - Initialfahrplan

für je 100 Simulationsläufe genutzt.

Verspätung an ausgewählten Haltepunkten Auch im Netz 2012 gewinnt jede der drei betrachteten Stationen Barbarossaplatz (siehe Abbildung 6.36), Ebertplatz (siehe Abbildung 6.37) und Neumarkt (siehe Abbildung 6.38) durch Anwendung des Optimalfahrplans 133 an Pünktlichkeit hinzu (siehe Abbildung 6.34). An den dazu gehörigen Haltepunkten reduziert sich die beobachtete Verspätung von durchschnittlich 22,4 um 4,3 Sekunden oder 19% auf 18,1 Sekunden (siehe Abbildung 6.35 und Tabelle 6.16). Mit Ausnahme des Haltepunkts BAB-449, an dem die beobachtete Verspätung um 0,4 Sekunden oder 1% ansteigt, verbessert sich die Pünktlichkeit an jedem betrachteten Haltepunkt.

Verspätung der Linien Die durchschnittliche Verspätung der Linien verringert sich von Fahrplan 133 zu Fahrplan 154 in ähnlichem Maße wie in Abschnitt 6.4.1.2 beschrieben

Richtung	Linie														
	1	1A	3	4	5	7	9	9A	12	13	15	15A	16	18	18A
FW	5	7	0	7	7	8	4	9	4	9	3	2	3	6	5
BW	7	9	3	6	1	2	1	4	6	2	2	2	8	4	0

Tabelle 6.15.: Fahrplan 154 - Optimalfahrplan

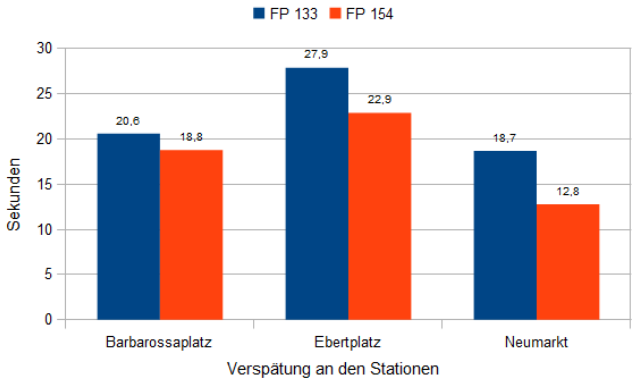


Abbildung 6.34.: Verspätung an ausgewählten Stationen (Fahrpläne 133 und 154)

Station	Haltepunkt	ØVerspätung		Abs. Verb.	Rel. Verb.
		FP 133	FP 154		
Barbarossaplatz	BAB-227	36,0	34,6	1,4	0,04
	BAB-274	4,0	0,1	3,9	0,98
	BAB-449	39,7	40,1	-0,4	-0,01
	BAB-488	2,6	0,3	2,3	0,89
Ebertplatz	EBP-198	20,5	12,8	7,6	0,37
	EBP-201	41,9	41,7	0,2	0,01
	EBP-233	7,1	4,4	2,7	0,38
	EBP-268	42,1	32,5	9,6	0,23
Neumarkt	NEU-12	15,7	0,1	15,6	0,99
	NEU-53	46,1	42,5	3,7	0,08
	NEU-77	2,6	1,2	1,4	0,54
	NEU-110	10,5	7,5	3,0	0,29
Durchschnitt		22,4	18,1	4,3	0,19

Tabelle 6.16.: Verspätung an ausgewählten Haltepunkten in Sekunden (Fahrpläne 133 und 154)

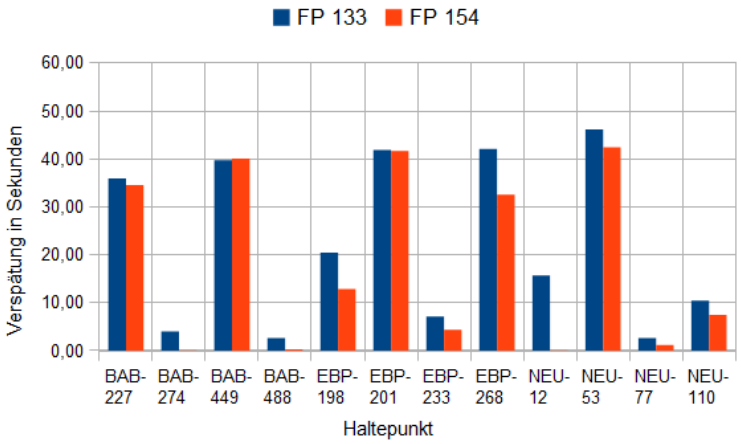


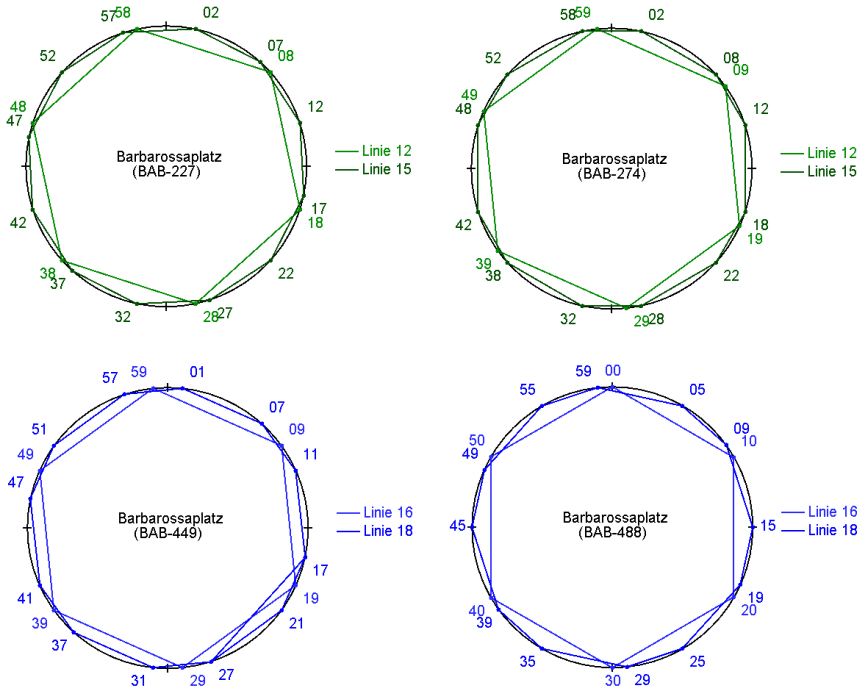
Abbildung 6.35.: Verspätung an ausgewählten Haltepunkten (Fahrpläne 133 und 154)

(siehe Abbildung 6.39 und Tabelle 6.17). Die Werte sinken durchschnittlich von 19,8 um 4,7 Sekunden oder 24% auf 15,1 Sekunden. Wie schon im Netz 2001 gewinnen lediglich die Linie 3, 4 und 5 nicht signifikant an Pünktlichkeit.

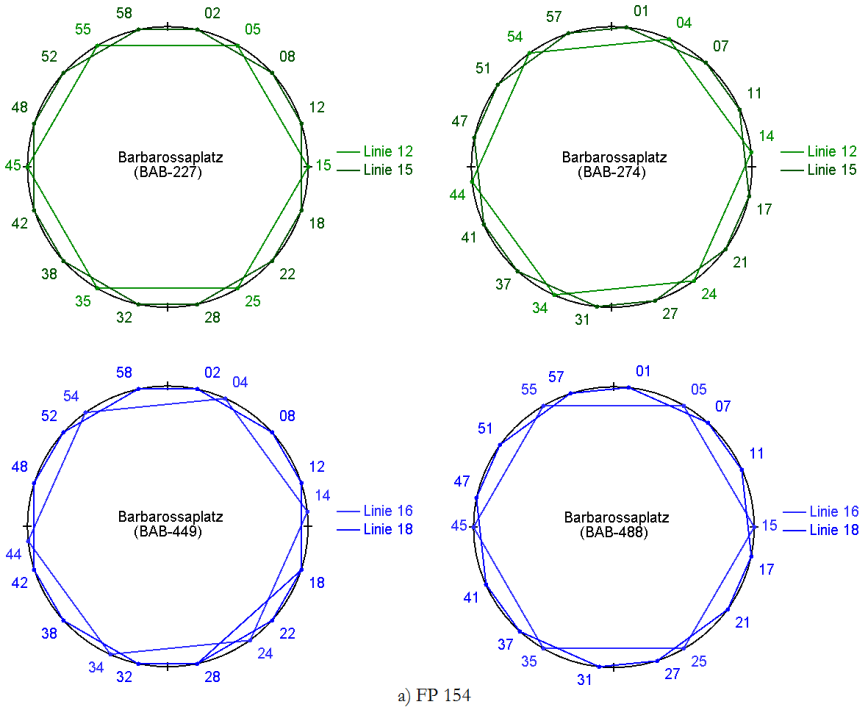
Besonders fällt auf, dass die Linie 7 ihre durchschnittliche Verspätung von 35,1 um 21,6 Sekunden oder 62% auf 13,5 Sekunden senken kann. Die Gewinne sind so hoch, dass die Frage aufkommt, ob die Veränderung allein durch den robusteren Fahrplan zu erklären sind oder ob hier Simulationsartefakte auftreten. Bei der nochmaligen Betrachtung der Verspätung an den Haltepunkten fällt auf, dass die größte Veränderung am Haltepunkt NEU-12 auftritt, an dem unter Fahrplan 133 die Linie 7 unmittelbar vor den Linien 1 und 9, und unter Fahrplan 154 unmittelbar vor Linie 1 abfahren soll (siehe nochmals Abbildung 6.38).

Verspätungsentwicklung der Linie 7 Die Linie 7 durchquert die Kölner Innenstadt in Ost-West-Richtung (siehe nochmals Abbildung 6.28). Fahrten in Richtung Osten beginnen alle 20 Minuten in Frechen Benzelrath und treffen nach 20 Minuten Fahrt an der Station Dürener Straße/Gürtel auf die Gürtelstrecke, die sie sich mit Linie 13 teilen. Bei der Station Aachener Straße/Gürtel biegt die Linie stadteinwärts ab, hier kommen neben der Linie 1 zeitversetzt weitere Fahrten der Linie 7 hinzu, so dass sich für die Durchquerung der Innenstadt ein 10-Minuten-Takt ergibt. Nach der Station Rudolfplatz kommt noch die Linie 9 hinzu. Nach Durchquerung der Innenstadt führt die Linie 7 rechtsrheinisch nach Süden weiter (die Linien 1 und 9 führen weiter nach Osten) und endet schließlich in Zündorf.

Die Linie 7 besteht aus den Varianten 7-BW10 und 7-FW11, die zwischen Zündorf und

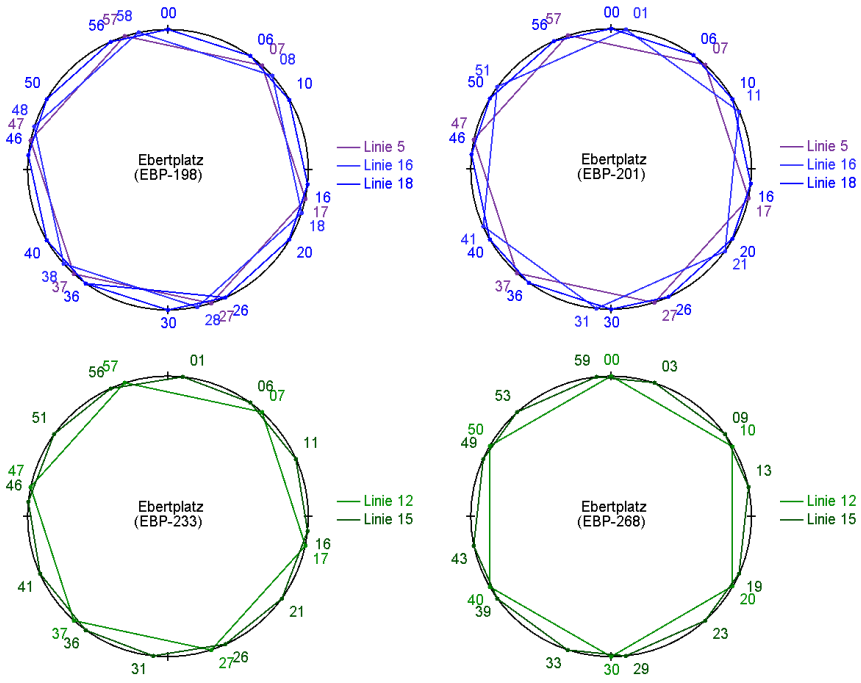


a) FP 133

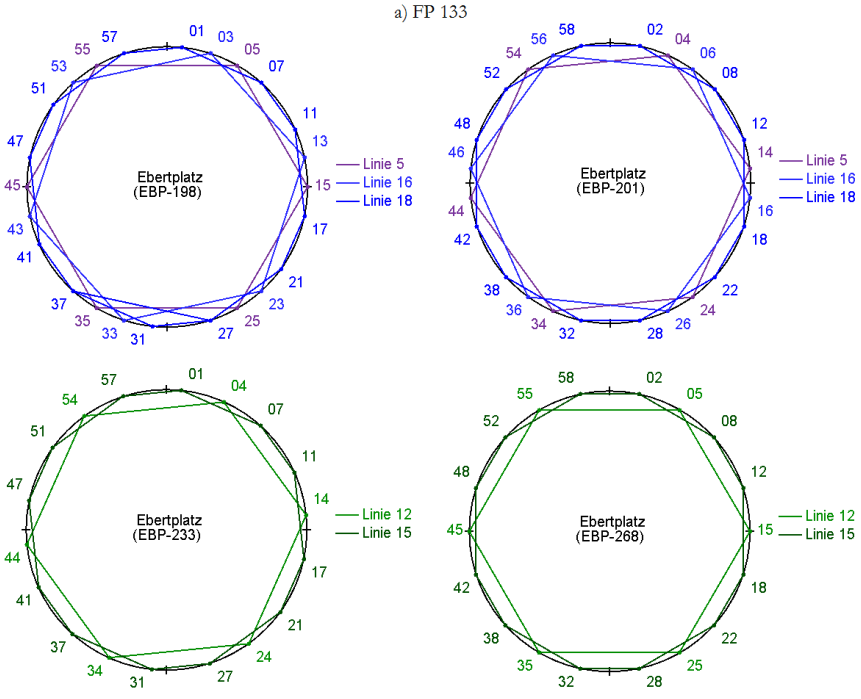


a) FP 154

Abbildung 6.36.: Fahrpläne 133 und 154: Abfahrten an der Station Barbarossaplatz



a) FP 133



a) FP 154

Abbildung 6.37.: Fahrpläne 133 und 154: Abfahrten an der Station Ebertplatz

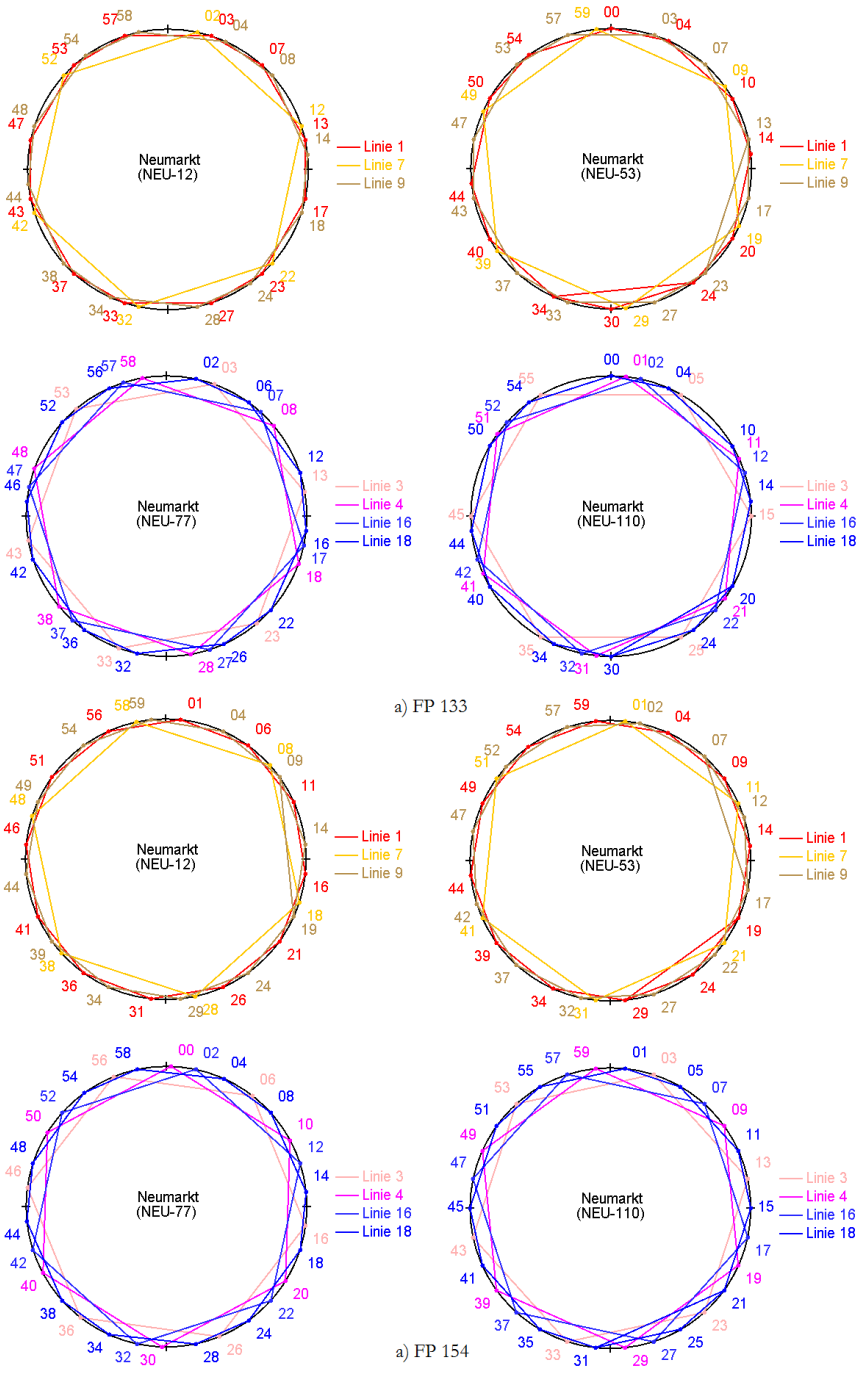


Abbildung 6.38.: Fahrpläne 133 und 154: Abfahrten an der Station Neumarkt

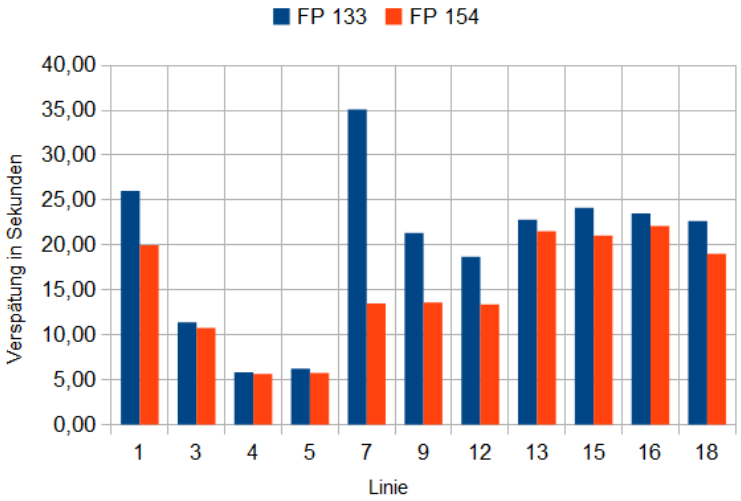


Abbildung 6.39.: Verspätung der Linien in Sekunden (Fahrpläne 133 und 154)

Linie	ØVerspätung		Abs. Verb.	Rel. Verb.
	FP 133	FP 154		
1	26,0	20,0	6,0	0,23
3	11,4	10,8	0,6	0,06
4	5,8	5,6	0,2	0,03
5	6,2	5,8	0,5	0,08
7	35,1	13,5	21,6	0,62
9	21,4	13,6	7,8	0,36
12	18,7	13,4	5,3	0,28
13	22,8	21,5	1,3	0,06
15	24,2	21,0	3,1	0,13
16	23,5	22,1	1,4	0,06
18	22,7	19,1	3,7	0,16
Durchschnitt	19,8	15,1	4,7	0,24

Tabelle 6.17.: Verspätung der Linien in Sekunden (Fahrpläne 154 und 133)

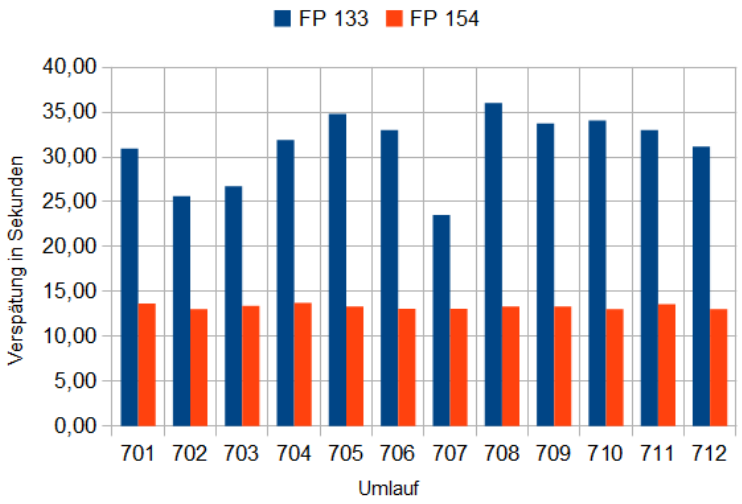


Abbildung 6.40.: Fahrpläne 133 und 154: Durchschnittsverspätungen der Umläufe der Linie 7

Aachener Straße/Gürtel pendeln und den Varianten 7-BW02 und 7-FW01, die alle 20 Minuten Gebiete der Stadt Frechen anbinden.

Die beobachteten Verspätungen der einzelnen Fahrzeuge sind unter Fahrplan 133 recht unterschiedlich, unter Fahrplan 154 sind sie deutlich reduziert, zeigen allerdings kaum Variationen (siehe Abbildung 6.40). Ein Erklärungsansatz für die divergierenden Verspätungswerte wäre sicherlich ein bei den einzelnen Fahrzeugen unterschiedlicher Anteil der langen gegenüber der kurzen Linienvarianten. Es bleibt dann allerdings die Frage, warum die Verspätungswerte sich unter Fahrplan 154 kaum unterscheiden und wie der hohe Gewinn zustande kommt.

Um darüber Aufschluss zu erlangen, wird das Fahrzeug 701 näher betrachtet. Unter dem Optimalfahrplan 154 bleibt die beobachtete Verspätung für jede der Fahrten im gleichen Bereich um 15 Sekunden (siehe Abbildung 6.41). Unter dem initialen Fahrplan 133 finden sich jedoch gravierende Ausreißer: während bei dem betrachteten Fahrzeug 701 die erste gemessene Fahrt 1070102 noch eine durchschnittliche Verspätung von etwa 15 Sekunden aufweist, wächst diese bei der Anschlussfahrt 1070103 auf 91,7 Sekunden an. Bei der darauf folgenden Fahrt ist die Verspätung auf das übliche Niveau gesunken, springt jedoch später am Betriebstag mit 95,6 Sekunden noch einmal auf das höhere Niveau.

Die Verspätungsentwicklung der Fahrt 1070102 verläuft unter beiden Fahrplänen sehr ähnlich und unspektakulär (siehe Abbildung 6.42). Der Anstieg vor der Station Poller

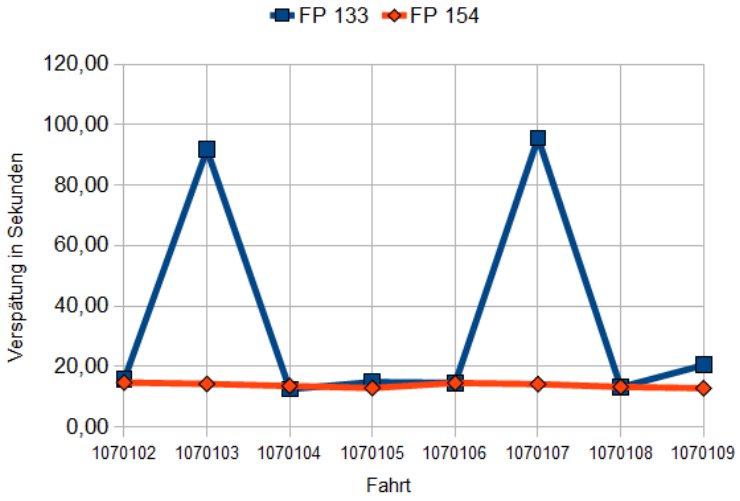


Abbildung 6.41.: Fahrpläne 133 und 154: Durchschnittsverspätungen der Fahrten des Umlaufs 701

Kirchweg (PKI) ist der mit einer Ampel besetzten Strecke von 1.260 Metern geschuldet, für die nur eine Minute Fahrzeit eingeplant sind. In der Innenstadt, wo die Linie 7 sich die vorhandenen Ressourcen mit den Linien 1 und 9 teilt, steigt die Verspätung bis auf knapp 69,9 Sekunden unter Fahrplan 133 und 67,1 Sekunden unter Fahrplan 154 an. Zum Ende der Fahrt ist die Verspätung auf 5,1 Sekunden gesunken, so dass die Anschlussfahrt 1070103 pünktlich beginnen kann.

