Endbericht PG 502

Stauvermeidung durch on-line Verkehrsplanung *STOP*

Autoren:
Zeynep Bay, Sven Becker, Sven Böttcher, Christian Brunner, Armin Büscher, Thomas Fürst, Anca Manuela Lazarescu, Elisei Rotaru, Sebastian Senge, Bastian Steinbach, Funda Yilmaz, Thomas Zimmermann

Projektgruppe
am Fachbereich Informatik
der Universität Dortmund

WS 2006/2007, SS 2007

Betreuer:
Prof. Dr. Horst F. Wedde
Dipl.-Inform. Sebastian Lehnhoff
Dr. Bernhard van Bonn

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

In modernen Industrie- und Dienstleistungsgesellschaften ist der Fahrzeugverkehr von immenser wirtschaftlicher und gesellschaftlicher Bedeutung. Verkehrsstaus sind dabei zu einem bedeutendem Problem in dichtbesiedelten Gebieten geworden, die die Fahrt- und Transportdauer für Privatpersonen und Unternehmen schwer einschätzbar machen. Die Entstehung eines Verkehrsstaus und die daraus entstehenden Effekte sind hochdynamische Vorgänge, die durch die bestehenden Ansätze nicht ausreichend gelöst werden können.

1.1 Bestehende Ansätze zur Stauvermeidung

Eine naheliegende Lösung zur Vermeidung von Verkehrsstaus ist ein Ausbau der vorhandenen und überlasteten Strecken. Das Straßennetz des überörtlichen Verkehrs in Deutschland (Autobahnen, Bundesstraßen, Landstraßen usw.) hat heute eine Länge von etwa 231000 km, davon etwa 12000 km Autobahn. Deutschland verfügt damit über das längste Autobahnnetz in Europa, international hinter den USA über eines der längsten Autobahnnetze. Die Ausweitung des Straßennetzes zur Bewältigung des Lkw-Güterverkehrs und des Pkw-Individualverkehrs ist in den vergangenen Jahren aufgrund begrenzter finanzieller Mittel der öffentlichen Hand stark zurückgefahren worden. Des Weiteren verhindert die begrenzte Verfügbarkeit von Flächen in einem so dicht besiedelten Land wie Deutschland den zu einer Lösung der Verkehrsstauprobleme nötigen Ausbau. Laut Angaben des statistischen Bundesamts [2] im Zeitraum von 1995 bis 2005 wurde das überortliche Straßennetzes in der Länge um etwa $1,2\%$ ausgebaut wohingegen im gleichen Zeitraum der Pkw-Bestand um $13,7\%$ wuchs. Daher ist eine Lösung der Stauproblematik durch Ausbaumaßnahmen des Verkehrsnetzes schon rein finanziell zum scheitern verurteilt.
Die heutzutage gebräuchlichen Navigationssysteme beschäftigen sich mit der individuellen Routenplanung unter dem Aspekt der kürzesten Entfernung. Neuere Systeme arbeiten mit Verkehrsinformationssystemen wie TMC oder TMCpro, übertragen Informationen über die Verkehrssituation über Radiodatenkanäle und reagieren auf bekannte Verkehrsengpässe, indem sie nach Kürzeste-Wege-Algorithmen eine alternative Route berechnen. Da die Navigationssysteme aller Fahrzeuge bei Verkehrsbehinderungen untereinander unkoordiniert und deterministisch nach denselben Algorithmen die gleiche Ausweichroute vorschlagen, wird der Engpass auf die Straßen der Ausweichroute verlagert.
Moderne Verkehrsleit- und Planungssysteme sollen durch dynamische Zufahrtsregelungen oder variable Geschwindigkeitsvorgaben den Verkehrsdurchsatz „optimieren“. Zum Einsatz kommen hierbei

Ampeln, die an Autobahnauffahrten in kurzen Intervallen Fahrzeuge einzeln auf die Autobahn fahren lassen, mit der Folge, dass sich Warteschlangen vor diesen Zufahrtsampeln bilden. Wieder wird der Engpass nur verlagert. Digitale Verkehrsbeschilderungen leiten Fahrzeuge bei erhöhtem Verkehrsaufkommen auf (vorher definierte) Umleitungen ohne die eigentlichen Start-Ziel-Vorgaben einzelner Fahrzeuge berücksichtigen zu können. Fahrzeuge, die sich nur wenige Kilometer von ihrem Fahrtziel entfernt befinden, werden ebenso umgeleitet wie Fahrzeuge, die soeben ihren Ausgangspunkt verlassen haben, was ganz offensichtlich eine suboptimale Behandlung der vorliegenden Probleme mit ihren jeweils unterschiedlichen Anforderungen darstellt. Hinzu kommt, dass eine basierend auf den schlechten Erfahrungen, die bislang mit solchen Systemen gemacht wurden, das Vertrauen in diese äußerst gering ist, was zur Folge hat, dass moderne Verkehrsleitsysteme vom Benutzer weitgehend ignoriert werden.