Unter Fahrplan 154 ist die durchschnittliche Verspätung der Fahrt 1070102 mit 14,7 Sekunden etwas niedriger als unter Fahrplan 133, bei dem 15,6 Sekunden beobachtet werden.

Obwohl ihre Vorgängerfahrt zum Ende hin nur um 5,1 Sekunden verspätet ist, beginnt die Fahrt 1070103 unter Fahrplan 133 an der Station Aachener Straße/Gürtel bereits mit 71,0 Sekunden Verspätung. Die Verspätung steigt bis zur Station Heumarkt auf 207,1 Sekunden an und wird erst ab der Station Deutzer Freiheit wieder nachhaltig abgebaut (siehe Abbildung 6.43). Dieses Verspätungsniveau ist sicherlich nicht allein durch hohe Streckenlängen oder ungünstige Ampelschaltungen zu erklären - die die gleichen Strecken befahrenen Linien 1 und 9 weisen keine so auffällig hohen Verspätungen auf.

Was zeichnet also die Fahrt 1070103 aus? Sie ist eine Fahrt der kurzen Linienvariante 7-FW11, die um 08:42 Uhr am Haltepunkt ASG-7 beginnt, an dem unter Fahrplan 133 eine Minute später eine Fahrt der Linie 1 abfahren soll (siehe Abbildung 6.44).

In den Datengrundlagen der Simulation liegen keine Daten für Rangierfahrten von

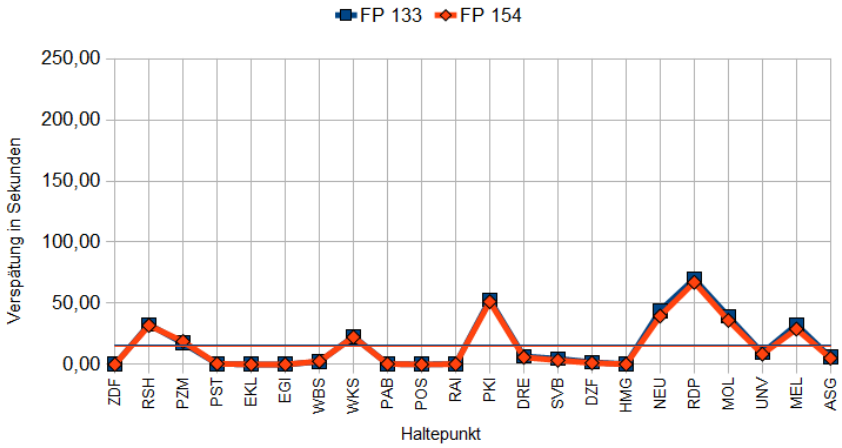


Abbildung 6.42.: Fahrpläne 133 und 154: Fahrzeug 701, Fahrt 1070102

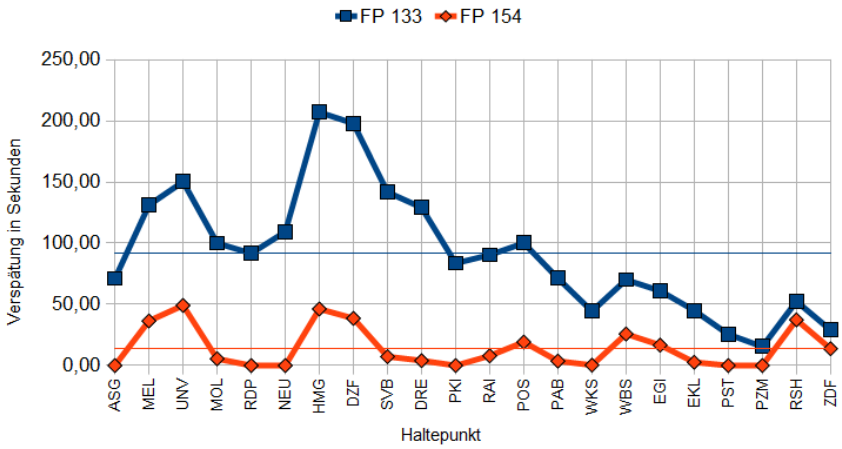


Abbildung 6.43.: Fahrpläne 133 und 154: Fahrzeug 701, Fahrt 1070103

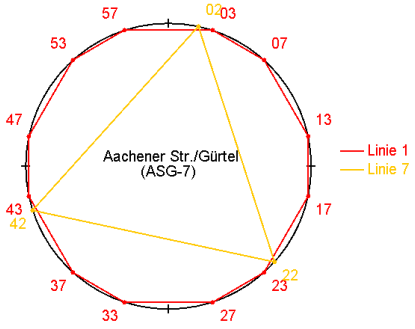


Abbildung 6.44.: Fahrplan 133: Abfahrten am Haltepunkt ASG-7

Endpunkten einer Fahrt zu den Startpunkten der Folgefahrten vor. Die Simulation behilft sich daher damit, die Fahrzeuge beim Erreichen der Endhaltestelle aus dem Modell zu nehmen und kurz vor der geplanten Abfahrt auf dem Starthaltepunkt der Folgefahrt zu platzieren.

Die Kombination dieser Umstände führt nun zu folgendem Verhalten: Das Fahrzeug 701 erreicht pünktlich um 08:38 Uhr den Endhaltepunkt ASG-58 der Fahrt 1070102, wird dort aus der Simulation genommen und soll um 08:42 Uhr im Starthaltepunkt ASG-7 eingesetzt werden. Die von Westen von der Station Maarweg kommende Fahrt der Linie 1 kommt wegen der für die Streckenlänge von 550 Metern (frei von Lichtsignalanlagen) recht langen geplanten Fahrzeit von zwei Minuten frühzeitig vor dem Haltepunkt ASG-7 an und reserviert ihn bis zur geplanten Abfahrt um 08:43 Uhr für sich. Der Haltepunkt ist blockiert - das Fahrzeug 701 ist zwar zur rechten Zeit am rechten Ort verfügbar, kann aber nicht pünktlich in die Simulation eingesetzt werden. Es kann seine Fahrt erst dann beginnen, wenn das Fahrzeug der Linie 1 seine Fahrt planmäßig fortgesetzt hat. Daher fährt die Bahn 701 - die planmäßig vor der Bahn der Linie 1 herfahren sollte - der Linie 1 hinterher und hat so bereits ab ihrem Start über eine Minute Verspätung. Die kann das Fahrzeug so lange nicht einholen, wie es von der vorfahrenden Linie 1 blockiert wird. Zwischen den Stationen Neumarkt und Heumarkt - einer dicht befahrenen Strecke der Länge 880 Meter mit sieben Lichtsignalanlagen und recht knappen zwei Minuten geplanter Fahrzeit - kommen nochmals fast 100 Sekunden Verspätung hinzu. Erst nachdem die mit den Linien 1 und 9 gemeinsam genutzte Strecke vor der Station Deutzer Freiheit (DZF) endet, kann das Fahrzeug die Verspätung von über drei Minuten nach und nach wieder abbauen. Da bis zur Zielstation Zündorf die Verspätung fast vollständig wieder eingefahren ist, kann die Anschlussfahrt 1070104 pünktlich beginnen. Sie verläuft analog zu Fahrt 1070102 und zeigt keine Auffälligkeiten. Die darauf folgende Fahrt 1070105

beginnt schon in Frechen und fährt einen anderen Haltepunkt der Station Aachener Straße/Gürtel an, kommt daher nicht auf die beschriebene Weise mit der Linie 1 in Konflikt. Ihr Verspätungsverlauf bleibt unauffällig.

Die außergewöhnlich hohe Durchschnittsverspätung einiger Fahrten der Linie 7 kommt also durch ein Simulationsartefakt zustande, das seine Ursache im Wesentlichen im Fehlen von Rangierfahrten in der Datenbasis hat. Das Modell muss sich hier mit Hilfskonstrukten begnügen, wie dem weit im Voraus Reservieren von Haltepunkten und dem Herausnehmen und Wiedereinsetzen von Fahrzeugen. Könnte die Rangierfahrt in der Simulation beachtet werden, dann könnte das Fahrzeug bei pünktlichem Eintreffen am Zielhaltepunkt der vorherigen Fahrt wenden und wie geplant vor der Linie 1 in den Starthaltepunkt einfahren. Die Fahrt verliefe unauffällig.

Zusammenfassung Bei der Anwendung auf das Stadtbahnnetz von 2012 konnte der Optimierer den Zielfunktionswert der untersuchten Fahrpläne von maximal 201,919 um 12,9% auf den besten Wert 175,860 senken. Durch die zusätzlichen planerischen Ansprüche zu Verstärkerfahrten im Netz 2012 sind mehr Linienabstände auf den Bereich von vier bis sechs Minuten festgelegt, viele potentiell schlechte Pläne (wie z.B. alle in ein 10-Minuten-Intervall einzuplanenden Abfahrten mit je einer Minute Abstand, dann ein einzelner großer Abstand) sind also nicht zulässig. Bester und schlechtester gefundener Zielfunktionswert sind u.a. deshalb etwas niedriger als bei der Betrachtung des Netzes von 2001 (dort lag das Optimum bei 181,467, der schlechteste Initialfahrplan bei 217,005).

Unter den betrachteten Fahrplänen sinkt der Durchschnitt der Abfahrtsverspätung von 19,4 um 17,5% oder 3,4 Sekunden auf 16,0 Sekunden. Es zeigt sich wieder, dass unter den Optimalfahrplänen weniger um mehr als 60 Sekunden verspätete Abfahrten auftreten als unter den Initialfahrplänen, es allerdings im Gegenzug zu mehr kleinen Verspätungen unter einer Minute kommt. Bei näherer Betrachtung zweier Fahrpläne 133 und 154 werden sowohl an über die Haltepunkte als auch über den Linien unter dem Optimalfahrplan geringere oder zumindest gleiche Verspätungswerte gemessen. Eine signifikante Verschlechterung tritt bei keiner Messung auf.

Die überraschend hohe Varianz in der Verspätung einiger Fahrten der Linie 7 kann auf ein Simulationsartefakt zurück geführt werden, das auf die Unvollständigkeit der Daten zurück geht.

Auch die Betrachtung des Netzes 2012 zeigt, dass sich die verwendeten Modelle plausibel verhalten und bestätigen die Grundannahme, dass größere Abstände zwischen den Abfahrten für eine höhere Robustheit gegenüber kleinen Störungen sorgen.

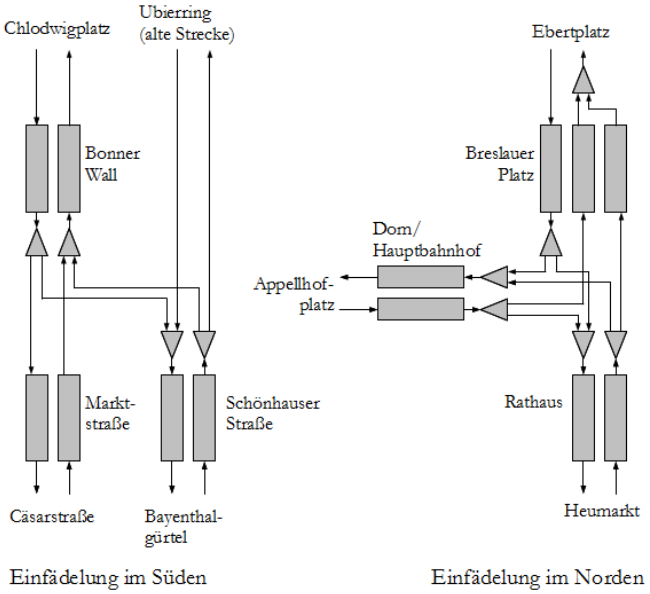


Abbildung 6.45.: Einfädelung des Nord-Süd-Tunnels

6.4.3. Der Einfluss des neuen Nord-Süd-Tunnels

Die Informationen über die im Bau befindliche Nord-Süd-Bahn wurden aus Publikationen und Pressemitteilungen der KVB zusammen getragen, siehe [31] und [32]. Der Ausbau der Nord-Süd-Bahn erfolgt demnach in drei Baustufen: In der ersten Stufe wird der ca. vier Kilometer lange Nord-Süd-Tunnel gebaut, der 11,50 bis 28,50 Meter tief unter dem Stadtzentrum verläuft. Mit dem Bau wurde im Januar 2004 begonnen. Im Verlauf der zweiten Baustufe wird der Tunnel nördlich der Station Schönhauser Straße (SHS) an das Rheinufer und die dort verlaufende Strecke der Linie 16 angebunden. Im Anschluss soll dann mit der dritten Baustufe die Strecke oberirdisch verlängert werden.

Die Fertigstellungsdaten sind spätestens in Folge der Katastrophe am Waidmarkt im März 2009 unübersichtlich geworden. Für die folgenden Untersuchungen wird trotzdem davon ausgegangen, dass alle drei Baustufen im geplanten Umfang[1] realisiert sind.

Neben dem Bau des eigentlichen Tunnels und seiner oberirdischen Verlängerungen werden eine Reihe von Stationen neu- oder ausgebaut: Komplett neu sind im Nord-Süd-Tunnel die Stationen Rathaus (RAT), Kartäuserhof (KAH), Bonner Wall (BOW) und Marktstraße (MAS). Oberirdisch kommen auf dessen etwa zwei Kilometer langen Erweiterung nach Süden die Stationen Cäsarstraße, Bonner Straße/Gürtel, Ahrweiler Straße und Arnoldshöhe bis zum Übergang der Bonner Straße auf die A555 am Bonner Ver-

[1] Stand Februar 2012

teilerkreis hinzu. Um zusätzliche Haltepunkte ausgebaut werden die Stationen Breslauer Platz, Heumarkt, Severinsstraße und Chlodwigplatz. Der Tunnel teilt sich im Norden und wird an die Stationen Dom/Hbf und Breslauer Platz angeschlossen und im Süden zwischen den Stationen Ubierring und Schönhauser Straße eingefädelt (siehe Abbildung 6.45).

Um den Tunnel zu nutzen, werden einige Linienverläufe, wie von der KVB in [31] beschrieben, verändert: Linie 5 verlässt an der Station Dom/Hbf den bisherigen Verlauf und fährt durch den Nord-Süd-Tunnel und über die Bonner Straße bis zur Station Arnoldshöhe im Süden der Stadt. Linie 16 fährt nicht mehr über den Südteil der Ringe, sondern verlässt nördlich der Station Schönhauser Straße den alten Linienverlauf und bedient die Stationen Bonner Wall, Chlodwigplatz, Kartäuserwall, Severinsstraße, Heumarkt und Rathaus, bevor sie ab Breslauer Platz dem bisherigen Verlauf weiter nach Norden folgt. Sie wird zwischen Reichensperger Platz und Marktstraße durch eine Kernlinie 16A ergänzt und erhält so auf dem Weg durch das Stadtzentrum eine Taktverdichtung.

Durch den Neumarkttunnel führen also durch die Umleitung der Linie 16 nicht mehr wie bisher vier, sondern nur noch drei Linien. Die auf Straßenniveau operierende, und damit verstärkt Störungen ausgesetzte Gleiswechselanlage im Bereich um die Station Barbarossaplatz wird ebenfalls entlastet.

Zusätzlich wird bis 2020 die eingleisige Strecke zwischen Bahnhof Brühl Mitte und Brühl Badorf zu einer zweigleisigen Strecke erweitert.

Einen Überblick über die Veränderungen im Bereich der Kölner Innenstadt gibt Abbildung 6.46.

6.4.3.1. Generieren von Fahrplänen

Die in diesem Szenario verwendeten planerischen Vorgaben entsprechen inhaltlich weitgehend den in Abschnitt 6.4.2.1 beschriebenen, müssen aber wegen der veränderten Gleis- und Linienführung angepasst werden. Dazu kommt eine Linie 16A, die mit der Linie 16 eine Innenstadtstrecke mit fünfminütigem Takt bildet (siehe nochmals Abbildung 6.46). Die bestehende Kopplung der Linien 15 und 16 an der Station Chlodwigplatz wird aufgelöst.

Eine Übersicht über die verwendeten Startzeit- und Abstandsvorgaben zeigen die Tabellen 6.18 und 6.19. Die bei der Optimierung berücksichtigten Linienvarianten werden in Tabelle 6.20 aufgeführt.

Die sonstigen Parameter werden wieder aus den vorhergehenden Szenarien übernommen. Der Optimierungslauf beginnt wieder mit dem zufälligen Erzeugen von 450 Individuen. Der beste Fitnesswert dieser initialen Generation liegt bei 182,494 (Durchschnitt:

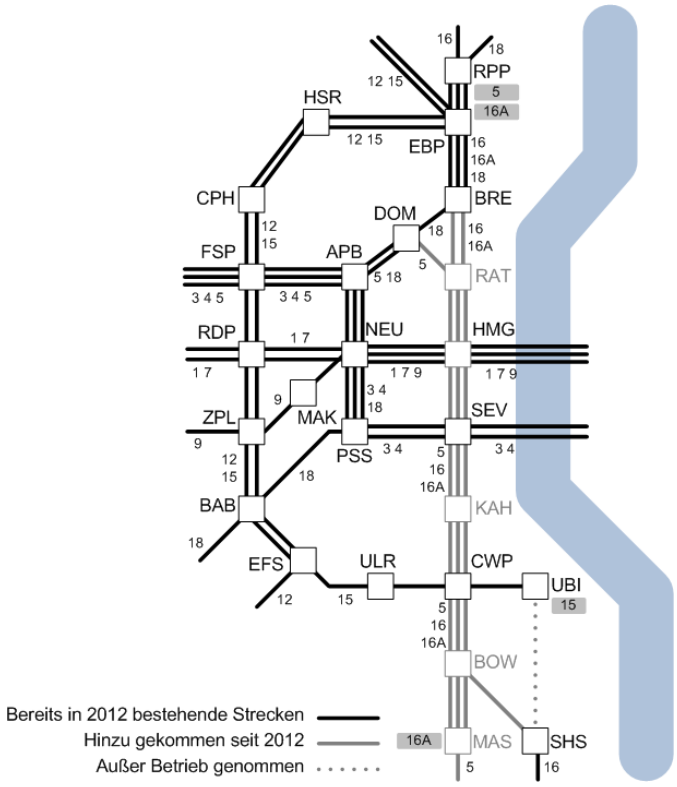


Abbildung 6.46.: Übersicht zum Nord-Süd-Tunnel

Nr.	Variante	Haltepunkt	Startzeitprioritäten									
			0	1	2	3	4	5	6	7	8	9
1	18-FW05	606	0	0	0	0	0	0	0	0	1	1
2	18-BW06	203	0	0	0	0	0	0	0	0	1	1

Table 6.18.: Verwendete Startzeitvorgaben für das KVB-Netz 2020

Nr.	Bed.	Variante 1	Variante 2	Hp. 1	Hp. 2	Abstandsprioritäten									
						0	1	2	3	4	5	6	7	8	9
1	2	1-FW05	1-FW01	1	-	0	0	0	0	1	1	1	0	0	0
2	2	1-BW06	1-BW02	41	-	0	0	0	0	1	1	1	0	0	0
3	2	9-FW03	9-FW01	398	-	0	0	0	0	1	1	1	0	0	0
4	2	9-BW04	9-BW02	50	-	0	0	0	0	1	1	1	0	0	0
5	2	15-FW03	15-FW01	627	-	0	0	0	0	1	1	1	0	0	0
6	2	15-BW04	15-BW02	258	-	0	0	0	0	1	1	1	0	0	0
7	2	16-FW03	16-FW01	1041	-	0	0	0	0	1	1	1	0	0	0
8	2	16-BW04	16-BW02	581	-	0	0	0	0	1	1	1	0	0	0
9	2	18-FW03	18-FW01	751	-	0	0	0	0	1	1	1	0	0	0
10	2	18-BW04	18-BW02	101	-	0	0	0	0	1	1	1	0	0	0
11	3	4-FW01	3-FW01	65	-	0	0	0	0	1	1	1	0	0	0
12	3	4-BW02	3-BW02	104	-	0	0	0	0	1	1	1	0	0	0
13	5	1-FW01	1-BW02	32	33	0	0	1	1	1	1	1	1	1	1
15	5	1-BW02	1-FW01	889	880	0	0	1	1	1	1	1	1	1	1
16	5	1-FW05	1-BW06	24	41	0	0	1	1	1	1	1	1	1	1
17	5	1-BW06	1-FW05	64	1	0	0	1	1	1	1	1	1	1	1
18	5	3-FW01	3-BW02	93	94	0	0	1	1	1	1	1	1	1	1
19	5	3-BW02	3-FW01	133	130	0	0	1	1	1	1	1	1	1	1
20	5	4-FW01	4-BW02	152	153	0	0	1	1	1	1	1	1	1	1
21	5	4-BW02	4-FW01	122	65	0	0	1	1	1	1	1	1	1	1
22	5	5-FW01	5-BW02	1050	1051	0	0	1	1	1	1	1	1	1	1
23	5	5-BW02	5-FW01	218	181	0	0	1	1	1	1	1	1	1	1
24	5	7-FW11	7-BW10	317	318	0	0	1	1	1	1	1	1	1	1
25	5	7-BW10	7-FW11	58	7	0	0	1	1	1	1	1	1	1	1
26	5	9-FW01	9-BW02	416	417	0	0	1	1	1	1	1	1	1	1
27	5	9-BW02	9-FW01	440	393	0	0	1	1	1	1	1	1	1	1
28	5	9-FW03	9-BW04	15	50	0	0	1	1	1	1	1	1	1	1
29	5	9-BW04	9-FW03	435	398	0	0	1	1	1	1	1	1	1	1
30	5	12-FW01	12-BW02	468	469	0	0	1	1	1	1	1	1	1	1
31	5	12-BW02	12-FW01	496	441	0	0	1	1	1	1	1	1	1	1
32	5	13-FW01	13-BW02	88	99	0	0	1	1	1	1	1	1	1	1
33	5	13-BW02	13-FW01	542	497	0	0	1	1	1	1	1	1	1	1
34	5	15-FW01	15-BW02	775	776	0	0	1	1	1	1	1	1	1	1
35	5	15-BW02	15-FW01	662	627	0	0	1	1	1	1	1	1	1	1
36	5	15-FW03	15-BW04	244	257	0	0	1	1	1	1	1	1	1	1
37	5	15-BW04	15-FW03	662	627	0	0	1	1	1	1	1	1	1	1
38	5	16-FW03	16-BW04	554	581	0	0	1	1	1	1	1	1	1	1
39	5	16-BW04	16-FW03	1024	1043	0	0	1	1	1	1	1	1	1	1
40	5	16-FW01	16-BW02	712	713	0	0	1	1	1	1	1	1	1	1
41	5	16-BW02	16-FW01	696	593	0	0	1	1	1	1	1	1	1	1
42	5	18-FW01	18-BW02	93	94	0	0	1	1	1	1	1	1	1	1
43	5	18-BW02	18-FW01	86	751	0	0	1	1	1	1	1	1	1	1
44	5	18-FW03	18-BW04	86	101	0	0	1	1	1	1	1	1	1	1
45	5	18-BW04	18-FW03	807	744	0	0	1	1	1	1	1	1	1	1

Table 6.19.: Verwendete Abstandsvorgaben für das KVB-Netz 2020

Linie	Hauptvarianten		Verstärkervarianten	
Linie 1	FW01	BW02	FW05	FW06
Linie 3	FW01	BW02		
Linie 4	FW01	BW02		
Linie 5	FW01	BW02		
Linie 7	FW01*	BW02*	FW11	BW10
Linie 9	FW01	BW02	FW03	BW04
Linie 12	FW01	BW02		
Linie 13	FW01	BW02		
Linie 15	FW01	BW02	FW03	FW04
Linie 16	FW01*	BW02*	FW03	BW04
Linie 18	FW01	BW02	FW03, FW05*	BW04, BW06*

* Verlängerung in Außenbereichen, Takt 20 Minuten

Tabelle 6.20.: Für die Optimierung verwendete Linienvarianten im KVB-Netz 2020

191,188, schlechtester Wert: 200,712), nach ca. 23 Minuten Rechenzeit verbessert sich dieser Wert im Laufe von 500 Generationen zu 176,140 (Durchschnitt: 181,286, schlechtester Wert: 194,224, siehe auch Abbildung 6.47). Der Branch-and-Bound-Algorithmus findet nach ca. 52 Stunden Rechenzeit 281.600 optimale Lösungen mit dem Zielfunktionswert 174,642.

6.4.3.2. Vergleich der Fahrpläne

Wie auch bei der Betrachtung der Netze von 2001 und 2012 werden je zehn Initial- und Optimalfahrpläne ausgewählt und jeweils zehn Simulationsläufe ausgeführt. Die beschriebenen Simulationsparameter und Erhebungsmethoden werden beibehalten.

Vergleich allgemeiner Merkmale Die Auswertung der Simulationsläufe zeigt ähnliche Muster wie schon in den Netzen 2001 und 2012: Die durchschnittliche Verspätung der einzelnen Abfahrten liegt bei der Verwendung der Initialfahrpläne bei 20,0 Sekunden, und sinkt bei der Benutzung der Optimalfahrpläne im Schnitt um 3,2 Sekunden oder 16,0% auf 16,8 Sekunden (siehe Abbildung 6.48). Werden pünktliche Abfahrten außer Acht gelassen, beträgt die mittlere Verspätung unter den initialen Fahrplänen 36,6 Sekunden, unter den optimalen Fahrplänen 31,4 Sekunden, also 5,2 Sekunden oder 14,3% weniger.

Auch bei den Häufigkeitsverteilungen der Abfahrtsverspätungen wird das bereits bekannte Muster sichtbar: Unter den Optimalfahrplänen treten mehr kleine Verspätungen auf als unter Initialfahrplänen (siehe Abbildung 6.49), bei Verspätungen von mehr als 60 Sekunden ist das Verhältnis umgekehrt (siehe Abbildung 6.50) - die durchschnittliche

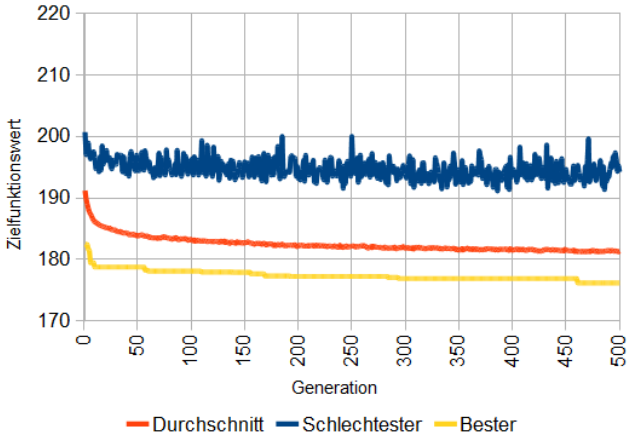


Abbildung 6.47.: Verlauf der Zielfunktion bei der Anwendung des Genetischen Algorithmus auf das KVB-Netz 2020

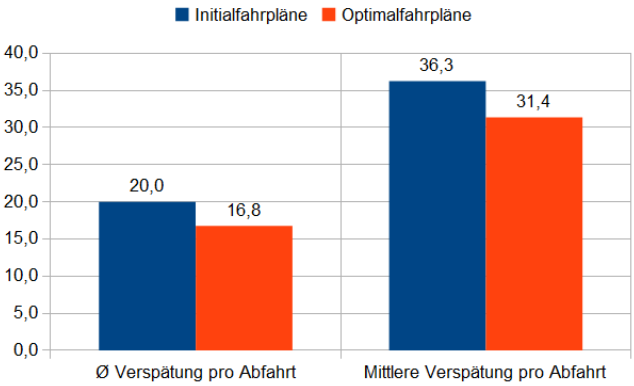


Abbildung 6.48.: Durchschnittliche und mittlere Verspätung pro Abfahrt (2020)

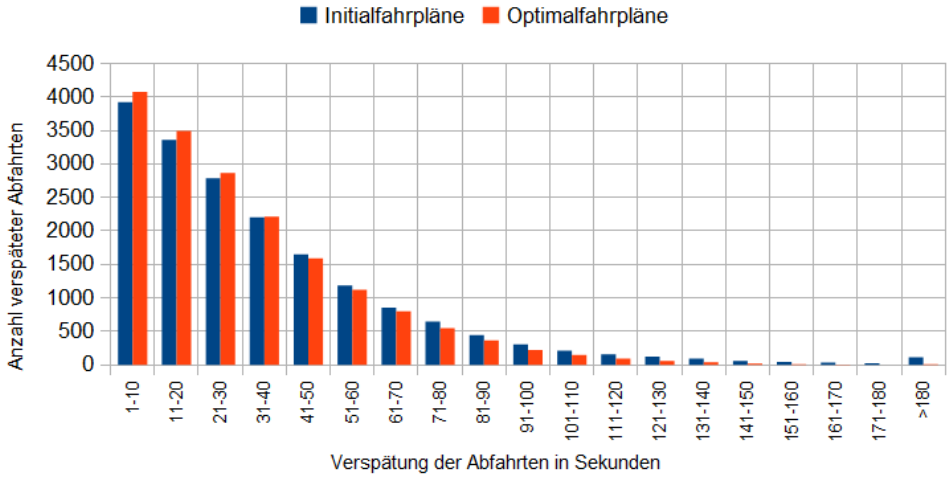


Abbildung 6.49.: Häufigkeitsverteilung der Verspätungen pro Abfahrt (2020)

Richtung	Linie															
	1	1A	3	4	5	7	9	9A	12	13	15	15A	16	16A	18	18A
FW	5	0	9	8	2	2	2	5	6	7	3	9	8	2	9	8
BW	4	0	8	6	6	3	7	8	9	8	7	5	8	0	9	4

Tabelle 6.21.: Fahrplan 109 - Initialfahrplan

Anzahl größerer Verspätungen sinkt von 3148,2 unter den initalen Fahrplänen um 803,4 Abfahrten oder 25,5% auf 2344,8 Abfahrten (siehe Abbildung 6.51).

Vergleich zweier Fahrpläne Um das Modellverhalten unter verschiedenen Fahrplänen betrachten zu können, wird auch hier wieder je ein Fahrplan aus den Pools der Initialfahrpläne (Fahrplan 109, siehe Tabelle 6.21) und Optimalfahrpläne (Fahrplan 145, siehe Tabelle 6.22) ausgewählt und jeweils als Grundlage für 100 Simulationsläufe genutzt.

Die wesentlichen Änderungen im Vergleich zum Stadtbahnnetz von 2012 sind natürlich die Inbetriebnahme des Nord-Süd-Tunnels, der die Kölner Südstadt direkt an die Innenstadt und den Hauptbahnhof anbindet. Der Tunnel wird von den mit neuen Verläufen versehenen Linien 5 und 16 befahren, die daher näher betrachtet werden sollen.

Verspätung an ausgewählten Haltepunkten Die durchschnittliche Abfahrtsverspätung an den schon bei den Netzen 2001 und 2012 betrachteten Stationen verringert sich unter dem Optimalfahrplan im Vergleich zum Initialfahrplan deutlich (siehe Abbildung 6.52). Zusätzlich in die Betrachtung aufgenommen wurden die unterirdischen Haltepunkte der

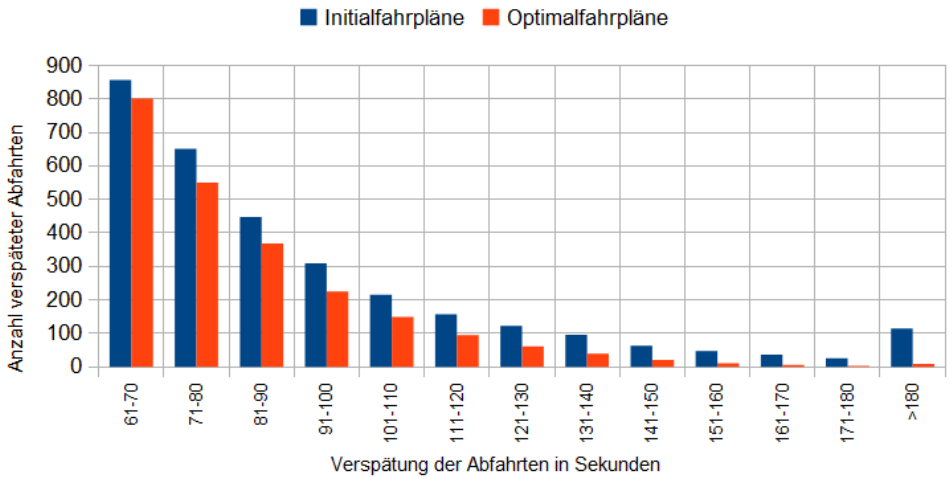


Abbildung 6.50.: Häufigkeitsverteilung größerer Verspätungen pro Abfahrt (2020)

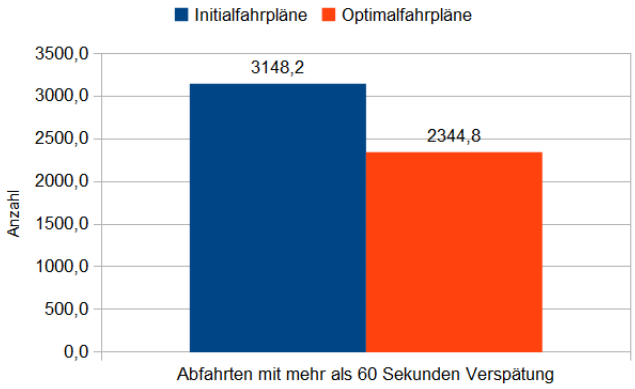


Abbildung 6.51.: Anzahl Abfahrten mit mehr als 60 Sekunden Verspätung (2020)

Richtung	Linie															
	1	1A	3	4	5	7	9	9A	12	13	15	15A	16	16A	18	18A
FW	5	0	2	1	0	4	9	1	5	1	2	6	7	3	8	7
BW	6	2	1	9	6	6	3	4	7	4	5	1	0	8	8	4

Tabelle 6.22.: Fahrplan 145 - Optimalfahrplan

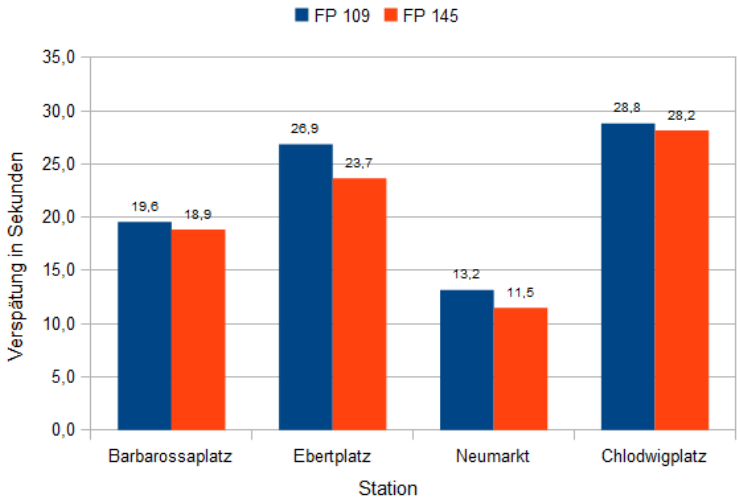


Abbildung 6.52.: Vergleich der Verspätungen an ausgewählten Stationen (Fahrpläne 109 und 145)

Station Chlodwigplatz, die Teil des Nord-Süd-Tunnels sind. Hier ist im Durchschnitt eine kleine Verbesserung von 0,63 Sekunden oder 2,2% zu beobachten.

Die Tabelle 6.23 und Abbildung 6.53 zeigen die Verspätung der Abfahrten an den betrachteten Stationen aufgeschlüsselt nach Haltepunkten. Im Durchschnitt über die betrachteten Haltepunkte fällt die Verspätung von 21,2 unter Fahrplan 109 um 1,7 Sekunden oder 8% auf 19,5 Sekunden unter Fahrplan 145. Die Abbildungen 6.54, 6.55, 6.56 und 6.57 zeigen die geplanten Abfahrtszeiten.

Abbildung 6.58 zeigt die Abfahrtsverspätung an den südwärts gerichteten Haltepunkten des Nord-Süd-Tunnels. Die Verspätung an den nordwärts gerichteten Haltepunkten wird in Abbildung 6.59 dargestellt.

Es fällt auf, dass die Verspätung im Tunnel kaum von den verwendeten Fahrplänen abhängt. Ein weiterer Blick auf Abbildung 6.57 zeigt allerdings, dass die Fahrpläne 109 und 145 hier auch keine starken Unterschiede aufweisen: Lediglich an den nach Norden gerichteten Haltepunkten (hier vertreten durch CWP-1038) kann jede zweite Fahrt der 16 unter Fahrplan 109 von einer gegebenenfalls mehr als 60 Sekunden verspäteten Fahrt der Linie 5 beeinflusst werden. Alle anderen Fahrten haben unter Fahrplan 109 einen geplanten Abstand von mindestens zwei Minuten, unter Fahrplan 145 sogar von mindestens drei Minuten. Bei drei zu beachtenden Fahrten pro zehnminütigem Intervall ist keine bessere Planung möglich.

Trotz optimaler Planung bleibt (abgesehen vom südlichen Einfahrtspunkt MAS-1043)

Station	Haltepunkt	ØVerspätung		Abs. Verb.	Rel. Verb.
		FP 109	FP 145		
Barbarossaplatz	BAB-227	36,8	36,6	0,2	0,01
	BAB-274	2,5	0,1	2,4	0,97
	BAB-449	39,0	38,4	0,6	0,01
	BAB-488	0,0	0,4	-0,3	-11,32
Ebertplatz	EBP-198	26,6	24,0	2,6	0,10
	EBP-201	40,6	40,2	0,4	0,01
	EBP-233	4,3	2,4	1,9	0,44
	EBP-268	35,9	28,0	7,9	0,22
Neumarkt	NEU-12	0,1	0,1	-0,1	-0,58
	NEU-53	45,9	39,1	6,8	0,15
	NEU-77	0,6	0,8	-0,3	-0,46
	NEU-110	6,0	5,8	0,2	0,04
Chlodwigplatz (U)	CWP-1038	28,2	27,0	1,2	0,04
	CWP-1039	29,4	29,3	0,1	0,00
Durchschnitt		21,2	19,5	1,7	0,08

Tabelle 6.23.: Verspätung an ausgewählten Haltepunkten in Sekunden (Fahrpläne 109 und 145)

eine relativ hohe Sockelverspätung von 19,3 bis 37,5 Sekunden bestehen. Um diesen zu verstehen, wird die Verspätungsentwicklung von Fahrten der Linien 16 und 5 näher betrachtet.

Verspätung der Linien Die meisten Linien gewinnen 5 bis 15% an Pünktlichkeit (siehe Abbildung 6.60 und Tabelle 6.25). Die über die Linien gemessene Verspätung beträgt unter dem Initialfahrplan im Durchschnitt 17,4 Sekunden und sinkt unter dem gewählten Optimalfahrplan um 1,3 Sekunden oder 7,4% auf 16,1 Sekunden.

Ausnahmen bilden wie schon in den Netzen 2001 und 2012 die Linien 3, 4 und 5, die teilweise Strecken gemeinsam nutzen. Eine weitere Ausnahme bildet Linie 13, die im zufällig generierten Initialfahrplan bereits sehr günstig gelegt ist und wie in den bisher betrachteten Netzen unter dem optimalen Fahrplan einen durchschnittlichen Verspätungswert von etwas mehr als 22 Sekunden erreicht.

Verspätungsentwicklung der Linie 16 Von Niehl/Sebastianstraße kommend verlässt die Linie 16 hinter der Station Breslauer Platz wie bereits beschrieben den aus dem Netz 2012 bekannten Verlauf und führt gemeinsam mit Linie 5 durch den Nord-Süd-Tunnel bis zur Station Marktstraße. Nach dieser Station verlässt die Linie die Neubaustrecke und

Station	Haltepunkt	ØVerspätung		Abs. Verb.	Rel. Verb.
		FP 109	FP 145		
Rathaus	RAT-1032	30,4	29,0	1,4	0,05
	RAT-1033	23,4	23,2	0,2	0,01
Heumarkt (U)	HMG-1030	20,8	19,5	1,3	0,06
	HMG-1031	37,0	36,9	0,1	0,00
Severinsstraße (U)	SEV-1034	36,9	35,7	1,2	0,03
	SEV-1035	20,8	20,7	0,1	0,00
Kartäuserhof	KAH-1036	35,5	34,3	1,2	0,03
	KAH-1037	21,8	21,7	0,1	0,00
Chlodwigplatz (U)	CWP-1038	28,2	27,0	1,2	0,04
	CWP-1039	29,4	29,3	0,1	0,00
Bonner Wall	BOW-1040	38,0	36,9	1,1	0,03
	BOW-1041	19,5	19,3	0,1	0,04
Marktstraße	MAS-1042	40,4	37,5	2,9	0,07
	MAS-1043	1,6	1,5	0,1	0,04
Durchschnitt		27,4	26,6	0,8	0,02

Tabelle 6.24.: Verspätung an den Haltepunkten des Nord-Süd-Tunnels (Fahrpläne 109 und 145)

Linie	ØVerspätung		Abs. Verb.	Rel. Verb.
	FP 109	FP 145		
1	21,3	19,5	1,8	0,08
3	11,7	11,4	0,3	0,02
4	5,8	6,0	-0,2	-0,04
5	8,4	8,1	0,3	0,04
7	15,3	13,3	2,0	0,13
9	15,1	13,4	1,8	0,12
12	15,7	13,3	2,4	0,15
13	22,9	22,7	0,2	0,01
15	23,4	20,7	2,7	0,12
16	29,7	28,3	1,4	0,05
18	21,5	20,2	1,3	0,06
Durchschnitt	17,4	16,1	1,3	0,07

Tabelle 6.25.: Verspätung der Linien in Sekunden (Fahrpläne 109 und 145)

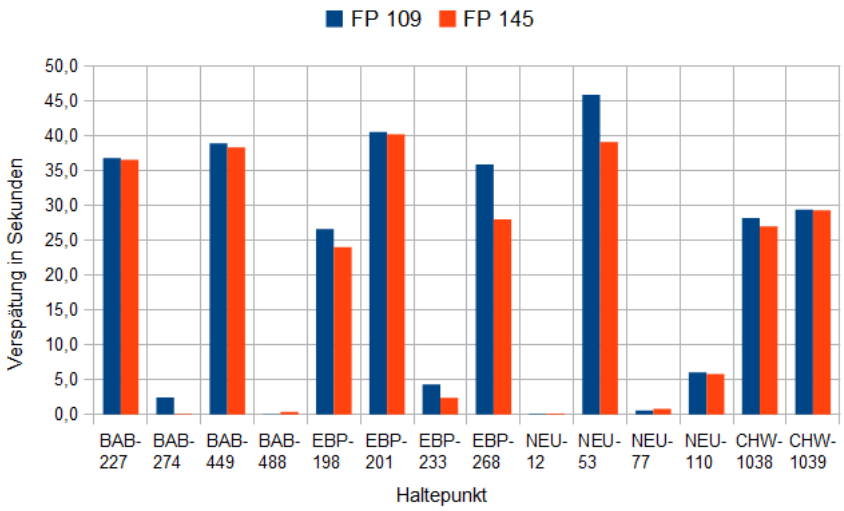
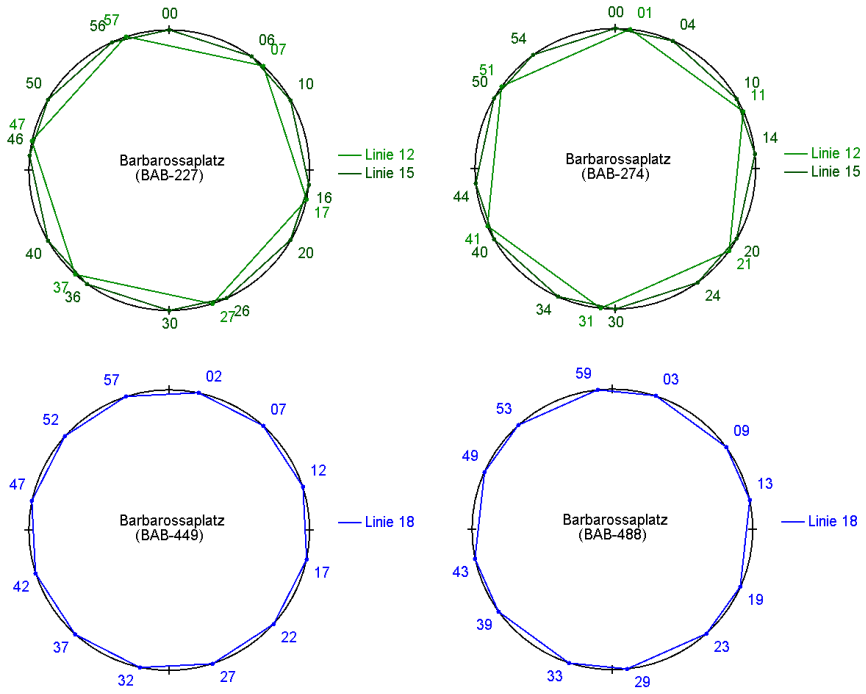


Abbildung 6.53.: Vergleich der Verspätungen an ausgewählten Haltepunkten (Fahrpläne 109 und 145)

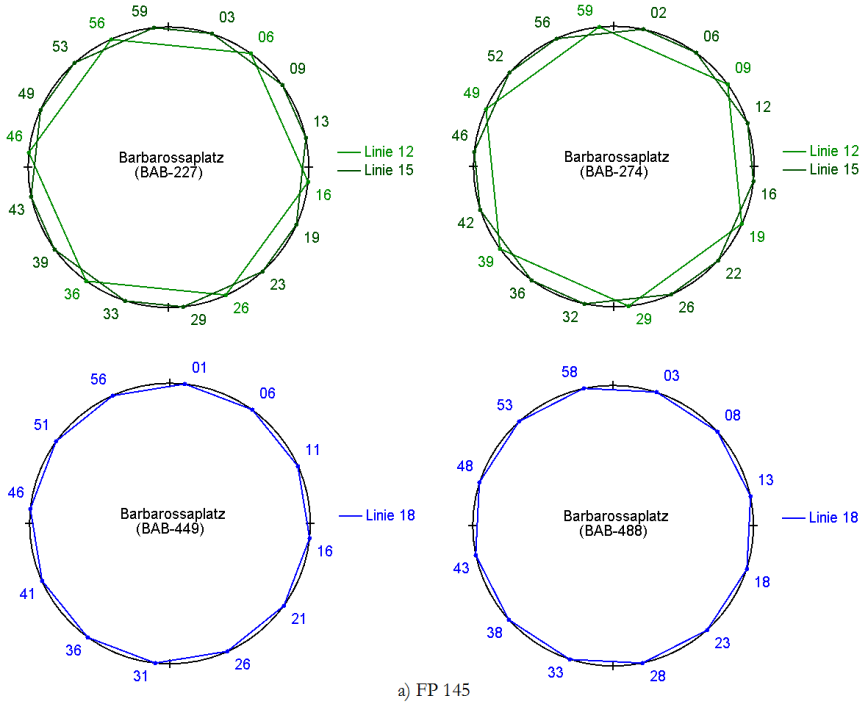
bedient ab der Station Schönhauser Straße wieder den aus 2012 bekannten Verlauf entlang des Rheinufers in Richtung Bonn. Diese langen Varianten der Linie 16 werden durch die ebenfalls im Zehn-Minuten-Takt fahrenden Varianten 16-FW03 und 16-BW04 verstärkt, die zwischen Reichensperger Platz im Norden und Marktstraße im Süden pendeln. So entsteht im Innenstadtbereich ein fünfminütiger Takt.

Die Fahrten der Linie 16 werden vom Optimierer unter Fahrplan 109 zwölf, unter Fahrplan 145 elf Fahrzeugen zugeordnet (siehe Abbildung 6.61). Es fallen sofort zwei Verspätungsniveaus ins Auge: die höheren Durchschnittsverspätungen von etwa 35 Sekunden treten bei Fahrzeugen auf, die die langen Linienvarianten 16-FW01 und 16-BW02 bedienen; die Fahrzeuge der kürzeren Varianten verspäten sich bei ihren Abfahrten durchschnittlich zwischen 20 und 25 Sekunden. Zudem scheint die Verspätung zumindest bei den langen Varianten fast vollständig unabhängig vom eingesetzten Fahrplan zu sein. Im Folgenden wird daher mit Umlauf 1604 ein Fahrzeug näher betrachtet, das die lange Strecke bedient.