1.2 Ansatz On-line Verkehrsrouting

Ein verteiltes on-line Verkehrsrouting ermöglicht die intelligente Verteilung von Fahrzeugen auf mehrere mögliche Ausweichrouten ohne die von herkömmlichen Navigationssystemen verursachte Verlagerung der Verkehrsengpässe. Dabei werden während der Fahrt von den am System beteiligten Fahrzeugen Daten über die Verkehrssituation und die von den Fahrern gewünschten Ziele gesammelt. Durch die Möglichkeit der Kommunikation zwischen den beteiligten Verkehrsagenten können drohende Verkehrsengpässe bereits in der Enstehung erkannt und darauf reagiert werden. Ein zentraler Ansatz zur Koordination der Fahrzeuge ist hierbei nicht zu realisieren, da die anfallenden Datenmengen zu groß sind und die Echtzeitfähigkeit der Routingentscheidungen eine wichtige Anforderung an das System ist. Die in diesem Bericht beschriebene Lösung setzt auf die Verwendung eines Multipfad-Algorithmus. Die Routingentscheidungen werden von einem verteilten Multiagentensystem getroffen.

1.3 BeeJamA

Der in diesem Bericht vorgestellte Algorithmus BeeJamA hat das Ziel, die Fahrzeuge auf einem Verkehrsnetz an ein von den Fahrern gewähltes Ziel in möglichst geringer Zeit zu routen und auf den Strecken im Straßennetz Verkehrsstaus zu vermeiden. Er basiert auf den Prinzipien der naturinspirierten Multipfad-Algorithmen BeeHive [19] und BeeAdHoc [20], die für das Routing von Datenpaketen in Computernetzen entworfen wurden.
Um die Effizienz des Algorithmus hinsichtlich der Stauvermeidung als auch der Minimierung der Fahrzeiten zu bewerten wurde der in diesem Bericht vorgestellte Verkehrssimulator entwickelt, der mit einem komplexen Modell des Straßennetzes des Ruhrgebiets arbeitet.

Kapitel 2

Bienenalgorithmen

2.1 Bienen in der Natur

Eine Bienenkolonie hat die Fähigkeit, die Ausnutzung mehrerer Nahrungsquellen verschiedener Typen (Nektar, Pollen, Wasser) den aktuellen Änderungen in der Versorgungslage anzupassen. Diese Fähigkeit wird alleine durch lokale Entscheidungen einzelner Bienen in dezentralisierter Weise getroffen und benötigt nur geringe Kommunikation zwischen den Bienen.

2.1.1 Das Verhalten der Kolonie

Eine Bienenkolonie besteht aus *Packern*, *Scouts* und *Foragers*.
Eine Kolonie hat lediglich wenige Scouts, deren Aufgabe es ist, neue Nahrungsquellen zu finden. Hat ein Scout eine Nahrungsquelle gefunden, so kehrt er als Forager zum Bienenstock zurück und begibt sich dort in einen Bereich, der *Dancefloor* genannt wird.
Durch den so genannten Schwänzeltanz wird dort versucht andere Bienen für den Nahrungstransport von der entdeckten Quelle zu rekrutieren. Dabei wird durch die Geschwindigkeit des Tanzes die Distanz zur Nahrungsquelle und die Richtung durch den Winkel der Richtung des Schwänzeltanzes zur Sonne mitgeteilt.
Forager die rekrutiert wurden, versuchen nach ihrer Rückkehr in den Bienenstock wiederum andere Bienen zu rekrutieren. Sollte ein Forager nach dem Transport der Nahrung allerdings im Bienenstock zu lange auf einen Packer gewartet haben, so deutet dies darauf hin, dass die Nahrung schneller in den Stock gebracht wird als sie dort verstaut werden kann. Dieser Forager teilt den Foragern dies mit, indem er den so genannten Zittertanz aufführt, der die tanzenden Forager veranlasst, ihre Tänze zu beenden. Als weitere Möglichkeit kann auch nach der Rückkehr des Foragers kein Tanz durchgeführt werden. Die Entscheidung hierüber hängt vom Nutzen der Nahrungsquelle und der Wartezeit auf einen Packer (siehe Abbildung) nach Rückkehr in den Stock ab.
Wenn eine Nahrungsquelle versiegt wenden sich die Forager von dieser Quelle ab und kehren zum Bienenstock zurück um für neue Aufgaben zur Verfügung zu stehen. Von Zeit zu Zeit prüft ein Forager eine ehemals benutzte Quelle, um zu erkennen, ob sich wieder Nahrung nachgebildet hat.
Die Packer haben die Aufgabe, die von den Foragern angelieferte Nahrung zu verstauen.