Die Durchschnittsverspätungen der einzelnen Fahrten des Umlaufs 1604 (siehe Abbildung 6.62) zeigen das bereits bekannte Muster: Je nach Linienrichtung liegen die Werte auf unterschiedlichen Niveaus, hier für die Fahrten in Richtung Süden (Fahrt 1160403 und weitere Fahrten mit ungeradem Bezeichner) bei durchschnittlich 44,7 Sekunden unter Fahrplan 109 und 46,3 Sekunden unter Fahrplan 145, für die Fahrten nach Norden (Fahrt 1160404 und weitere Fahrten mit geradem Bezeichner) bei 25,6 unter Fahrplan

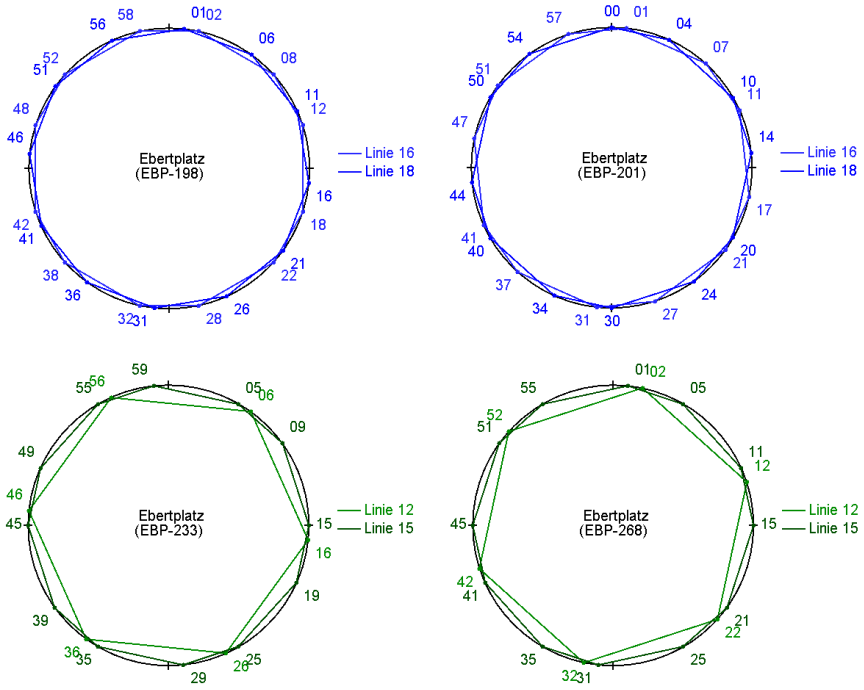


a) FP 109

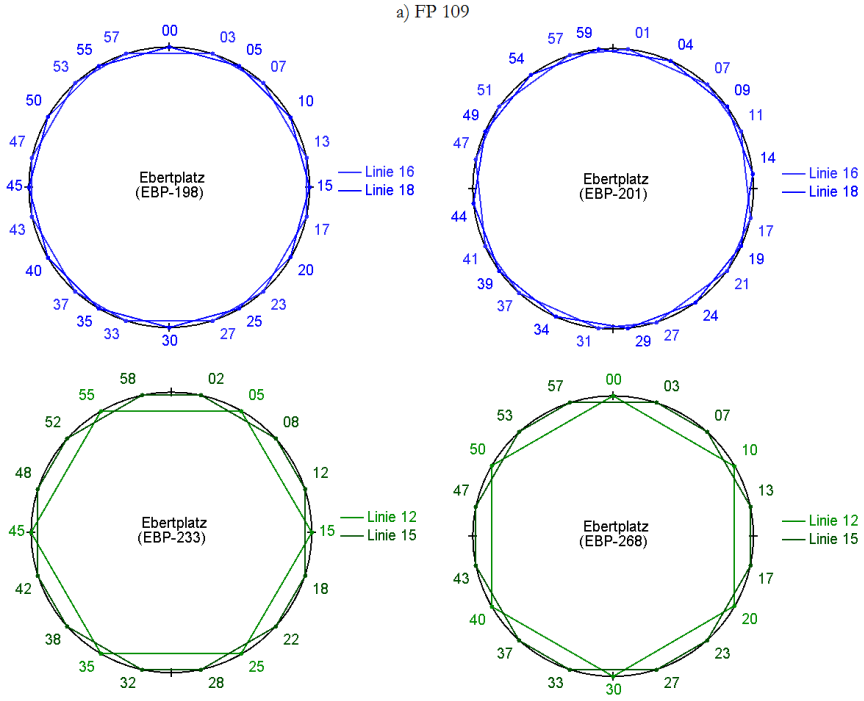


a) FP 145

Abbildung 6.54.: Fahrpläne 109 und 145: Abfahrten an der Station Barbarossaplatz



a) FP 109



a) FP 145

Abbildung 6.55.: Fahrpläne 109 und 145: Abfahrten an der Station Ebertplatz

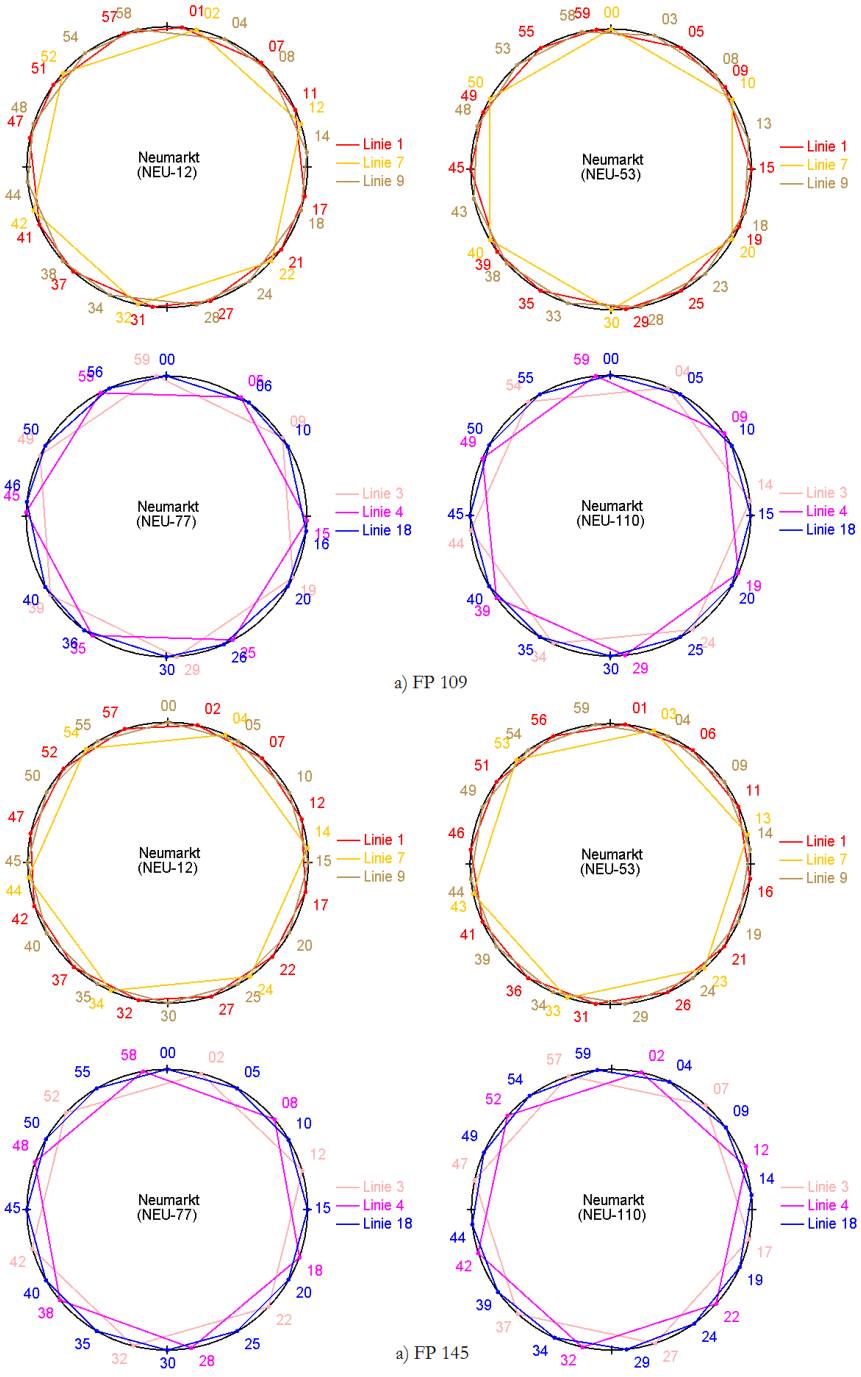


Abbildung 6.56.: Fahrpläne 109 und 145: Abfahrten an der Station Neumarkt

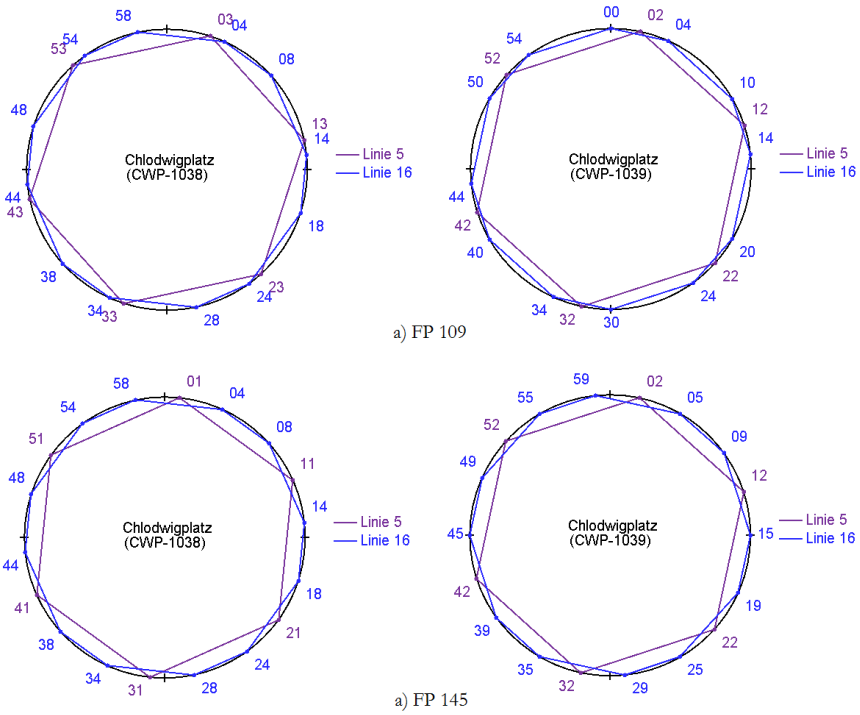


Abbildung 6.57.: Fahrpläne 109 und 145: Abfahrten an den unterirdischen Haltepunkten der Station Chlodwigplatz

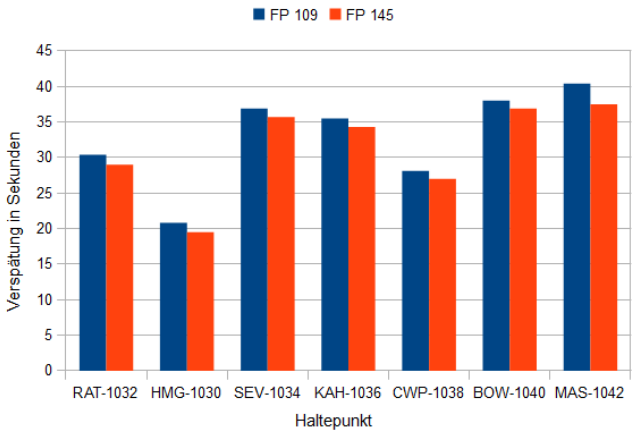


Abbildung 6.58.: Verspätung an den Haltepunkten des Nord-Süd-Tunnels, Richtung Süden

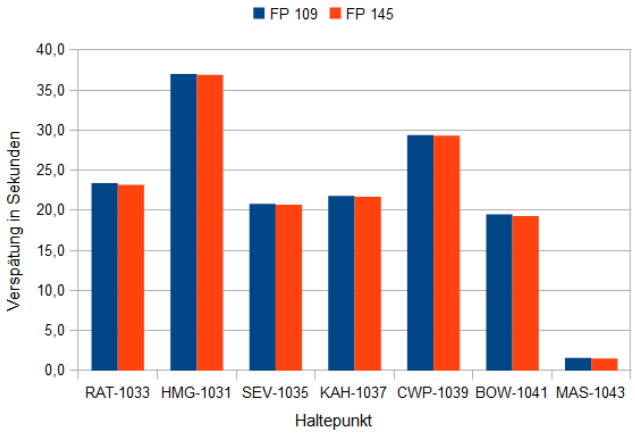


Abbildung 6.59.: Verspätung an den Haltepunkten des Nord-Süd-Tunnels, Richtung Norden

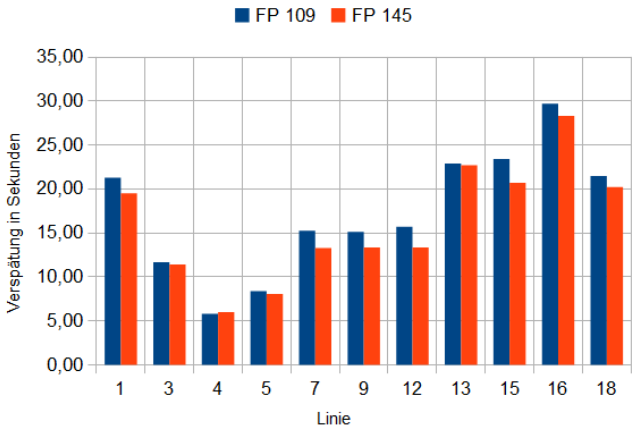


Abbildung 6.60.: Verspätung der Linien in Sekunden (Fahrpläne 109 und 145)

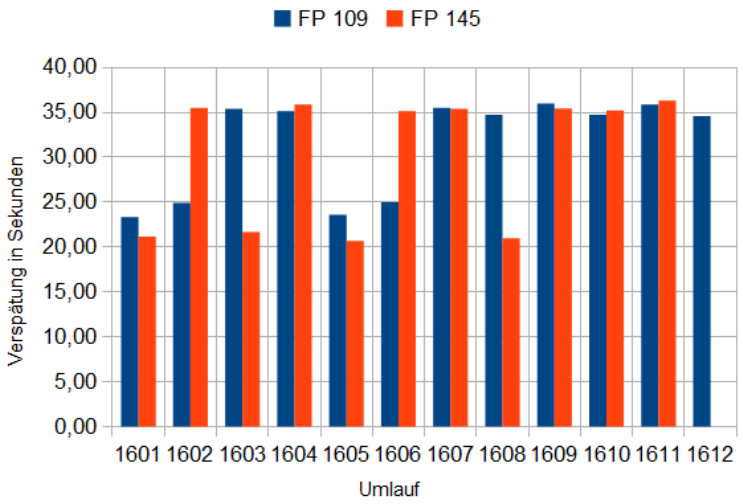


Abbildung 6.61.: Fahrpläne 109 und 145: Durchschnittsverspätungen der Umläufe der Linie 16

109 und 25,5 Sekunden unter Fahrplan 145. Im Durchschnitt beträgt die Abfahrtverspätung unter Fahrplan 109 35,2 Sekunden, unter Fahrplan 145 liegt sie mit 35,9 Sekunden um 0,7 Sekunden oder 2% höher.

Im Folgenden werden die beiden ersten im Messzeitraum erfassten Fahrten, also 1160403 und 1160404, näher betrachtet. Dabei wird insbesondere auf die Streckenabschnitte im Nord-Süd-Tunnel eingegangen.

Die Verspätung von Fahrt 1160403 (siehe Abbildung 6.63) in Richtung Sürth entwickelt sich unter beiden verwendeten Fahrplänen sehr ähnlich. Es zeigen sich leichte Vorteile für Fahrplan 109 im Bereich des Nord-Süd-Tunnels, die aber bei der nachfolgenden Überlandfahrt wieder ausgeglichen werden. Die durchschnittliche Verspätung der Abfahrten liegt unter Fahrplan 109 bei 46,3 Sekunden, unter Fahrplan 145 bei 46,4 Sekunden.

Im Bereich des Tunnels verändert sich die Pünktlichkeit des Fahrzeugs eher mäßig. Die vorhandene Verspätung nimmt von Breslauer Platz über Rathaus nach Heumarkt um etwa 24 Sekunden ab, um dann von Heumarkt bis Severinstraße erneut um etwa 16 Sekunden zu steigen. Bis zur Ausfahrt aus dem Tunnel nach der Station Bonner Wall hält sich dann die Verspätung, trotz einer kurzfristigen, kleinen Verbesserung an der Station Chlodwigplatz. Die Verspätungsentwicklung verläuft also analog zum Durchschnitt auf dieser Strecke (siehe nochmals Abbildung 6.58), allerdings auf höherem Niveau.

Innerhalb des Tunnels entwickelt sich der Verspätungswert der Fahrt 1160404 analog zum Verhalten der Vorgängerfahrt (siehe Abbildung 6.64): Nach der Einfahrt in den

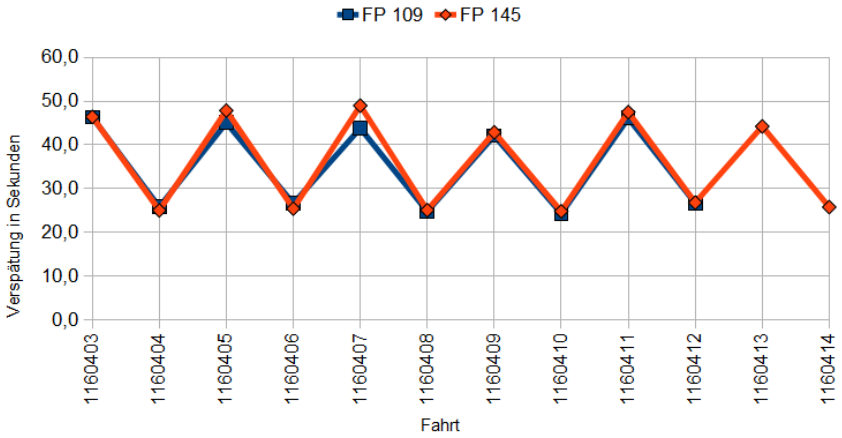


Abbildung 6.62.: Fahrpläne 109 und 145: Durchschnittsverspätungen der Fahrten des Umlaufs 1604

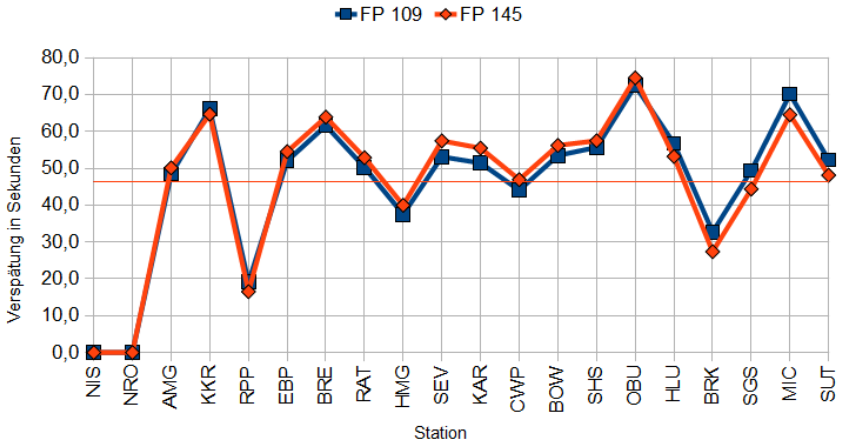


Abbildung 6.63.: Fahrpläne 109 und 145: Fahrzeug 1604, Fahrt 1160403

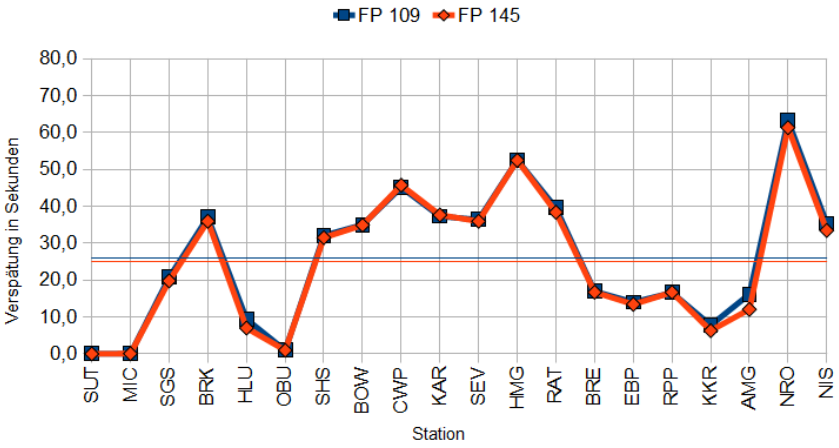


Abbildung 6.64.: Fahrpläne 109 und 145: Fahrzeug 1604, Fahrt 1160404

Tunnel zwischen Schönhauser Straße und Bonner Wall steigt die Verspätung leicht an, um dann nördlich der Station Chlodwigplatz wieder abzunehmen. Zwischen Severinsstraße und Heumarkt nimmt die Verspätung wie schon in Fahrt 1160403 leicht zu, um dann, wie bei der Vorgängerfahrt, bis zur Station Rathaus wieder ungefähr auf das bei der Einfahrt in den Tunnel vorhandene Niveau abzusinken.

Die Zu- und Abnahme der Werte entspricht den in Abbildung 6.59 gezeigten durchschnittlichen Abfahrtsverspätungen an den nach Norden gerichteten Haltepunkten des Tunnels.

Verspätungsentwicklung der Linie 5 Der Verlauf der Linie 5 im Netz 2020 gleicht im Westteil dem im Netz von 2012. Östlich der Station Dom/Hbf verlässt die Linie allerdings, wie bereits in Abbildung 6.45 gezeigt, die bisherige Linienführung, wendet sich südwärts und bedient beginnend mit dem Haltepunkt RAT-1032 die Stationen des Nord-Süd-Tunnels. Südlich der Station Marktstraße endet der Tunnel, die Linie 5 befährt die Stationen des Bauabschnitts 3 und endet an der Station Arnoldshöhe am Bonner Verteilerkreis.

Die Fahrten der Linie 5 werden durch neun Fahrzeuge durchgeführt, die unter Fahrplan 109 im Schnitt 8,4 Sekunden verspätet sind (siehe Abbildung 6.65). Die Verspätung reduziert sich unter dem als optimal bewerteten Fahrplan 145 um 0,3 Sekunden oder 4% auf 8,1 Sekunden.

Beim näheren Betrachten eines Umlaufs (hier Fahrzeug 501) zeigt sich wieder das bekannte Muster (siehe Abbildung 6.66): Die Fahrten einer Linienvariante (in diesem Fall 5-FW01) sind etwas stärker verspätet als die der anderen Variante (hier 5-BW02).

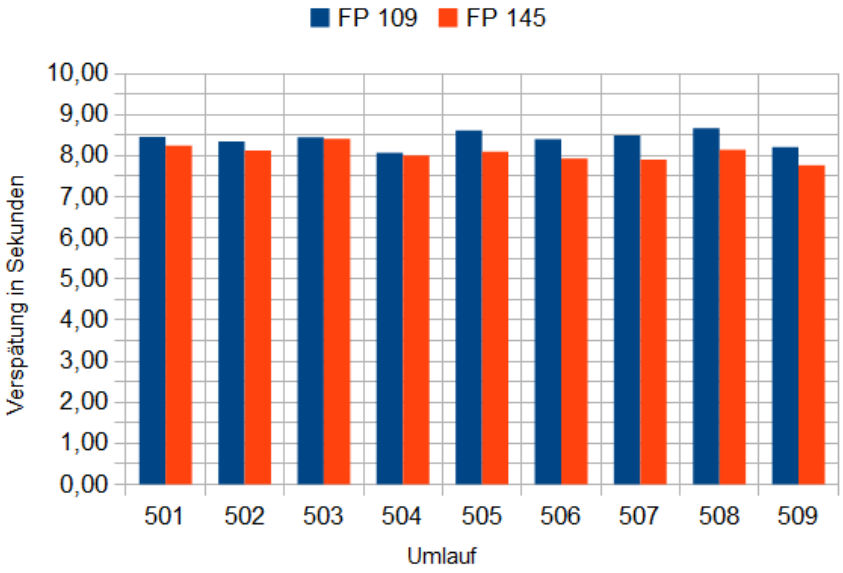


Abbildung 6.65.: Fahrpläne 109 und 145: Durchschnittsverspätungen der Umläufe der Linie 5

Die durchschnittliche Verspätung der einzelnen Fahrten des Fahrzeugs 501 schwanken zwischen 6,7 und 10,3 Sekunden und weichen unter den verschiedenen Fahrplänen kaum voneinander ab.

Bei der Fahrt 1050103 (siehe Abbildung 6.67) fährt das Fahrzeug vor der Station Rathaus unter Fahrplan 145 mit einer durchschnittlichen Verspätung von 1,0 Sekunden südwärts in den neuen Tunnel ein. Die Verspätung sinkt bis zur Station Heumarkt fast auf Null (0,1 Sekunden Durchschnittswert bleiben), um dann wie auf dieser Strecke schon bei den anderen beschriebenen Fahrten beobachtet zwischen Heumarkt und Severinsstraße leicht anzusteigen. Bis zum Chlodwigplatz sinkt der Wert dann wieder, um zur Station Bonner Straße hin wieder zuzunehmen. Hier trennt sich die Linie 5 von der zum Rheinufer führenden Linie 16. Auf der 450 Meter langen Strecke zur Marktstraße, für die eine Minuten Fahrzeit geplant sind, steigt die Verspätung durchschnittlich um 9,6 auf 29,0 Sekunden, dem bei dieser Fahrt höchsten Wert. Auf dem Weg bis zur Endstation kann die Bahn einen Teil dieser Verspätung wieder abbauen, sie erreicht die Station mit 24,4 Sekunden Verzug.

Die Verspätung im Tunnel ist unter dem Initialfahrplan 109 etwas geringer als unter dem Optimalfahrplan 145. Durch die unregelmäßige Verteilung der Abfahrten an den südwärts gerichteten Haltepunkten im Tunnel (siehe nochmals Abbildung 6.57) hat die Linie 5 unter Fahrplan 109 einen zeitlichen Puffer von fünf Minuten zu den vor ihr her

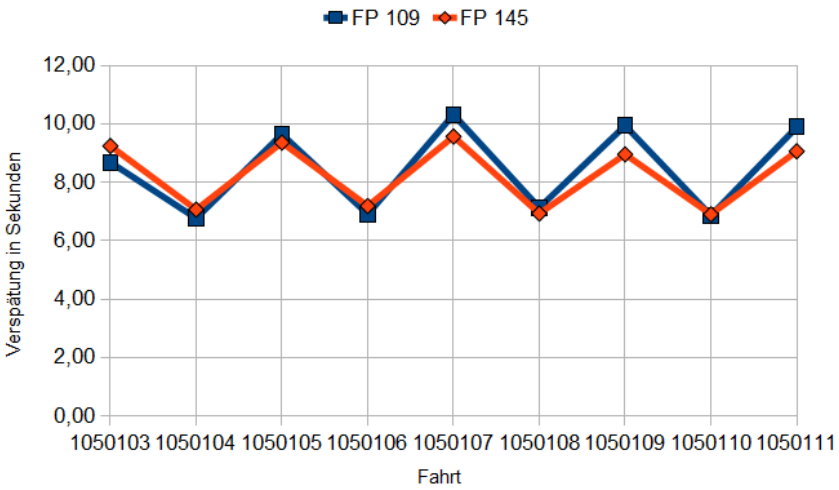


Abbildung 6.66.: Fahrpläne 109 und 145: Durchschnittsverspätungen der Fahrten des Umlaufs 501

fahrenden Fahrzeugen der Linie 16, unter Fahrplan 145 sind es nur drei Minuten.

Die Rückfahrt 1050104 des Fahrzeuges 501 verläuft unter den Fahrplänen 109 und 145 fast identisch (siehe Abbildung 6.68). Die Fahrt beginnt mit dem Bauabschnitt 3 der Neubaustrecke an der Station Arnoldshöhe. Die Bahn startet pünktlich, es besteht also ausreichend Puffer nach der verspäteten Ankunft der vorhergehenden Fahrt. Wie schon bei Fahrt 1040103 steigt die Verspätung zwischen Arnoldshöhe und Bonner Straße/Gürtel etwas an (unter Fahrplan 145 bis auf 8,4 Sekunden), um dann bis Marktstraße wieder etwas abzusinken. Auf den beiden folgenden Strecken bis zum Chlodwigplatz steigt die Verspätung bis auf 22,4 Sekunden. Zwischen Severinsstraße und Heumarkt steigt die Verspätung wieder an - auch das wurde für andere Fahrten bereits beschrieben. Zwischen Heumarkt und Rathaus sinkt der Wert um 13,9 auf 15,6 Sekunden.

Das Auf und Ab der Verspätungswerte im Nord-Süd-Tunnel verhält sich also analog zu dem der Fahrt 105013. Es entspricht ebenfalls der Betrachtung der Durchschnittswerte über die nach Norden gerichteten Haltepunkte im Tunnel (siehe nochmals Abbildung 6.59).

Zusammenfassung Das betrachtete Netz unterscheidet sich gegenüber dem Netz von 2012 durch die Aufnahme des Nord-Süd-Tunnels in das Modell und die damit einher gehende Veränderung der Verläufe der Linien 5 und 16.

Der Optimierer verbessert die Zielfunktionswerte von den Initiallösungen bis zu den optimalen Lösungen von 200,712 um 13,0% oder 26,07 Punkte auf den Wert 174,642.

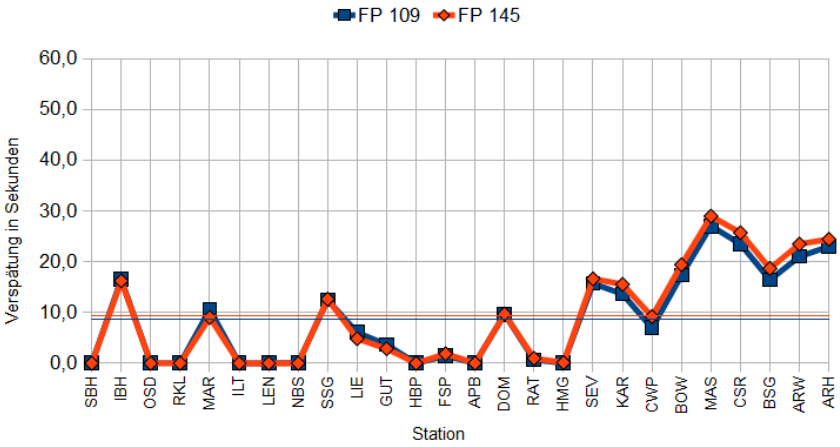


Abbildung 6.67.: Fahrpläne 109 und 145: Fahrzeug 501, Fahrt 1050103

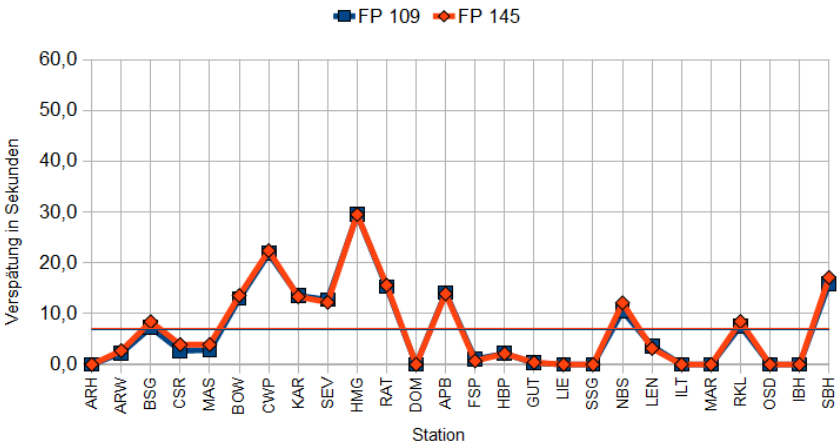


Abbildung 6.68.: Fahrpläne 109 und 145: Fahrzeug 501, Fahrt 1050104

Auffällig ist die im Vergleich zu den anderen Netzen große Zahl von 281.600 gefundenen besten Lösungen. Offenbar gibt es eine Reihe von Plateaus in der Zielfunktion, so dass in bestimmten Bereichen des Lösungsraums viele, sich kaum unterscheidende Optimallösungen gefunden werden. Die Laufzeit des Branch-and-Bound-Lösers ist im Vergleich zu den anderen Netzen mit 52 Stunden relativ hoch. Anscheinend gelingt es dem Algorithmus nicht, frühzeitig gute Abschätzungen für die lokale untere Schranke zu finden.

Die durchschnittliche Verspätung der Abfahrten reduziert sich von den initialen zu den optimalen Fahrplänen von 20,0 um 3,2 Sekunden oder 16,0% auf 16,8 Sekunden. Die Anzahl der um mehr als 60 Sekunden verspäteten Abfahrten geht von 3.148,2 um 803,4 Abfahrten oder 25,5% auf 2.344,8 Abfahrten zurück. Die Betrachtung der Simulationsergebnisse für die ausgewählten Fahrpläne 109 und 145 zeigt, dass die über die Linien und die Haltepunkte gemessene Verspätung unter dem optimalen Fahrplan zumeist geringer ist als unter dem initialen Fahrplan. Die Unterschiede der Pünktlichkeit sind unter den ausgewählten Fahrplänen allerdings im Vergleich zu den bei den Netzen von 2001 und 2012 beobachteten eher gering. Die Verspätung an den einzelnen Haltepunkten des Tunnels ist weitgehend unabhängig von den verwendeten Fahrplänen, sie wird durch die beschriebenen Eigenschaften der einzelnen Teilstrecken bestimmt. Die Werte schwanken zwischen den einzelnen Haltepunkten nicht stark; die Fahrzeuge sind bei der Ausfahrt aus dem Tunnel etwas unpünktlicher als bei der Einfahrt.

Es ist davon auszugehen, dass für die Neubaustrecken neuere Fahrzeugtypen (wie die Reihen Vossloh Kiepe K4500 und K5000) eingesetzt werden. Durch deren besseres Beschleunigungs- und Bremsvermögen werden die realen Verspätungswerte auf den relativ kurzen Untergrundstrecken sicherlich geringer sein als in der Simulation beobachtet.

6.4.4. Nebeneinanderstellen der Netze

Die Ergebnisse von Optimierung und Simulation können wegen der unterschiedlichen Ausgangsbedingungen über die betrachteten Netze hinweg nicht direkt verglichen werden. Im Laufe des Untersuchungszeitraums wurde für viele Linien die Route verändert, Strecken und Haltepunkte wurden ergänzt, andere gestrichen. Dazu kommt, dass für jedes Netz je ein optimaler und ein zufällig erstellter, gültiger Fahrplan ausgewählt und verglichen wurden. Die optimalen Fahrpläne für ein Netz (und erst recht die zufälligen) können jedoch sehr verschiedenen sein, obwohl sie auf einem gemeinsamen Niveau der Zielfunktion liegen. Es ist also nicht sicher, welcher Anteil am Zustandekommen der gemessenen Werte auf die gewählten Fahrpläne zurück zu führen ist und welcher auf die Eigenschaften des zugrunde liegenden Netzes.

Um die Anwendbarkeit der Optimierungssoftware zu demonstrieren wurde ein Satz

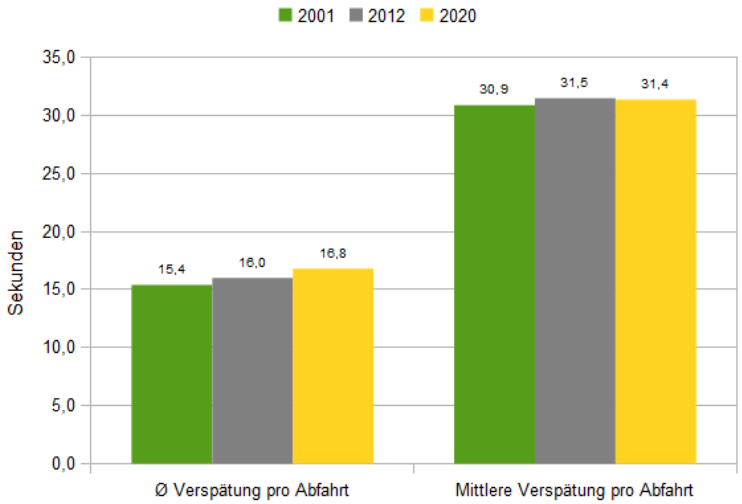


Abbildung 6.69.: Netze 2001, 2012 und 2020: Durchschnittliche und mittlere Verspätung pro Abfahrt

plausibel erscheinender planerischer Vorgaben gewählt, die jedoch kaum mit den real verwendeten überein stimmen werden. Hinzu kommt, dass eine Reihe von Simulationsparametern, wie die Schaltzeiten der Lichtsignalanlagen, Trödelwahrscheinlichkeit und -ausmaß mangels besseren Wissens willkürlich gewählt wurden. Schließlich wurden auch für die Netze 2012 und 2020 in der Simulation ausschließlich Fahrzeuge des Typs K4000 modelliert. Bereits in 2012 werden allerdings Fahrzeuge der neueren Typen K4500 und K5000 eingesetzt. Es ist also davon auszugehen, dass die Verspätungswerte für die neuen Netze in der Realität kleiner sind als in der Simulation beobachtet. Die gezeigten Werte sollten daher auf keinen Fall als Grundlage für Handlungsempfehlungen verwendet werden.

Für das Netz von 2001 wurden pro Simulationslauf insgesamt 32.633,4 Abfahrten gemessen, die durchschnittlich um 15,4 Sekunden verspätet sind (siehe Abbildung 6.69). Zum Netz von 2012 sinkt die Anzahl der Abfahrten um 764,4 Abfahrten oder 2,4% auf 31.869,0 mit durchschnittlich 16,0 Sekunden Verspätung. Das letzte betrachtete Netz von 2020 weist pro Lauf durchschnittlich 33.150,5 Abfahrten mit 16,8 Sekunden Verspätung auf. Im Vergleich zu 2012 ist das ein Zuwachs vom 4,0% oder 1281,5 Abfahrten. Die angegebenen Werte beziehen sich dabei immer auf die im jeweiligen Netz verwendeten Optimalfahrpläne.

Ein Blick auf die Häufigkeitsverteilung der Abfahrtsverspätungen (siehe Abbildungen 6.70 und 6.71) zeigt, dass das Netz von 2012 in fast allen Verspätungsklassen die je-

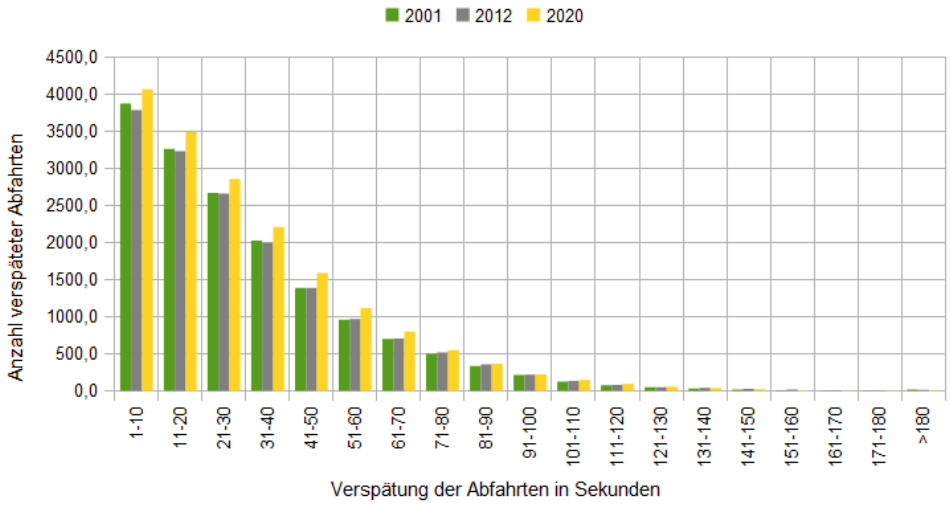


Abbildung 6.70.: Netze 2001, 2012 und 2020: Häufigkeitsverteilung der Verspätung pro Abfahrt

weils höchsten Werte liefert. Bei Verspätungen über 140 Sekunden sind die beobachteten Häufigkeiten für das Netz von 2020 allerdings geringer als für das Netz von 2012. Im Vergleich zum Netz von 2001 sind die bei den Versuchen mit dem Netz von 2012 beobachteten Häufigkeiten bei kleinen Verspätungen bis zu 40 Sekunden etwas geringer. Verspätungen größer 60 Sekunden treten wiederum im Netz von 2012 häufiger auf.

Die Anzahl der Abfahrten mit mehr als 60 Sekunden Verspätung steigt ebenfalls an: Für das Netz 2001 wurden 2.108,0 verspätete Abfahrten beobachtet, für 2012 2.198,5 und für 2020 2.344,8 Abfahrten (siehe Abbildung 6.72). Veränderungen um 4,3 und 6,7% stehen also Änderungen der Gesamtzahl der Abfahrten von -2,4 und 4,0% gegenüber.

Abbildung 6.73 und Tabelle 6.26 stellen die Durchschnittsverspätung der einzelnen Linien dar. Auch hier ist zu bedenken, dass die Linienführung sich im Betrachtungszeitraum deutlich geändert hat: Von Jahr 2001 bis 2012 wurden die Linien 1, 3 und 5 verlängert, die Verläufe der Linien 12, 15, 16 und 18 verändert, die Linien 9, 15 und 18 mit innerstädtischen Verstärkerfahrten versehen und im Gegenzug die Linien 6, 8 und 19 aufgegeben. Zum Jahr 2020 werden die Verläufe der Linien 5 und 16 verändert und durch den Nord-Süd-Tunnel geführt.

Trotz der beschriebenen Veränderungen sind die Verspätungen der die Ost-West-Achse bildenden Linien 1, 7 und 9 in den drei betrachteten Netzen in etwa gleich geblieben. Der Wegfall der in 2001 noch vorhandenen Linie 8 wurde kompensiert durch Verstärkerfahrten der Linie 9, so dass die Belastung der Streckenabschnitte in weiten Teilen gleich

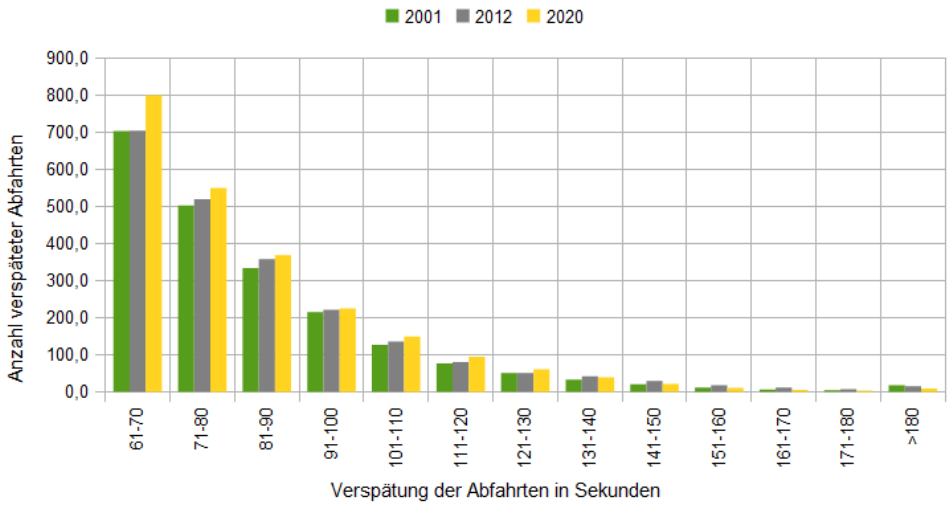


Abbildung 6.71.: Netze 2001, 2012 und 2020: Häufigkeitsverteilung der größeren Verspätungen pro Abfahrt

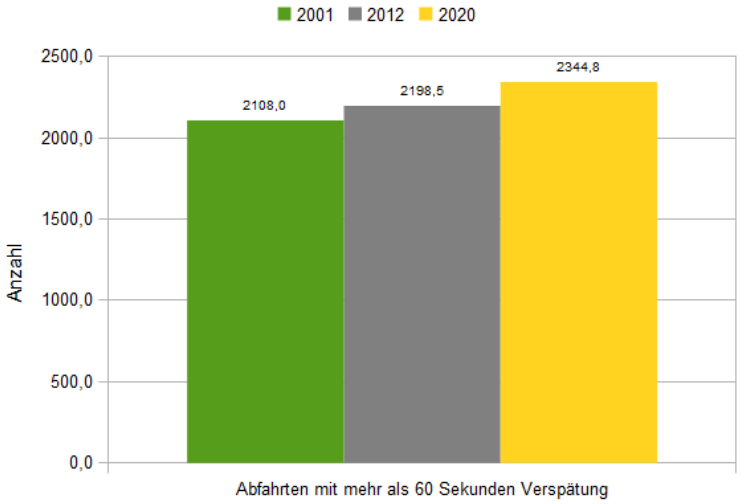


Abbildung 6.72.: Netze 2001, 2012 und 2020: Anzahl größerer Verspätungen

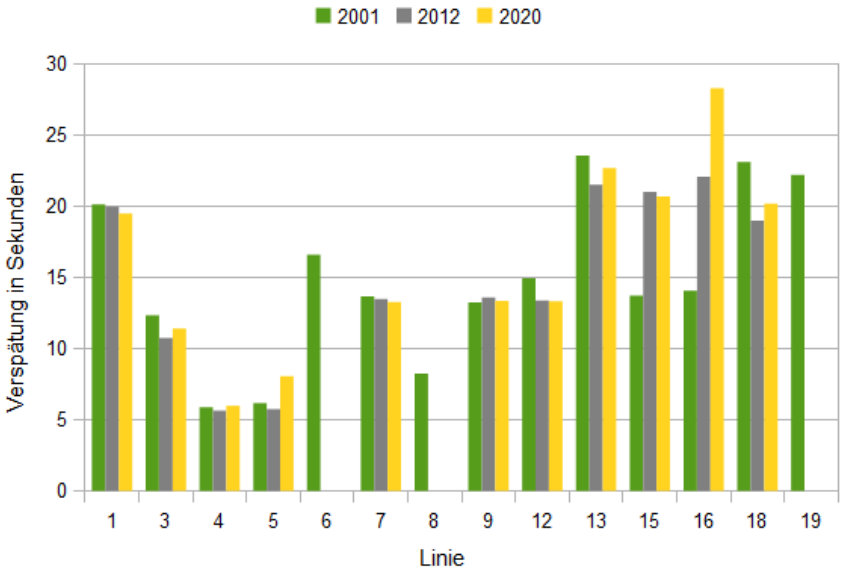


Abbildung 6.73.: Netze 2001, 2012 und 2020: Verspätung der Linien

geblieben ist. Die Verspätungswerte der Linien 3 und 4 ändern sich ebenfalls kaum, trotz der Entlastung des zu durchfahrenden Neumarkttunnels durch den Wegfall der Linie 16 im Netz 2020. Für die Linie 5 werden im Netz 2020 mit durchschnittlich 8,1 Sekunden stärkere Verspätungen als 2012 (5,8 Sekunden) gemessen. Die Linie befährt im Netz von 2020 zusammen mit der Linie 16 den Nord-Süd-Tunnel. Linie 12 ändert den Verlauf von 2001 nach 2012, gewinnt dabei 1,6 Sekunden Pünktlichkeit. Die Linie 13 bleibt im Verlauf unverändert, die gemessene Verspätung ändert sich unter den ausgewählten Optimalfahrplänen nur wenig. Die Linie 15 tauscht zum Netz 2012 den Nordteil ihres Verlaufs mit der Linie 18, die gemessene Verspätung ist 2012 und 2020 deutlich höher als im Netz von 2001. Die Linie 18 gewinnt durch den Tausch etwas an Pünktlichkeit. Bleibt noch Linie 16, die bei allen drei betrachteten Netzen unterschiedliche Verläufe abfährt. Die für sie gemessenen Werte sind daher nicht vergleichbar.

Die beobachtete Durchschnittsverspätung an den Stationen Barbarossaplatz, Ebertplatz und Neumarkt wird in Abbildung 6.74 gezeigt. Auch diese Werte sind natürlich beeinflusst von der Veränderung der sie befahrenden Linien. Eine Aufschlüsselung auf die einzelnen Haltepunkte findet sich in Abbildung 6.75 und Tabelle 6.27.

Linie	2001	2012	2020
	FP 73	FP 154	FP 145
1	20,2	20,0	19,5
3	12,4	10,8	11,4
4	5,9	5,6	6,0
5	6,2	5,8	8,1
6	16,6	-	-
7	13,7	13,5	13,3
8	8,3	-	-
9	13,3	13,6	13,4
12	15,0	13,4	13,3
13	23,6	21,5	22,7
15	13,7	21,0	20,7
16	14,1	22,1	28,3
18	23,2	19,1	20,2
19	22,2	-	-
Durchschnitt	13,8	15,1	16,1

Tabelle 6.26.: Netze 2001, 2012 und 2020: Verspätung der Linien

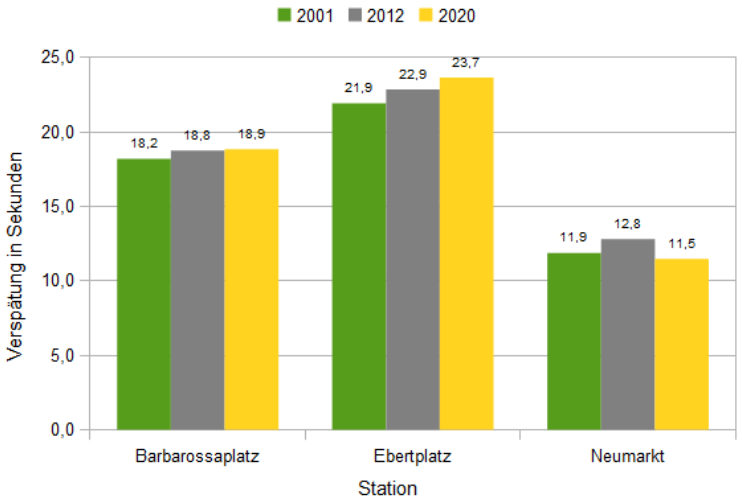


Abbildung 6.74.: Netze 2001, 2012 und 2020: Verspätung an ausgewählten Stationen

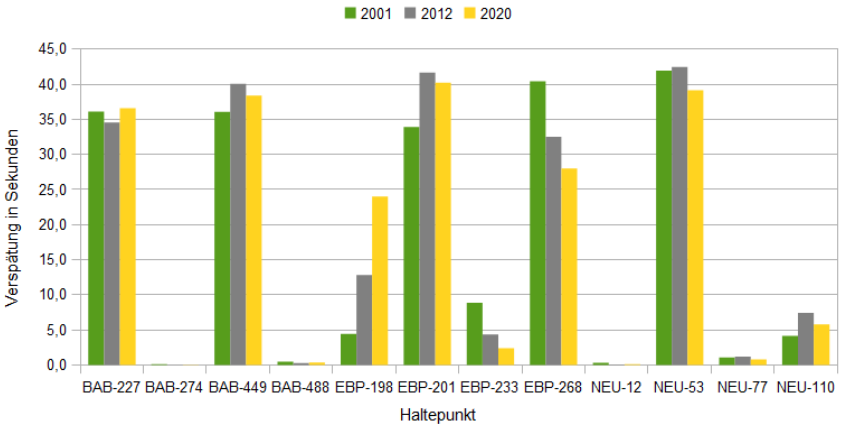


Abbildung 6.75.: Netze 2001, 2012 und 2020: Verspätung an ausgewählten Haltepunkten

Station	Haltepunkt	2001	2012	2020
		FP 73	FP 154	FP 145
Barbarossaplatz	BAB-227	36,1	34,6	36,6
	BAB-274	0,1	0,1	0,1
	BAB-449	36,1	40,1	38,4
	BAB-488	0,5	0,3	0,4
Ebertplatz	EBP-198	4,4	12,8	24,0
	EBP-201	33,9	41,7	40,2
	EBP-233	8,9	4,4	2,4
	EBP-268	40,4	32,5	28,0
Neumarkt	NEU-12	0,3	0,1	0,1
	NEU-53	41,9	42,5	39,1
	NEU-77	1,1	1,2	0,8
	NEU-110	4,2	7,5	5,8
Durchschnitt		17,4	18,1	27,0

Tabelle 6.27.: Netze 2001, 2012 und 2020: Verspätung an ausgewählten Haltepunkten

6.5. Optimierung und Simulation des Stadtbahnnetzes von Montpellier

Die südfranzösische Stadt Montpellier wächst schnell, ihre Bevölkerung hat sich im Laufe der letzten fünfzig Jahre verdreifacht (siehe INSEE in [24]). Das Stadtbahnnetz „Tramway" ist ein zentraler Bestandteil der öffentlichen Infrastruktur, es wird von der *Transports de l'agglomération de Montpellier* (TAM) betrieben. Die erste der zur Zeit der Niederschrift vier Tramway-Linien wurde im Jahr 2000 in Betrieb genommen, sie verbindet die östlichen und westlichen Vorstädte mit der Innenstadt. Mittlerweile werden an jedem Werktag ca. 282.000 Fahrgäste befördert (siehe TAM in [70]), was etwa der Hälfte der Bevölkerung des Großraums Montpellier entspricht. Drei weitere Linien sind in Auftrag gegeben, deren erste den Betrieb im Jahr 2017 aufnehmen soll.

Wie auch in Köln teilen sich im Netz der TAM verschiedene Linien gemeinsame Ressourcen wie Haltepunkte, Weichen und Gleisabschnitte. Daher entstehen auch hier Abhängigkeiten, die dazu führen können, dass kleine, lokale Verspätungen sich auf nachfolgende Fahrzeuge übertragen und so zu größeren Verspätungen addieren. Im Folgenden soll daher untersucht werden, ob ein robuster Fahrplan helfen kann, die Verspätungen im aktuellen Tramway-Netz (siehe Abbildung 6.76) zu reduzieren. Das Netz besteht im Jahr 2013 aus 84 Stationen mit 176 Haltepunkten und 46 Weiche, die durch 232 Gleisabschnitte verbunden sind (siehe TAM in [72]). Diese Gleisabschnitte erreichen bei einer Durchschnittslänge von ca. 240 Metern eine Gesamtlänge von etwa 56 Kilometern. An jedem Betriebstag werden 1.215 Fahrten ausgeführt, die vier Linien mit insgesamt 24 Linienvarianten (siehe Abbildung 6.77) bedienen.

6.5.1. Generieren von Fahrplänen

Der von der TAM implementierte Fahrplan nutzt keinen gemeinsamen Linientakt, die Fahrzeuge bedienen die Routen in über den Betriebstag wechselnden Abständen. In den Hauptverkehrszeiten durchfahren die Linien 1 und 2 das Stadtzentrum alle vier bis fünf Minuten in wechselnden Intervallen. Linie 3 wird alle sechs bis acht Minuten bedient, die Abstände zwischen aufeinander folgenden Fahrten der Linie 4 wechseln zwischen acht und neun Minuten. Um hier eine angemessene Annäherung zu finden, wird ein Taktintervall von acht Minuten angenommen, wobei die zusätzlichen Verstärkerlinien 1A und 2A hinzugefügt werden, um das Taktintervall der Linien 1 und 2 auf vier Minuten zu halbieren. Ein Satz von verkehrsplanerischen Vorgaben wurde definiert (siehe Tabelle 6.28), die die Verstärkerlinien 1A und 2A (Vorgaben 1 bis 4) und minimale Wendezeiten an den Endhaltepunkten (Vorgaben 5 bis 16) vorsehen.

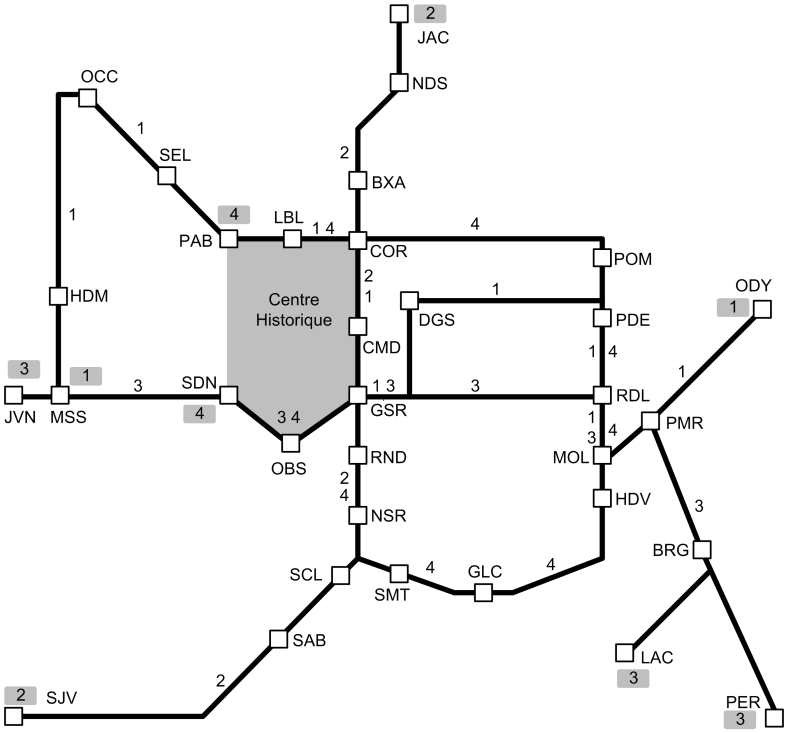


Figure 6.76.: Übersicht zum Stadtbahnnetz von Montpellier

Figure 6.77.: Linienrouten im Tramway-Netz

Nr.	Variante 1	Variante 2	Hp. 1	Hp. 2	Abstandsvorgaben							
					0	1	2	3	4	5	6	7
1	1T01	1T05	9	-	0	0	2	1	1	1	2	0
2	1T02	1T06	115	-	0	0	2	1	1	1	2	0
3	2T01	2T05	35	-	0	0	2	1	1	1	2	0
4	2T02	2T06	136	-	0	0	2	1	1	1	2	0
5	1T01	1T02	30	115	0	0	1	1	1	1	1	1
6	1T02	1T01	173	174	0	0	1	1	1	1	1	1
7	1T05	1T06	30	115	0	0	1	1	1	1	1	1
8	1T06	1T05	94	9	0	0	1	1	1	1	1	1
9	2T01	2T02	55	140	0	0	1	1	1	1	1	1
10	2T02	2T01	116	31	0	0	1	1	1	1	1	1
11	2T05	2T06	51	136	0	0	1	1	1	1	1	1
12	2T06	2T05	120	35	0	0	1	1	1	1	1	1
13	3T01	3T02	78	163	0	0	1	1	1	1	1	1
14	3T02	3T01	151	56	0	0	1	1	1	1	1	1
15	4T01	4T02	152	67	0	0	1	1	1	1	1	1
16	4T02	4T01	100	15	0	0	1	1	1	1	1	1

Table 6.28.: Abstandsvorgaben für das Stadtbahnnetz von Montpellier

Der Genetische Algorithmus wird wieder mit 450 zufällig erzeugten Individuen initialisiert. Der beste Fitnesswert dieser ersten Generation beträgt 75,55 (Durchschnitt: 83,58, schlechtester Wert: 95,00). Nach der Berechnung von 500 Generationen und einer Laufzeit von 313 Sekunden findet der Algorithmus einen besten Lösungskandidaten mit einem Fitnesswert von 75,25 (Durchschnitt: 75,51, schlechtester Wert: 80,11). Der Branch-and-Bound-Solver verbessert diesen Wert im Laufe von 200 Sekunden weiter auf 75,22 und findet 128 optimale Lösungen.

6.5.2. Vergleich der Fahrpläne

Wieder werden jeweils zehn Fahrpläne der Gruppen der initialen und der optimalen Fahrpläne entnommen, und für jeden dieser 20 Fahrpläne zehn Simulationsläufe ausgeführt. Die Maximalgeschwindigkeit wird dabei auf 40 km/h gesetzt, was einen Kompromiss aus der im Stadtzentrum beobachteten Höchstgeschwindigkeit von 30 km/h und den höheren Geschwindigkeiten auf Streckenabschnitten in den Vorstädten darstellt. Die Trödelwahrscheinlichkeit wird wieder auf $p_d = 0,3$ gesetzt, als Trödelfaktor wird wieder $d = 1,3$ gewählt.

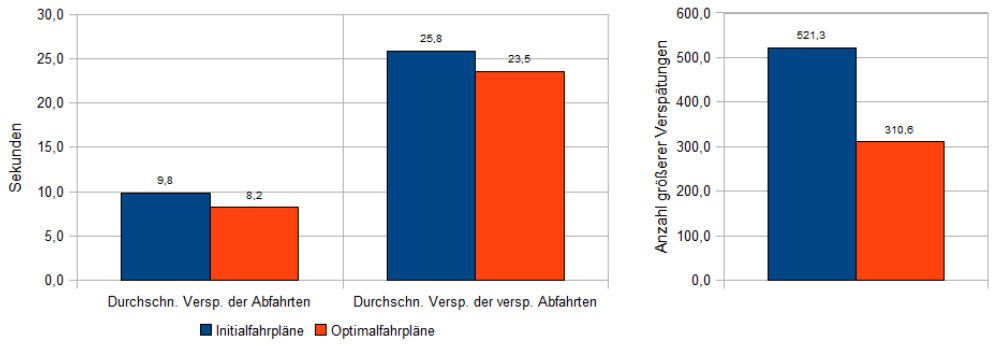


Figure 6.78.: Durchschnittliche und mittlere Verspätung pro Abfahrt (links) und Anzahl der Abfahrten mit größeren Verspätungen (rechts)

Vergleich allgemeiner Merkmale Die Simulationsläufe ergeben unter den initialen Fahrplänen eine durchschnittliche Verspätung aller Abfahrten von 9,8 Sekunden. Unter den optimalen Fahrplänen liegt der Wert bei 8,2 Sekunden, wird also um 16,3% oder 1,6 Sekunden reduziert (siehe Abbildung 6.78, links). Die durchschnittliche Verspätung aller verspäteten Abfahrten wird dabei von 25,8 um 2,3 Sekunden oder 8,9% auf 23,5 Sekunden gesenkt.

Die Häufigkeitsverteilung der auftretenden Verspätungen (siehe Abbildung 6.79) zeigt, dass die Anzahl der Verspätungen jeder Klasse unter den optimalen Fahrplänen geringer ist. Dieser Effekt ist bei Verspätungen über 60 Sekunden besonders signifikant (siehe Abbildung 6.80). Die Gesamtzahl der Abfahrten mit größeren Verspätungen wird von 521,3 um 210,7 Abfahrten oder 40,4% auf 310,6 Abfahrten gesenkt (siehe Abbildung 6.78, rechts).

Robuste Fahrplänen können also auch im Tramway-Stadtbahnnetz dazu beitragen die durchschnittliche Verspätung der Abfahrten zu senken, außerdem reduzieren sie die Anzahl der größeren Verspätungen deutlich. Kleine Verspätungen übertragen sich unter den optimalen Plänen mit ihren besser verteilten Abständen zwischen den Fahrzeugen offenbar weniger häufig auf nachfolgende Bahnen. Größere Verspätungen entstehen kaum, da sich kleinere Verspätungen unter den robusten Fahrplänen seltener aufaddieren können.

Vergleich zweier Fahrpläne Um das Modellverhalten näher zu betrachten wird der Gruppe der Initialfahrpläne eine typischer Fahrplan A (siehe Tabelle 6.29, links) mit einem Fitnessfunktionswert von 92,69 entnommen, der Gruppe der optimalen Lösungskandidaten ein typischer Fahrplan B (siehe Tabelle 6.29, rechts). Für beide Fahrpläne werden je 100 Simulationsläufen mit den bereits beschriebenen Parametern ausgeführt

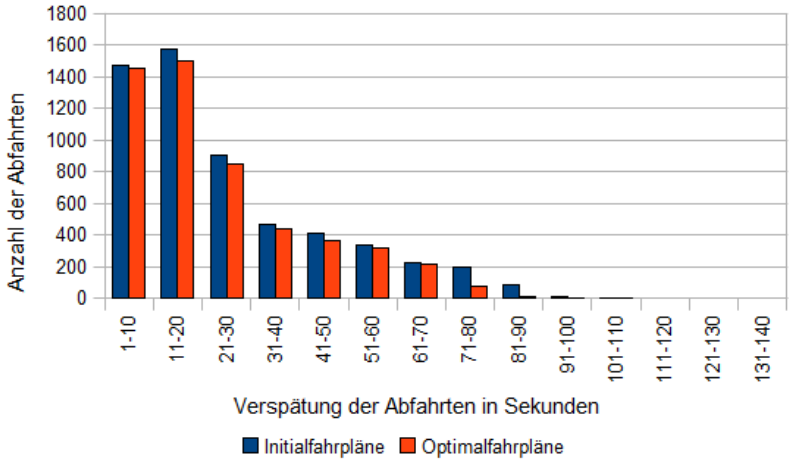


Figure 6.79.: Häufigkeitsverteilung der Verspätungen pro Abfahrt

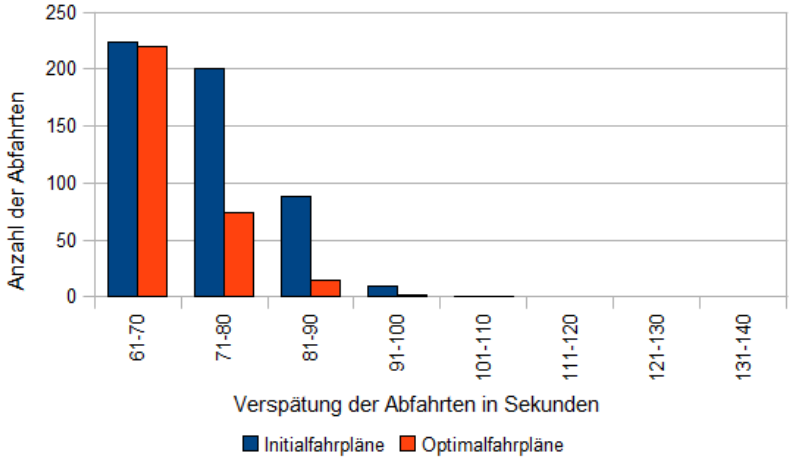


Figure 6.80.: Häufigkeitsverteilung der größeren Verspätungen pro Abfahrt

Richtung	Fahrplan A					
	1	1A	2	2A	3	4
01	0	6	3	1	7	3
02	5	3	3	1	6	2

Richtung	Fahrplan B					
	1	1A	2	2A	3	4
01	6	1	4	0	1	5
02	3	7	7	3	0	5

Tabelle 6.29.: Initialer (links) und optimaler (rechts) Fahrplan

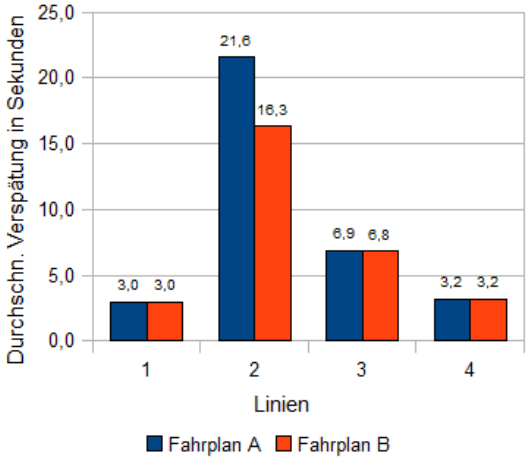


Figure 6.81.: Durchschnittliche Verspätung der Linien

und die Ergebnisse verglichen.

Unter Fahrplan A ergeben sich durchschnittliche Verspätungen von 8,7 Sekunden, die unter Fahrplan B um 17,2% oder 1,5 Sekunden auf 7,3 Sekunden reduziert werden. Linie 2 zeigt jedoch als einzige Linie eine signifikante Senkung der durchschnittlichen Verspätung, sie ist unter dem optimalen Fahrplan mit 16,3 Sekunden um 5,3 Sekunden oder 24,3% niedriger als unter dem initialen Fahrplan mit 21,6 Sekunden (siehe Abbildung 6.81).

Um diese Verbesserung zu untersuchen, werden im Folgenden die Fahrten 3 und 4 des Fahrzeugs 2005 (siehe Abbildung 6.82) untersucht, das auf den Routen 205 und 206 der Linie 2A eingesetzt ist. Während die gemessenen Verspätungen auch an einigen anderen Haltepunkten variieren, werden die deutlichsten Verbesserungen in der Innenstadt um die Stationen Corum (COR, siehe Abbildung 6.76) und Gare Saint-Roch beobachtet.

Im Laufe von Fahrt 3 in Richtung Sabines (siehe Abbildung 6.83) nutzen die Bahnen von Linie 2A nach der Abfahrt von der Station Corum ein Cluster von Weichen, die sie mit den Linien 1, 1A, 2 und 4 teilen. Unter Fahrplan A muss das Fahrzeug hier warten, bis diese Ressourcen zur Verfügung stehen, und kann die daraus resultierende Verspätung bis vor der Station Nouveau Saint-Roch (NSR) nicht aufholen. Unter Fahrplan B mit den

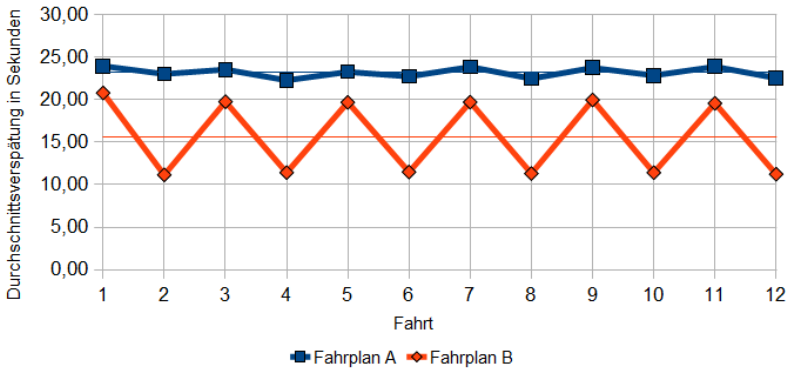


Figure 6.82.: Durchschnittliche Fahrtverspätung des Fahrzeugs 2005 der Linie 2A

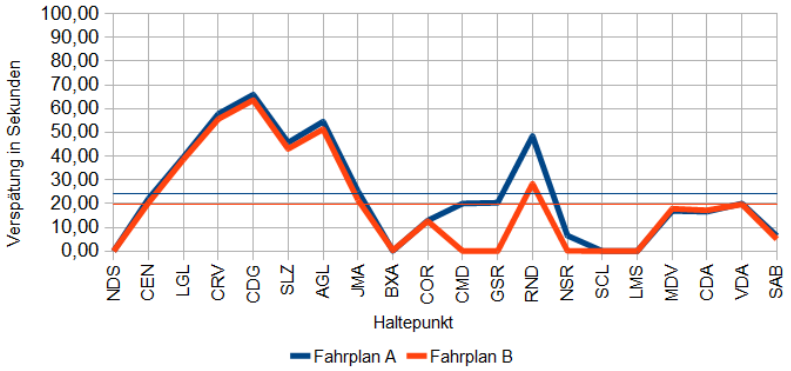


Figure 6.83.: Abfahrtverspätung der Fahrt 3 des Fahrzeugs 2005

besser verteilten Abfahrten stehen diese Ressourcen sofort zur Verfügung, so dass keine Verspätungen zustande kommen.

Auf der Rückfahrt in Richtung Notre-Dame de Sablassou (siehe Abbildung 6.84) muss das Fahrzeug zwischen den Stationen Rondelet (RND) und Gare Saint-Roch vier aufeinander folgende Weichen befahren, die es mit allen anderen Linien teilt. Unter dem zufällig erzeugten Fahrplan A gerät es hier hinter eine Bahn der Linie 1, obwohl es ihr planmäßig um eine Minute vorausfahren sollte. Das Fahrzeug 2005 muss daher mit der Einfahrt in Gare Saint-Roch so lange warten, bis die Bahn der Linie 1 den Haltepunkt frei gemacht hat. Hierdurch entsteht eine Verspätung von etwa 80 Sekunden, die erst wieder aufgeholt werden kann, nachdem die Linienrouten der Linien 1 und 2 sich zwischen den Stationen Comedie und Corum trennen.

Wie beschrieben, zeigt lediglich die Linie 2 (und ihre Verstärkerlinie 2A) eine signifikant geringere Verspätung unter dem robusten Fahrplan B, die Verspätung der anderen Linien

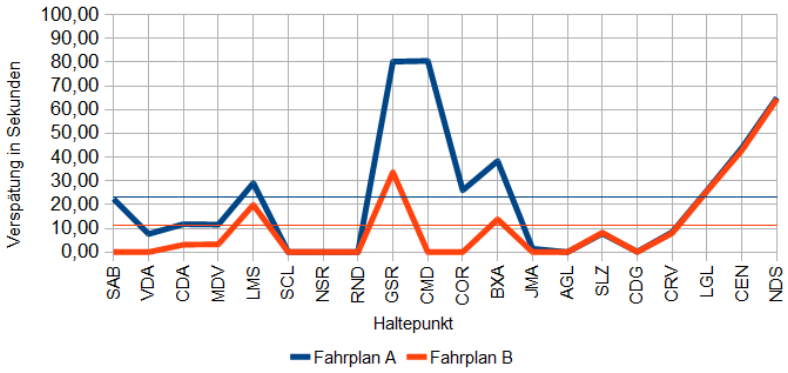


Figure 6.84.: Abfahrtverspätung der Fahrt 4 des Fahrzeugs 2005

ändert sich nicht. Die Linien 1 und 4 verlaufen eine Weile gemeinsam, trennen sich dann und vereinen sich nach Streckenabschnitten mit unterschiedlicher geplanter Fahrzeit wieder (siehe Abbildung 6.76). Da der Optimierer nur valide Fahrpläne generieren kann, bei denen an jedem Haltepunkt zwischen zwei Abfahrten zumindest ein Abstand von einer Minute vorgesehen ist, sind diese Linien unter diesen Bedingungen relativ zueinander verklemmt. Der Optimierer kann bezüglich dieser beiden Linien keine besseren (oder schlechteren) Fahrpläne generieren. Das Gleiche gilt für die Kombination aus den Linien 3 und 4, auch diese Linien sind unter jedem validen Fahrplan verklemmt und bilden so gemeinsam mit der Linie 1 einen Block von Linien, die relativ zueinander nicht frei gelegt werden können.

Diese Phänomen tritt für die Linie 2 nicht auf. Sie läuft zwar parallel zu den Linien 1 und 4, allerdings nur in je einem zusammenhängenden Abschnitt. Die Linien trennen sich, vereinen sich aber nicht wieder nach Abschnitten mit unterschiedlichen Fahrzeiten. Diese Linien sind daher nicht verklemmt, der Optimierer kann Linie 2 freier legen.

Die Experimente zeigen, dass der Einsatz von robusten Fahrplänen auch im Stadtbahnnetz von Montpellier die Verspätungen reduzieren kann. Sie bestätigen weiterhin, dass Robustheit als Gütekriterium in denjenigen Regionen des Netzes ihren Haupteinfluss zeigt, in denen Ressourcen von den meisten Linien gemeinsam genutzt werden (siehe hierzu auch Ullrich, Lückerath und Speckenmeyer in [75]). Im Tramway-Netz sind das die Weichencluster bei den Stationen Gare Saint-Roch und Corum.

6.5.3. Vergleich mit dem real eingesetzten Fahrplan

Um die Betrachtung des Stadtbahnnetzes von Montpellier zu vervollständigen, soll noch der von der TAM im Jahr 2013 eingesetzte Fahrplan untersucht werden (beschrieben von

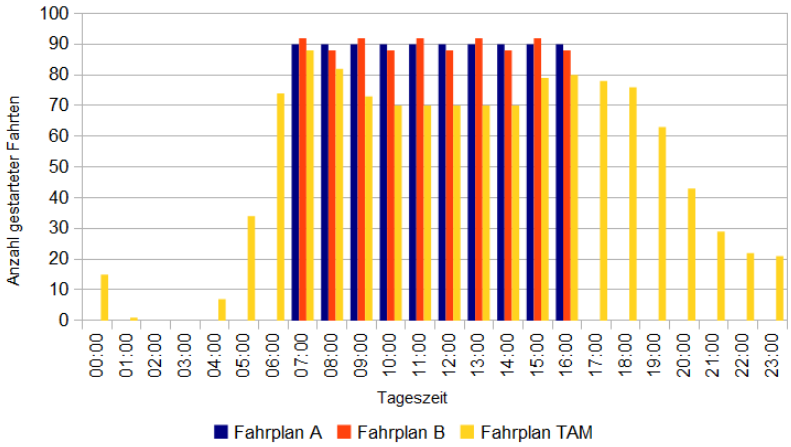


Figure 6.85.: Anzahl der gestarteten Fahrten pro Stunde

TAM in [66], [67], [68] und [69]). Da der Fahrplan nicht mit einem globalen Taktintervall arbeitet und sicherlich eine Reihe von nicht zur Verfügung stehenden verkehrsplanerischen Ansprüchen befolgt, können die Ergebnisse nicht direkt mit den generierten Fahrplänen verglichen werden. Insbesondere sollte nicht angenommen werden, dass aus dem Nebeneinanderstellen der Resultate verkehrsplanerische Schlüsse gezogen werden könnten.

Wie in Abschnitt 6.5.1 beschrieben, wurde für die generierten Fahrpläne als Annäherung an die wechselnden Muster des realen Fahrplans ein gemeinsames Taktintervall von acht Minuten angenommen. Daher weichen die Anzahlen der pro Stunde gestarteten Fahrten in dem Messzeitraum von 08.00 bis 16.59 Uhr (siehe Abbildung 6.85) voneinander ab.

Die durch das Ausführen von 100 Simulationsläufen mit den bereits beschriebenen Parametern gewonnenen Daten liegen für den real verwendeten Fahrplan in einer Größenordnung mit denen der generierten Fahrpläne. Die durchschnittliche Verspätung der Abfahrten liegt bei 8,1 Sekunden und ist damit geringfügig kleiner als die Verspätung unter Fahrplan B und 1,7 Sekunden kleiner als unter Fahrplan A (siehe Abbildung 6.86). Die durchschnittliche Verspätung aller verspäteten Abfahrten beträgt 24,4 Sekunden und liegt damit zwischen den Werten der Fahrpläne A (25,8 Sekunden) und B (23,5 Sekunden). Die Anzahl größerer Verspätungen liegt bei 314,4 und damit auf etwa dem gleichen Niveau des Werts von Fahrplan B.

Die Häufigkeitsverteilung der Verspätungen (siehe Abbildung 6.87) zeigt, dass der von der TAM eingesetzte Fahrplan eine geringere Anzahl von kleinen Verspätungen erzeugt, die aber durch eine höhere Anzahl größerer Verspätungen ausgeglichen werden.

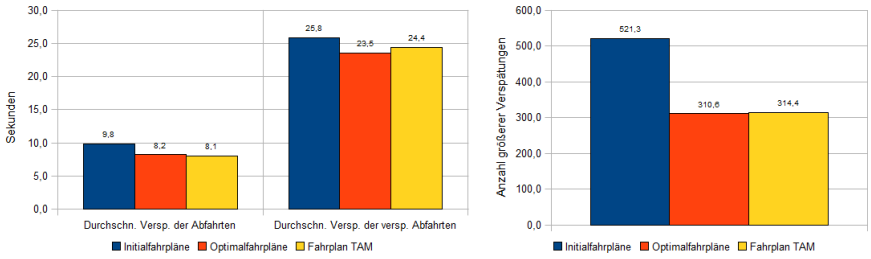


Figure 6.86.: Durchschnittliche und mittlere Verspätung pro Abfahrt (links) und Anzahl der Abfahrten mit größeren Verspätungen (rechts) unter dem real verwendeten Fahrplan

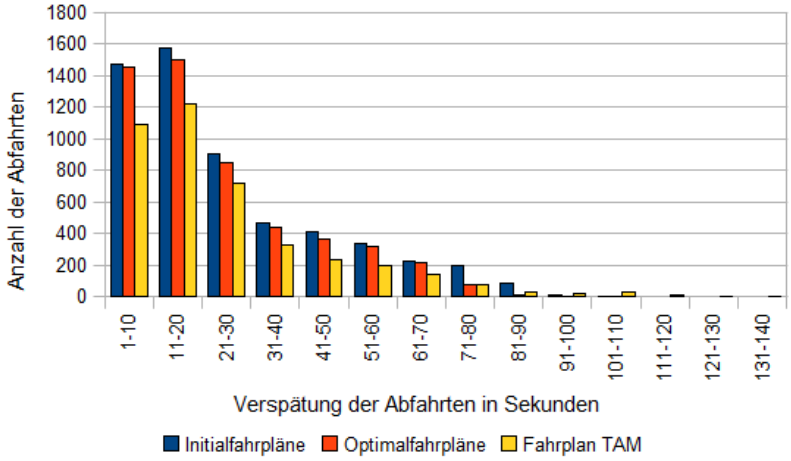


Figure 6.87.: Häufigkeitsverteilung der Verspätungen pro Abfahrt unter dem real verwendeten Fahrplan

Auch bei der Durchschnittsverspätung der einzelnen Linien ist der real eingesetzte Fahrplan vergleichbar mit den generierten Plänen (siehe Abbildung 6.88): Linie 1 hat unter allen drei betrachteten Fahrplänen den gleichen Wert, Linie 2 liegt beim realen Fahrplan zwischen den Werten der Fahrpläne A und B, die Werte der Linien 3 und 4 sind unter dem Fahrplan der TAM etwas höher als ihre Entsprechung bei den generierten Plänen.

6.6. Vergleich der Laufzeiten

Zum Abschluss soll noch das Laufzeitverhalten der Anwendung bei mehreren eingesetzten Prozessoren untersucht werden.

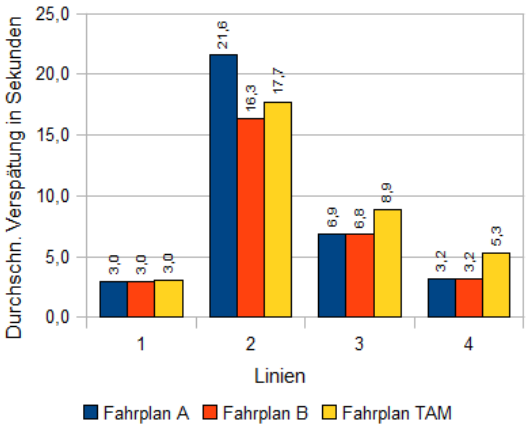


Figure 6.88.: Durchschnittliche Verspätung der Linien unter dem real verwendeten Fahrplan

Betrachtet wird dazu nochmals das Kölner KVB-Netz von 2001 mit den in Abschnitt 6.4.1 beschriebenen Kenngrößen. Der durch die Anwendung aufgespannte Modellgraph enthält dabei 1.170 Knoten und 2.336 Kanten. Bei der initialen Partitionierung des Graphen wird das Kernighan/Lin-Verfahren eingesetzt, das die Anzahl der externen Kanten bei einem typischen Lauf auf zwei Prozessorkernen von 1.180 Kanten auf 35,7% oder 422 Kanten reduziert.

Simuliert wird ein mit der Optimierungsanwendung generierter Fahrplan über 16 Betriebsstunden, die Berechnungen werden auf zwei durch ein 100-MBit-Netzwerk verbundenen Notebooks des in Abschnitt 5.2 beschriebenen Typs durchgeführt. Bei Experimenten mit bis zu vier Prozessorkernen wird ein einzelner Rechner genutzt, ab dem fünften Kern wird das zweite Notebook hinzu geschaltet.

Die Laufzeiten und der Speedup der Anwendung werden in Abbildung 6.89 und Tabelle 6.30 gezeigt. Der Speedup steigt mit eingeschaltetem Lastausgleich auf einen Wert von 2,83 bei vier eingesetzten Prozessoren, die Laufzeit sinkt von 263,8 Sekunden um 63,9% auf 93,1 Sekunden. Zum Vergleich: Bei der Simulation zufällig erzeugter Graphen liegt der Speedup wie in Abschnitt 5.2.1 beschrieben bei vier eingesetzten Prozessoren bei 3,89.

Auffallend ist der Absturz der Leistung auf einen Speedup von 1,89 (resp. 1,83) beim Hinzuschalten eines fünften Prozessorkerns. Die leichte Verbesserung beim Einsatz der verbleibenden Prozessorkerne kann diesen Verlust nicht mehr ausgleichen. Ohne Lastausgleich bleibt der Speedup wie zu erwarten stets unter den Ergebnissen mit Lastausgleich, allerdings wird die Differenz nie größer als 0,20. Dieser Maximalwert tritt beim Einsatz

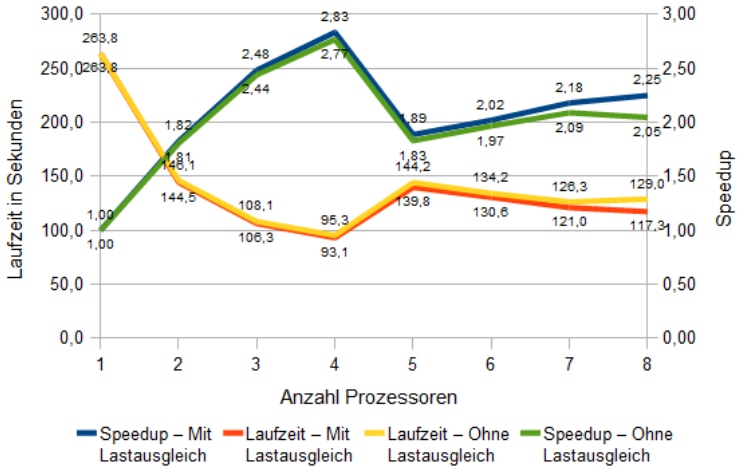


Abbildung 6.89.: Speedup und Laufzeiten mit und ohne Lastausgleich

von acht Kernen auf, wo die Leistung ohne Ausgleich im Vergleich zu sieben Kernen bereits leicht einbricht.

Die Beobachtungen lassen sich wie folgt erklären: Werden mindestens fünf Prozessorkerne (und damit zwei Rechner) eingesetzt, erfolgt ein Teil der Kommunikation über das Netzwerk. Hierdurch entstehen Verzögerungen im Vergleich zur Kommunikation bei der Verwendung von bis zu vier Kernen, die vollständig innerhalb des Caches oder des Hauptspeichers eines gemeinsamen Rechners abläuft. Hinzu kommt, dass die benötigte Rechenzeit für die einzelnen Simulationsschritte (bei einem einzelnen eingesetzten Prozessorkern durchschnittlich 4,6 Millisekunden) sehr kurz ist. Der im Vergleich zur Rechenlast hohe Kommunikationsaufwands sorgt wie in Abschnitt 3.3.4 beschrieben für den Abfall der Gesamtleistung. Bei einem größeren Modell wäre der Anteil des Kommunikationsaufwands an der Gesamtbelastung wesentlich geringer, so dass die Leistungseinbußen durch die Netzwerkkommunikation durch den Vorteil der zusätzlichen Rechenleistung aufgewogen werden. Das Verfahren skaliert wie in Abschnitt 3.3.4 betrachtet für größere Modelle besser.

Die geringe Rechenzeit pro Schritt erklärt auch die relativ geringe Wirkung des Lastausgleichs: Die Synchronisierungszeit wird bei den einzelnen Prozessorkernen nur selten so groß, dass Modellknoten effizient migriert werden können.

Die Anwendung sollte daher auf einem einzelnen Notebook ausgeführt werden, um Verzögerungen durch die Netzwerkkommunikation zu vermeiden. Mit vier eingesetzten Prozessorkernen kann so fast eine Verdreifachung der Ausführungsgeschwindigkeit erreicht werden. Das zu berechnende Simulationsmodell ist also groß genug für die effiziente

Prozessoren	Mit Lastausgleich		Ohne Lastausgleich		Differenz
	Laufzeit	Speedup	Laufzeit	Speedup	Speedup
1	263,8	1,00	263,8	1,00	0,00
2	144,5	1,82	146,1	1,81	0,02
3	106,4	2,48	108,1	2,44	0,04
4	93,1	2,83	95,3	2,77	0,07
5	139,8	1,89	144,2	1,83	0,06
6	130,6	2,02	134,2	1,97	0,05
7	121,0	2,18	126,3	2,09	0,09
8	117,3	2,25	129,0	2,05	0,20

Tabelle 6.30.: Speedup und Laufzeiten (in Sekunden) mit und ohne Lastausgleich

Nutzung des Verfahrens auf einem Parallelrechner, jedoch zu klein für einen effizienten Ablauf in einem LAN.

7. Zusammenfassung und Ausblick

7.1. Zusammenfassung

Nach einer Einführung in Kontext und Thema der Arbeit wurden bestehende parallele Simulationsverfahren betrachtet, dabei wurde insbesondere unterschieden zwischen allgemeinen Verfahren, die für alle diskreten Modelltypen propagiert werden, und anwendungsbezogenen Verfahren, die auf spezielle Untergruppen von Modellen zugeschnitten sind.

Anschließend wurde ein Ansatz zur modellbasierten Parallelisierung von Simulationsanwendungen vorgestellt, der insbesondere Modelle anspricht, die eine Reihe von bei Abbildungen von Stadtbahnnetzen auftretenden Eigenschaften nutzen. Die während der Simulationsläufe auftretende Verlagerung der Last wird durch ein dynamisches und adaptives Lastausgleichsverfahren kompensiert.

Auf Basis des beschriebenen Ansatzes wurde ein Framework für Simulationsanwendungen entworfen und entwickelt. Um dessen dynamisches Verhalten näher zu betrachten, wurden Experimente mit Modellen zufällig erzeugter Graphen durchgeführt. Hierbei zeigte sich, dass das vorgeschlagene Lastausgleichsverfahren die zur Verfügung stehende Rechenleistung effizient ausnutzt. Das Verfahren eignet sich für den Einsatz in inhomogenen Rechnernetzen und auch dann, wenn die verwendeten Rechner nicht exklusiv zur Verfügung stehen.

Auf der Basis des Frameworks wurde das für das CATS-Projekt gewünschte Modul zur parallelen Simulation von Stadtbahnfahrplänen entwickelt. In diesem Kontext wurde ein Optimierungsmodell beschrieben, das taktbasierte Fahrpläne hinsichtlich Robustheit optimiert und gleichzeitig weiter gehende planerische Ansprüche an Fahrpläne berücksichtigt. Als Indikatoren für Robustheit wurden dabei möglichst hohe Abstände zwischen den einzelnen Abfahrten an den Haltepunkten genutzt, die verhindern sollen, dass sich unweigerlich auftretende kleinere Störungen auf Folgefahrzeuge übertragen.

Die beschriebene Software wurde auf das Kölner Stadtbahnnetz aus dem Jahr 2001, das Netz von 2012, das für 2020 geplante Netz inklusive des im Bau befindlichen Nord-Süd-Tunnels und das Stadtbahnnetz von Montpellier angewendet. Das dynamische Verhalten

der erzeugten Fahrpläne wurde dabei mithilfe der Simulationsanwendung geprüft. Dazu wurden zufällig erstellte mit optimalen Fahrplänen verglichen und soweit vorhanden auch real verwendeten Plänen gegenüber gestellt. Die durchschnittliche Pünktlichkeit der Abfahrten verbesserte sich beim Einsatz der optimierten Fahrpläne im Vergleich zu randomisierten Plänen jeweils deutlich. Neben Auswertungen über das gesamte Netz hinweg wurden auch ausgewählte Linien, Haltepunkte und Stationen betrachtet und Abweichungen von den Durchschnittswerten auf bestehende Besonderheiten im untersuchten Netz zurück geführt. Einzelne Simulationsartefakte begründen sich in der schmalen Datengrundlage. Die Verbesserung sowohl der netzweiten Verspätungswerte als auch der bei den meisten Linien und Haltepunkten beobachtete Zugewinn an Pünktlichkeit zeigten, dass Robustheit ein geeignetes Kriterium zur Reduktion von Verspätungen in dichten Fahrplänen ist.

Die in Abschnitt 1.2 gesetzten Ziele konnten im Rahmen dieser Arbeit alle erreicht werden.

7.2. Ausblick

In weiteren Schritten sollte das Überführen der vorhandenen Software in den praktischen Einsatz in Angriff genommen werden. Hierzu müssen die realen planerischen Bedingungen an einen Fahrplan erfasst werden, bislang waren diese Vorgaben nur beispielhaft. Die vorhandene Datengrundlage zu den beiden Stadtbahnnetzen ist allerdings im Wesentlichen ausgeschöpft; für weiter gehende Untersuchungen müssten von Seiten interessierter Verkehrsunternehmen detailliertere Daten zur Verfügung gestellt werden.

Betrachtenswert ist ebenfalls die Erweiterung der Anwendung um ein multimodales Simulationsmodell des Stadtverkehrs, etwa unter Einbeziehung des Busnetzes und von Teilen des Individualverkehrs. Daraufhin ist dann der Übergang zur Onlinesimulation denkbar, also das Weitersimulieren eines realen Betriebstags von einem gegebenen Schnappschuss aus. Mit einer solchen Anwendung könnten nach dem Auftreten einer größeren Störung verschiedene Optionen zur Wiederherstellung des Regelverkehrs geprüft werden, wie der Einsatz von Bussen als Schienenersatzverkehr oder die Umleitung von Bahnen. Hierzu wären mäßige Änderungen an der Simulationsanwendung notwendig, insbesondere müsste eine Echtzeitverbindung zu der von dem betreffenden Verkehrsunternehmen genutzten Datenbank geschaffen werden.

Zudem ist das dargestellte Parallelisierungsverfahren auch abseits der Verkehrssimulation einsetzbar. Gut geeignet ist das Framework z.B. für die Simulation von Kommissioniertouren, wie sie von Werth, Ullrich und Speckenmeyer in [83] beschrieben werden. Hier

lassen sich die Gänge und Regalreihen der Lagenhallen als dünn besetzter Graph modellieren, der von Transienten - in diesem Fall Kommissionierarbeitern - durchwandert wird. Während des Betriebs treffen laufend neue Aufträge ein, zu denen ein Optimierungsmodul passende Kommissioniertouren generiert. Mithilfe einer ausreichend schnellen Simulationsanwendung könnten diese Touren noch in der Planungsphase auf ihre Verträglichkeit zu bereits in der Ausführung befindlichen Touren geprüft werden und so Verzögerungen an bestimmten Engstellen im System vermieden werden.

A. Erweiterung der ÖPNV5-Datenbasis

A.1. Ergänzende Relationen zur Verwaltung mehrerer Netz- und Fahrplaninstanzen

Um in der Datenbank mehrere Netz- und Fahrplaninstanzen gemeinsam verwalten zu können, wird das durch das ÖPNV5-Modell vorgegebene Datenschema etwas erweitert. Die Kompatibilität zu bereits bestehenden Optimierungs- und Simulationsmodulen bleibt dabei bestehen.

Bei den folgenden Relationen bedeutet ein Eintrag in der Spalte *PK*, dass das entsprechende Attribut Bestandteil des Primärschlüssels ist. Ein Eintrag in der Spalte *ÖPNV5* deutet darauf hin, dass sich der Eintrag auf ein im ÖPNV5-Modell vorhandenes Attribut bezieht.

Die Relation CATS_INDEX (siehe Tabelle A.1) enthält für jeden verwalteten Fahrplan einen Eintrag mit einem identifizierenden Wert ID. Der Inhalt des Feldes BASIS_VERSION weist auf das dem Fahrplan zugrunde liegende Netz hin. Weitere Felder enthalten einen erläuternden Text, einen ggf. berechneten Zielfunktionswert des Fahrplans und einige weitere Angaben.

CATS_INDEX: Verwaltung verschiedener Fahrplan- und Umlaufinstanzen

PK	Name	Typ	Erläuterung	ÖPNV5
PK	ID	int	Identifizierender Bezeichner	
	BASIS_VERSION	int	Zugrunde liegende Basisversion	x
	Bezeichnung	str	Erläuternder Text	
	Autor	str	Ersteller der Fahrplan-/Umlaufinstanz	
	InVerwendung	bool	true: Instanz liegt in REC_FRT und REC_UMLAUF vor, false: Instanz liegt nicht vor	
	Ganztag	bool	true: Ganztagsfahrplan, false: Nur Spitzenzeiten	
	ZFW_Franz	real	Durch das Optimierungstool berechneter Zielfunktionswert	

Tabelle A.1.: Relation CATS_INDEX

Die einzelnen Fahrten werden durch die Relation CATS_FRT_POOL (siehe Tabelle A.2) verwaltet und hier über das Feld ID den in CATS_INDEX beschriebenen Fahrplaninstanzen zugewiesen. Es wird beschrieben, welches Fahrzeug (Feld UM_UID) ab welchem Startzeitpunkt (Feld FRT_START) welche Linienvariante (Feld STR_LI_VAR) abzufahren hat. Die Relation enthält alle im ÖPNV5-Modell enthaltenen Daten zu den einzelnen Fahrten, kann aber im Gegensatz zu dessen Relation REC_FRT mehrere Fahrplaninstanzen zugleich verwalten.

CATS_FRT_POOL: Ablage der verschiedenen Fahrplaninstanzen

PK	Name	Typ	Erläuterung	ÖPNV5
PK	ID	int	Bezeichner des Fahrplans	
PK	FRT_FID	int	Bezeichner der Fahrt	x
	FRT_START	int	Startzeit der Fahrt	x
	LI_NR	int	Liniennummer	x
	TAGESART_NR	int	Tagesart	x
	LI_KU_NR	int	Kurznummer der Linie (= Liniennummer)	x
	FAHRTART_NR	int	Fahrtart	x
	FGR_NR	int	Fahrtgruppennummer	x
	STR_LI_VAR	str	Bezeichnung der Linienvariante	x
	UM_UID	int	Bezeichner des ausführenden Umlaufs	x

Tabelle A.2.: Relation CATS_FRT_POOL

Analog dazu enthält die Relation CATS_UMLAUF_POOL (siehe Tabelle A.3) alle im ÖPNV5-Modell abgelegten Daten zu den eingesetzten Umläufen und zusätzlich eine Zuweisung der einzelnen Datensätze zu den in CATS_INDEX definierten Fahrplaninstanzen.

CATS_UMLAUF_POOL: Ablage der verschiedenen Umlaufinstanzen

PK	Name	Typ	Erläuterung	ÖPNV5
PK	ID	int	Bezeichner des Fahrplans	
PK	UM_UID	int	Bezeichner des Umlaufs	x
	ANF_ONR_TYP	int	Typ des Startpunkts	x
	ANF_ORT	int	Indexwert des Startpunkts	x
	END_ONR_TYP	int	Typ des Endpunkts	x
	END_ORT	int	Indexwert des Endpunkts	x

Tabelle A.3.: Relation CATS_UMLAUF_POOL

A.2. Ergänzende Relationen für optionale Daten

Das CATS-Projekt verwendet die schmale Datengrundlage des ÖPNV5-Datenmodells. Um den Genauigkeitsgrad der Simulation zu erhöhen, werden optional einige zusätzliche Daten berücksichtigt. Sind die Daten nicht vorhanden, werden sie wie bereits beschrieben parametrisiert.

Bei den folgenden Relationen bedeutet ein Eintrag in der Spalte *PK*, dass das entsprechende Attribut Bestandteil des Primärschlüssels ist.

In der Relation CATS_OPT_PERSSTAT (siehe Tabelle A.4) werden empirische Daten zu ein- und aussteigenden Passagieren verwaltet, die im Laufe mehrerer Stichprobenfahrten von der KVB ermittelt und für das CATS-Projekt zur Verfügung gestellt wurden.

CATS_OPT_PERSSTAT:
Statistik zu- und aussteigender Passagiere nach Haltepunkt, Linie, und Zeit

PK	Name	Typ	Erläuterung
PK	BASIS_VERSION	int	Basisversion
PK	LI_NR	int	Liniennummer
PK	LI_RICHTUNG	str	ID der Linienrichtung
PK	HP_ID	int	ID des Haltepunkts
	LFD_NR	int	Index des Haltepunkts im Linienverlauf
PK	ZEIT	int	Zeitpunkt des Halts
	PERS_EIN	int	Anzahl der beim Halt zusteigenden Passagiere
	PERS_AUS	int	Anzahl der beim Halt aussteigenden Passagiere
	PERS_BESTAND	int	Anzahl der Passagiere im Fahrzeug (rechnerisch)
	NR_ZAEHLFAHRT	int	Kennung der zugrunde liegenden Stichprobe

Tabelle A.4.: Relation CATS_OPT_PERSSTAT

Für die Haltezeit an den Haltepunkten wird von der Simulation eine modifizierte Dreiecksverteilung angenommen (siehe Abschnitt 6.3.2.4). Um für einzelne Haltepunkte andere Verteilungen vorzusehen, wurde die Relation CATS_OPT_ORT_VERT (siehe Tabelle A.5) angelegt. Hier kann für einzelne Haltepunkte der Typ der Verteilung mitsamt Parametern abgelegt werden.

CATS_OPT_ORT_VERT:
Verteilungen für Haltezeiten an Haltepunkten

PK	Name	Typ	Erläuterung
PK	BASIS_VERSION	int	Basisversion
PK	ONR_TYP_NR	int	Typ des Startorts der Strecke
PK	ORT_NR	int	Index des Startorts der Strecke
	HZT_VERT_TYP	int	Typ der Verteilung der Haltezeiten
	VERT_PARAMETER_A	double	Verteilungsparameter A
	VERT_PARAMETER_B	double	Verteilungsparameter B
	VERT_PARAMETER_C	double	Verteilungsparameter C

Tabelle A.5.: Relation CATS_OPT_ORT_VERT

In der Relation CATS_OPT_LSA_TYP (siehe Tabelle A.6) kann jeder Lichtsignalanlage je nach deren Charakteristika ein bestimmter Typ zugeordnet werden. Die möglichen Werte repräsentieren hier u.a. Ampeln mit Vorangschaltung, Ampeln an Fußgängerüberwegen oder anderen Kreuzungen mit Individualverkehr und Signalen, die den exklusiven Zugang zu Gleisabschnitten regeln. Ist für eine Lichtsignalanlage kein Eintrag vorhanden, wird der globale Standardwert angenommen.

CATS_OPT_LSA_TYP: Typ einzelner Lichtsignalanlagen

PK	Name	Typ	Erläuterung
PK	BASIS_VERSION	int	Basisversion
PK	ONR_TYP_NR	int	Typ des Startorts der Strecke
PK	ORT_NR	int	Index des Startorts der Strecke
PK	SEL_ZIEL	int	Typ des Zielorts der Strecke
PK	SEL_ZIEL_TYP	int	Index des Zielorts der Strecke
PK	SEL_LSA_LAENGE	int	Offset der LSA auf der Strecke
	LSA_TYP	int	Typ der LSA

Tabelle A.6.: Relation CATS_OPT_LSA_TYP

In Stadtbahnnetzen kann es weichenlose Kreuzungen von Gleisen geben. Hier sind in der Regel Lichtsignalanlagen aufgestellt, die den Zugang regeln. Diese einzelnen Lichtsignale werden in der Relation CATS_OPT_LSA_GRUPPE (siehe Tabelle A.7) zu Gruppen zusammen gefasst, die dann von der Simulationsanwendung so koordiniert werden, dass die konfliktfreie Nutzung der sich kreuzenden Gleise möglich ist.

CATS_OPT_LSA_GRUPPE:
Gruppenzugehörigkeit einzelner Lichtsignalanlagen

PK	Name	Typ	Erläuterung
PK	BASIS_VERSION	int	Basisversion
PK	ONR_TYP_NR	int	Typ des Startorts der Strecke
PK	ORT_NR	int	Index des Startorts der Strecke
PK	SEL_ZIEL	int	Typ des Zielorts der Strecke
PK	SEL_ZIEL_TYP	int	Index des Zielorts der Strecke
PK	SEL_LSA_LAENGE	int	Offset der LSA auf der Strecke
	LSA_GRUPPE	int	Gruppe der LSA
	LSA_UNTERGRUPPE	int	Untergruppe der LSA

Tabelle A.7.: Relation CATS_OPT_LSA_GRUPPE

In der Relation CATS_OPT_SEL_GESCHWINDIGKEIT (siehe Tabelle A.8) können für einzelne Streckenabschnitte Höchstgeschwindigkeiten. Ist für einen Abschnitt kein Wert hinterlegt, wird der globale Standardwert angenommen.

CATS_OPT_SEL_GESCHWINDIGKEIT:
Höchstgeschwindigkeiten für Strecken

PK	Name	Typ	Erläuterung
PK	BASIS_VERSION	int	Basisversion
PK	ONR_TYP_NR	int	Typ des Startorts der Strecke
PK	ORT_NR	int	Index des Startorts der Strecke
PK	SEL_ZIEL	int	Typ des Zielorts der Strecke
PK	SEL_ZIEL_TYP	int	Index des Zielorts der Strecke
	SEL_MAX_GESCHW	int	Höchstgeschwindigkeit auf der Strecke

Tabelle A.8.: Relation CATS_OPT_SEL_GESCHWINDIGKEIT

Eingleisige Strecken werden in beiden Richtungen von Bahnen befahren. Da sie im ÖPNV5-Modell nicht direkt abgebildet werden können, werden in der Relation CATS_OPT_SEL_BIDIREKTIONAL (siehe Tabelle A.9) zwei gerichtete Streckenabschnitte ST1 und ST2 markiert, die zusammen eine eingleisige Strecke abbilden. Sobald eine Bahn eine der Strecken befährt, wird die andere für entgegenkommenden Verkehr gesperrt.

CATS_OPT_SEL_BIDIREKTIONAL:
Bidirektionale Streckenabschnitte

PK	Name	Typ	Erläuterung
PK	BASIS_VERSION	int	Basisversion
PK	ST1_ONR_TYP_NR	int	Erste Strecke: Typ des Startorts
PK	ST1_ORT_NR	int	Erste Strecke: Index des Startorts
PK	ST1_SEL_ZIEL	int	Erste Strecke: Typ des Zielorts
PK	ST1_SEL_ZIEL_TYP	int	Erste Strecke: Index des Zielorts
PK	ST2_ONR_TYP_NR	int	Zweite Strecke: Typ des Startorts
PK	ST2_ORT_NR	int	Zweite Strecke: Index des Startorts
PK	ST2_SEL_ZIEL	int	Zweite Strecke: Typ des Zielorts
PK	ST2_SEL_ZIEL_TYP	int	Zweite Strecke: Index des Zielorts

Tabelle A.9.: Relation CATS_OPT_SEL_BIDIREKTIONAL

In der Relation CATS_OPT_WEICHE_GESCHWINDIGKEIT (siehe Tabelle A.10) können für einzelne Weichen maximal zulässige Überfahrtgeschwindigkeiten hinterlegt werden. Ist für eine Weiche kein Wert hinterlegt, wird der globale Standardwert angenommen.

CATS_OPT_WEICHE_GESCHWINDIGKEIT: Höchstgeschwindigkeit auf einzelnen Weichen

PK	Name	Typ	Erläuterung
PK	BASIS_VERSION	int	Basisversion
PK	ONR_TYP_NR	int	Typ des Orts
PK	ORT_NR	int	Index des Orts
	SEL_MAX_GESCHW	int	Zulässige Höchstgeschwindigkeit

Tabelle A.10.: Relation CATS_OPT_WEICHE_GESCHWINDIGKEIT

Einige Linien können nur von einigen Fahrzeugtypen bedient werden, da z.B. nur Niederflurbahnen in Frage kommen. Mit der Relation CATS_OPT_LINIE_FZG_TYP (siehe Tabelle A.11) können Zuordnungen von Fahrzeugtypen zu Linien verwaltet werden. Die Beschreibungen der Fahrzeugtypen werden in der zu ÖPNV5 gehörenden Relation MENGE_FZG_TYP verwaltet.

CATS_OPT_LINIE_FZG_TYP: Zulässige Fahrzeugtypen je Linie

PK	Name	Typ	Erläuterung
PK	BASIS_VERSION	int	Basisversion
PK	LI_NR	int	Liniennummer
	FZG_TYP	int	Zulässiger Fahrzeugtyp

Tabelle A.11.: Relation CATS_OPT_LINIE_FZG_TYP

In der Relation CATS_OPT_UMLAUF_FZG_TYP (siehe Tabelle A.12) können einzelnen Umläufen von der Standardeinstellung der Simulation abweichende Fahrzeugtypen zugewiesen werden. Auch hierzu wird die ÖPNV5-Relation MENGE_FZG_TYP benötigt.

CATS_OPT_UMLAUF_FZG_TYP: Fahrzeugtyp der Umläufe

PK	Name	Typ	Erläuterung
PK	BASIS_VERSION	int	Basisversion
PK	UM_UID	int	Umlaufnummer
	FZG_TYP	int	Fahrzeugtyp

Tabelle A.12.: Relation CATS_OPT_UMLAUF_FZG_TYP

B. Literaturverzeichnis

[1] Amdahl, G.: Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In: AFIPS Conference Proceedings. 30, 1967, pp. 483-485.

[2] Avril, H., Tropper, C.: Clustered Time Warp and Logic Simulation. Proceedings of the Ninth Workshop on Parallel and Distributed Computing, 1995, pp. 112-119.

[3] Avril, H., Tropper, C.: The Dynamic Load Balancing of Clustered Time Warp for Logic Simulation. In: Proceedings of the tenth workshop on Parallel and distributed simulation, pp. 20-27, 1996.

[4] Bampas, E., Kaouri, G., Lampis, M., and Pagourtzis, A.: Periodic Metro Scheduling. In: Proc. ATMOS. 2006.

[5] Banks, J., Carson, J.S., Nelson B.L., Nicol D.M.: Discrete-Event System Simulation, Upper Saddle River: Pearson, 2010.

[6] Bryant, R.E.: Simulation of Packet Communication Architecture Computer Systems. Computer Science Laboratory. Technical Report, Cambridge, Massachusetts, Massachusetts Institute of Technology, 1977.

[7] Cacchiana, V., Caprara, A., Fischetti, M.: A Lagrangian Heuristic for Robustness, with an Application to Train Timetabling. Transportation Science, Vol. 46, No. 4, 2012, pp. 124-133.

[8] Caimi, G., Fuchsberger, M., Laumanns, M. and Schüpbach, K.: Periodic Railway Timetabling with Event Flexibility. In: Networks. 2010, Vol. 57, Number 1, pp. 3-18.

[9] Chandy, K. M., Misra, J.: Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. In: IEEE Transactions on Software Engineering, SE-5(5), pp. 440-452.

[10] Chandy, K. M., Misra, J.: Asynchronous distributed simulation via a sequence of parallel computations. In: Communications of the ACM, Vol. 24, No. 4, 1981, pp. 198-205.

[11] Dakin, R. J.: A tree-search algorithm for mixed integer programming problems. In: The Computer Journal, Volume 8, 1965, pp. 250–255.

[12] Deelman, E., Szymanski, B. K.: Dynamic Load Balancing in Parallel Discrete Event Simulation for Spatially Explicit Problems. In: Proceedings of the Twelfth Workshop on Parallel and Distributed Simulation (PADS), 1998, pp. 46-53.

[13] Deelman, E., Szymanski, B. E., Caraco, T.: Simulating Lyme disease using parallel discrete event simulation. In: Proceedings of the 28th Winter Simulation Conference, 1996, pp. 1191-1198.

[14] Dréo, J., Pétrowski, A., Siarry, P. and Taillard, E.: Metaheuristics for Hard Optimization. Springer, 2006.

[15] Franz, S.: Entwurf und Entwicklung eines mehrstufigen Optimierungsverfahrens für Stadtbahnfahrpläne unter Berücksichtigung verkehrsplanerischer Vorgaben. Diplomarbeit, Universität zu Köln, 2011.

[16] Fujimoto, R. M.: Parallel Discrete Event Simulation. In: Proceedings of the 1989 Winter Simulation Conference, 1989, pp. 19-28.

[17] Fujimoto, R. M.: Parallel and Distributed Simulation. New York: John Wiley & Sons, 2000.

[18] Fujimoto, R. M.: Parallel and Distributed Simulation Systems. In: Proceedings of the 2001 Winter Simulation Conference, 2001.

[19] Garey, M., Johnson, D., Stockmeyer, L.: Some simplified NP-complete graph problems. In: Theoretical Computer Science, 1976, pp. 237-267.

[20] Genç, Z.: Ein Neuer Ansatz zur Fahrplanoptimierung im ÖPNV: Maximierung von Zeitlichen Sicherheitsabständen. Dissertation. Mathematisch-Naturwissenschaftliche Fakultät, Universität zu Köln. 2003.

[21] Greenberg, A. G., et al.: Algorithms for Unboundedly Parallel Simulations. In: ACM Transactions on Computer Systems, Vol. 9, No. 3, 1991, pp. 201-221.

[22] Gropp, W., Lusk, E., Skjellum, A.: MPI - Eine Einführung. München: Oldenbourg Verlag, 2007.

[23] Heidelberger P., Stone, H.: Parallel Trace-Driven Cache Simulation by Time Partitioning. In: Proceedings of the 1990 Winter Simulation Conference, 1990, pp. 734-737.

[24] Institut national de la statistique et des études économiques: La population de Montpellier Agglomération a triplé au cours des cinquante dernières années. http://www.insee.fr/fr/themes/document.asp?ref_id=16088, accessed on June, 26th, 2013.

[25] Intel Corp.: Intel Turbo Boost Technology in Intel Core Microarchitecture (Nehalem) Based Processors. White Paper, Intel Corp., 2008.

[26] Intel Corp.: Mobile Intel® Atom™ Processor N270 Single Core. Datasheet, Intel Corp., 2008.

[27] Intel Corp.: Intel Core i7-900 Mobile Processor Extreme Edition Series, Intel Core i7-800 and i7-700 Mobile Processor Series. Datasheet, Intel Corp., 2009.

[28] Jefferson, D. R.: Virtual Time. In: ACM Transactions on Programming Languages and Systems, Vol. 7, No. 3, 1985, pp. 404-425.

[29] Joisten, M.: Simulation von Fahrplänen für den ÖPNV mittels Zellularautomaten. Diplomarbeit, Univ. Köln, 2002.

[30] Kernighan, B. W., Lin, S.: An Efficient Heuristic Procedure for Partitioning Graphs. Bell Syst. Tech Journal, Volume 49, Number 2, 1970, pp. 291-307.

[31] Kölner Verkehrsbetriebe AG (Hrsg.): Nord-Süd Stadtbahn Köln - Tunnelbau in der City.

[32] Kölner Verkehrsbetriebe AG (Hrsg.): Nord-Süd Stadtbahn Köln - Die zweite Baustufe.

[33] Kölner Verkehrsbetriebe AG, Deutsche Bahn AG, VRS GmbH (Hrsg.): Bahnen in Köln 2012. Netzplan, 2012.

[34] Knuth, D.: Estimating the efficiency of backtrack programs. Mathematics of Computation 29 (129), 1975, pp. 121–136.

[35] Liebl, F.: Simulation. München: R. Oldenbourg Verlag, 1995.

[36] Lubachevsky, B., Schwartz, A., Weiss, A.: Rollback Sometimes Works ... If Filtered. Proceedings of 1989 Winter Simulation Conference, 1989, pp. 630-639.

[37] Lückemeyer, G.: Datenmodellierung und Datenschnittstellen - Basis einer effizienten Fahrplangestaltung für den ÖPNV. Diplomarbeit. Universität zu Köln, 2000.

[38] Lückemeyer, G.: A Traffic Simulation System Increasing the Efficiency of Schedule Design for Public Transport Systems Based on Scarce Data. Dissertation. Aachen: Shaker Verlag, 2007.

[39] Lückemeyer, G., Speckenmeyer, E.: Comparing Applicability of Two Simulation Models in Public Transport Simulation. In: Becker, M., Szczerbicka, H. (Ed.): ASIM 2006 – 19. Symposium Simulationstechnik. ASIM/Universität Hannover, 2006.

[40] Lückerath, D.: Entwurf und Entwicklung einer Anwendung zur parallelen Simulation von schienengebundenem Öffentlichen Personennahverkehr. Diplomarbeit, Universität zu Köln, 2011.

[41] Lückerath, D., Ullrich, O., Speckenmeyer, E.: Modeling time table based tram traffic. In: Simulation Notes Europe (SNE), ARGESIM/ASIM Pub., TU Vienna, Volume 22, Number 2, August 2012, pp. 61-68.

[42] Lückerath, D., Ullrich, O., Speckenmeyer, E.: Applicability of re-scheduling strategies in tram networks. In: Proceedings of ASIM-Workshop STS/GMMS 2013, ARGESIM/ASIM Pub., TU Vienna, 2013.

[43] Marr, D. T., et al.: Hyper-Threading Technology Architecture and Microarchitecture. In: Intel Technology Journal, Volume 6, Issue 1, 2002, pp. 4-15.

[44] Mattern, F.: Efficient algorithms for distributed snapshots and global virtual time approximation. Journal of Parallel and Distributed Computing, Vol. 18, No. 4, 1993, pp. 423-434.

[45] Meisgen, F.: Dynamic Load Balancing for Simulations of Biological Aging. In: International Journal of Modern Physics C, Vol. 8, Issue 3, June 1997, pp. 575-582.

[46] Meisgen, F.: Dynamische Lastausgleichsverfahren in heterogenen Netzwerken. Aachen: Shaker Verlag, 1998.

[47] Meisgen, F., Speckenmeyer, E.: A dynamic load balancing algorithm with adjustable degree of imbalance on heterogeneous networks. In: Proc. of ARCS 2002, Ed.: Brinkschulte, E., Großpietsch, K.-E., Hochberger, C., Mayr, E.W., VDE-Verlag, Karlsruhe, April 08-12, 2002, pp. 161-168.

[48] Merz, M., Bröcker, E.: Einsatz von Open Source Frameworks zur Parallelisierung von Dymola Simulationen. In: Proc. ASIM/GI Workshop STS/GMMS, Ed.: W. Commerell, ISBN 978-3-9810998-3-6, Ulm, März 04-05, 2010, S. 283-289.

[49] Middelkoop, D., Bouwman, M.: Simone - Large Scale Train Network Simulations. In: Proc. Winter Simulation Conference, 2001, pp. 1042-1145.

[50] Oberbürgermeister der Stadt Köln: Stadtentwicklung in Köln – Mobilitätsentwicklung in Köln bis 2025. Stadt Köln, 2008.

[51] Nagel, K., Schreckenberg, M.: A cellular automaton model for freeway traffic. In: Journal de Physique I, Volume 2, Issue 12, December 1992, pp. 2221-2229.

[52] Nash, A., Huerlimann, D.: Railroad Simulation Using OpenTrack. In: Allan, J., R.J. Hill, C.A. Brebbia, G. Sciutto, S. Sone (Ed.): Computers in Railways IX, WIT Press, Southampton, 2004, pp. 45-54.

[53] Nicol, D. M.: The cost of conservative synchronization in parallel discrete event simulations. Journal of the Association of Computing Machinery, Vol. 40, No. 2, 1993, pp. 304-333.

[54] Rönngren, R., Ayani, R.: A Comparative Study of Parallel and Sequential Priority Queue Algorithms. In: ACM Transactions on Modeling and Computer Simulation, Vol. 7, No. 2, 1997, pp. 157-209.

[55] Samadi, B.: Distributed simulation, algorithms and performance analysis. Computer Science Department, PhD Thesis, University of Los Angeles, 1985.

[56] Sargent, R. G.: Verification and validation of simulations models. In: Proc. Winter Simulation Conference, 2010. pp. 166-183.

[57] Schlagenhaft, R.: Dynamischer Lastausgleich optimistisch synchronisierter, verteilter Simulation. In: Proc. ASIM-Workshop VSPP, 1999.

[58] Schlagenhaft, R., Ruhwandel, M., Sporrer, C., Bauer, H.: Dynamic Load Balancing of a Multi-Cluster Simulator on a Network of Workstations. In: Proc. PADS95, 1995, pp. 175-180.

[59] Schöbel, A.: A Model for the Delay Management Problem based on Mixed-Integer-Programming. In: Proceedings of ATMOS. 2001.

[60] Schöneburg, E., Heinzmann, F., Feddersen, S.: Genetische Algorithmen und Evolutionsstrategien. Bonn: Addison-Wesley, 1994.

[61] Sedgewick, R.: Algorithms in C++, Vol. 1. Boston: Addison-Wesley, 1998.

[62] Speckenmeyer, E., Li, N., Lückerath, D., Ullrich, O.: Socio-Economic Objectives in Tram Scheduling. Technical Report, Universität zu Köln, 2012.

[63] Sporrer, Ch., Bauer, H.: Corolla partitioning for distributed logic simulation of VLSI-circuits. In: Proceedings of the Workshop on Parallel and Distributed Simulation (PADS), 1993, pp. 85-92.

[64] Steinman, J.: SPEEDES: Synchronous parallel environment for emulation and discrete event simulation. Advances in Parallel and Distributed Simulation, SCS Simulation Series, Vol. 23, 1991, pp. 95-103.

[65] Suhl, L. and Mellouli, T.: Managing and preventing delays in railway traffic by simulation and optimization. In: Mathematical Methods on Optimization in Transportation Systems (2001), pp. 3–16.

[66] Transports de l'agglomération de Montpellier (Ed.: Subra, R.): Horaires tram 1. 2012.

[67] Transports de l'agglomération de Montpellier (Ed.: Subra, R.): Horaires tram 2. 2012.

[68] Transports de l'agglomération de Montpellier (Ed.: Subra, R.): Horaires tram 3. 2012.

[69] Transports de l'agglomération de Montpellier (Ed.: Subra, R.): Horaires tram 4. 2012.

[70] Transports de l'agglomération de Montpellier: Un réseau en étoile. http://www.montpellier-agglo.com/tam/page.php?id_rubrique=31, accessed on Feb, 21st, 2013.

[71] Transports de l'agglomération de Montpellier: Le tracé du ligne 5. http://www.ligne5-montpellier-agglo.com/?page_id=16, accessed on Feb, 21st, 2013.

[72] Transports de l'agglomération de Montpellier (Ed.: Subra, R.): Plan du réseau du centre de l'agglomération. 2012.

[73] Ullrich, O.: CATSview - Eine Benutzerschnittstelle zur Steuerung, Visualisierung und Ergebnisauswertung von Fahrplansimulationen im ÖPNV. Diplomarbeit, Univ. Köln, 2006.

[74] Ullrich, O., Lückerath, D., Franz, S., Speckenmeyer, E.: Simulation and optimization of Cologne's tram schedule. In: Simulation Notes Europe (SNE), ARGESIM/ASIM Pub., TU Vienna, Volume 22, Number 2, August 2012, pp. 69-76.

[75] Ullrich, O.; Lückerath, D.; Speckenmeyer, E.: Reduzieren robuste Fahrpläne Verspätungen in Stadtbahnnetzen? - Es kommt drauf an! Submitted to HEUREKA 2014, 19 pg.

[76] Ullrich, O.; Lückerath, D.; Speckenmeyer, E.: A robust schedule for Montpellier's Tramway network. Technical Report, Univ. Köln, 2013, 17 pg.

[77] Ullrich, O.; Lückerath, D.; Speckenmeyer, E.: Model-based parallelization of traffic simulation systems. Technical Report, Univ. Köln, 2013, 17 pg.

[78] Ullrich, O., Proff, I., Lückerath, D., Kuckertz, D., Speckenmeyer, E.: Agent-based modeling and simulation of individual traffic as an environment for bus schedule simulation. In: Proceedings of TUM Mobil 2013, München, to appear.

[79] Verband Deutscher Verkehrsunternehmen eV: VDV-Standardschnittstelle Liniennetz/Fahrplan. VDV-Schriften 452, 2008.

[80] Vossloh Kiepe GmbH: Elektrische Ausrüstung des Niederflur-Stadtbahnwagens K4000 der Kölner Verkehrs-Betriebe AG. Druckschrift 00KV7DE, 2003.

[81] Walter, D.: Optimierung von ÖPNV-Fahrplänen unter Berücksichtigung verkehrsplanerischer Vorgaben. Diplomarbeit, Univ. Köln, 2010.

[82] Watts, J., Taylor, S.: A Practical Approach to Dynamic Load Balancing. In: IEEE Transactions on Parallel and Distributed Systems, Vol. 9, No. 3, 1998, pp. 235-248.

[83] Werth, F.; Ullrich, O.; Speckenmeyer, E.: Reducing blocking effects in multi-block layouts. In: Bödi, R.; Maurer, W. (Ed): Proceedings of ASIM 2011, Winterthur, 2011, 10 pg.

C. Software

Die zu dieser Dissertation gehörende Software findet sich auf den Internet-Seiten des Instituts für Informatik der Universität zu Köln unter der Adresse

http://scale.uni-koeln.de/index.php?id=16243

Versicherung

Ich versichere, dass ich die von mir vorgelegte Dissertation selbständig angefertigt, die benutzten Quellen und Hilfsmittel vollständig angegeben und die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken im Wortlaut oder dem Sinn nach entnommen sind, in jedem Einzelfall als Entlehnung kenntlich gemacht habe; dass diese Dissertation noch keiner anderen Fakultät oder Universität zur Prüfung vorgelegen hat; dass sie abgesehen von den in unten angegebenen Teilpublikationen noch nicht veröffentlicht worden ist, sowie dass ich eine solche Veröffentlichung vor Abschluss des Promotionsverfahrens nicht vornehmen werde. Die Bestimmungen dieser Promotionsordnung sind mir bekannt. Die von mir vorgelegte Dissertation ist von Prof. Dr. Ewald Speckenmeyer betreut worden.

Köln, am 14. Oktober 2013,

(Oliver Ullrich)

- Ullrich, O., Lückerath, D., Speckenmeyer, E.: Reduzieren robuste Fahrpläne Verspätungen in Stadtbahnnetzen? - Es kommt drauf an! Submitted to HEUREKA 2014, 19 pg. (Anteil des Autors: 70%)
- Ullrich, O., Lückerath, D., Speckenmeyer, E.: A robust tram schedule for Montpellier's Tramway network. Technical Report, Univ. Köln, 2013, 17 pg. (Anteil des Autors: 80%)
- Ullrich, O., Lückerath, D., Franz, S., Speckenmeyer, E.: Simulation and optimization of Cologne's tram schedule. In: Simulation Notes Europe (SNE), ARGESIM/ASIM Pub., TU Vienna, Volume 22, Number 2, August 2012, pp. 69-76. (Anteil des Autors: 80%)
- Lückerath, D., Ullrich, O., Speckenmeyer, E.: Modeling time table based tram traffic. In: Simulation Notes Europe (SNE), ARGESIM/ASIM Pub., TU Vienna, Volume 22, Number 2, August 2012, pp. 61-68. (Anteil des Autors: 40%)

Lebenslauf

25.03.1975	Geboren in Duisburg
05/94	Abitur am Konrad-Adenauer-Gymnasium in Langenfeld (Rheinland)
10/94 - 11/95	Zivildienst
10/95 - 09/06	Studium der Wirtschaftsinformatik an der Universität zu Köln, Abschluss mit Diplom
10/06 - 12/12	Wissenschaftlicher Mitarbeiter des Instituts für Informatik der Universität zu Köln
10/13	Abschluss der Dissertation