2.1.2 Der Schwänzeltanz

Der Schwänzeltanz wird von den Foragern benutzt, um andere Forager für die Transportarbeit an einer Nahrungsquelle zu rekrutieren. Dabei wählen die Forager zufällig aus, welchem Tanz sie folgen, ohne dabei mehrere Tänze zu vergleichen. Ein Tanz scheint keine Information zu enthalten, die den Foragern bei der Wahl der Nahrungsquelle helfen.
Durch das zufällige Verhalten der Forager wird eine reichere Nahrungsquelle nur bevorzugt, wenn mehr Bienen diese durch ihre Tänze bewerben.

2.1.3 Der Zittertanz

Die Forager führen mit höherer Wahrscheinlichkeit den Zittertanz nach der Rückkehr in den Bienenstock durch, wenn sie lange auf einen Packer gewartet haben, um die Nahrung abzuladen. Forager führen den Zittertanz auf dem *Dancefloor* sowie im Brutnest durch, wohingegen der Schwänzeltanz nur auf dem Dancefloor durchgeführt wird. Also werden durch den Zittertanz wohl auch Bienen im Stock angesprochen. Nach Seeley [16] werden Arbeiterbienen im Stock durch den Tanz aufgefordert, ihre aktuelle Arbeit zu beenden und stattdessen als Packer auszuhelfen.
Im Dancefloor werden die anderen Forager durch den Zittertanz angeregt, ihre Schwänzeltänze zu beenden.

2.2 Die Anpassungsfähigkeit der Bienen an ihre Umgebung

Ein Forager bewertet die Qualität einer Nahrungsquelle ohne Kommunikation mit anderen Bienen und ohne direkten Vergleich mit anderen Nahrungsquellen. Eine sehr weit vom Bienenstock entfernte Quelle mit hohem Nahrungsgehalt wird von einem Forager schlechter bewertet als nahe Quellen mit niedrigerem Nahrungsgehalt. Die Bewertung geschieht hierbei durch die Wahrscheinlichkeit eines Schwänzeltanzes nach Rückkehr in den Bienenstock.
Die Forager im Dancefloor können folglich den Nahrungsgehalt einer von einem anderen Forager beworbenen Nahrungsquelle nicht bewerten.

2.3 BeeHive

BeeHive [19] ist ein effizienter, skalierbarer, dynamischer und fehlertoleranter Routingalgorithmus, inspiriert vom Verhalten der Honigbienen. Das Kommunikationsmodell der Bienen wurde für das Design intelligenter Agenten benutzt, die für den Einsatz in grossen, komplexen Topologien geeignet sind.
Agentenbasierte Algorithmen, in denen Agenten als spezialisierte Einheiten zentrale Aufgaben wahrnehmen, haben den Vorteil, dass sie keinen globalen Controller benötigen. Einzelne Agenten helfen dabei Informationen über den Netzwerkstatus zu sammeln und an den besuchten Knoten weiterzugeben. Diese Informationen helfen dann in einer dezentralisierten Art Entscheidungen zu treffen.
Der BeeHive Algorithmus ist ausserdem fähig, sich selbständig an den Änderungen des Netzwerks und des Netzwerkverkehrs anzupassen. Durch mobile Agenten können zusätzlich Verwaltung und Kontrolle des Netzwerks gesichert werden.

2.3.1 Das BeeHive Agentensystem

Das BeeHive Agentensystem besteht aus zwei Agententypen: *short distance bee agents* und *long distance bee agents*. Beide Agenten haben die gleiche Aufgabe: das Netzwerk zu erkunden und die Wege, die sie zurücklegen zu bewerten. Sie unterscheiden sich lediglich in der Anzahl von Knoten, auch Hops genannt, die sie besuchen können.
Short distance bee agents können, wie der Name schon vermuten lässt, nur kurze Strecken zurücklegen. Sie haben eine obere Grenze an Hops, *short distance limit* genannt, die sie besuchen können. Ist dieser Wert erreicht, entscheiden sich die Agenten ihre Reise zu beenden. Man sagt deswegen dass sie Informationen nur in der Nachbarschaft verbreiten. Im Rahmen der Projektgruppe 439, an der Universität Dortmund, wurde der BeeHive Algorithmus implementiert und dessen Performance untersucht [1]. Dabei entschieden sich die Teilnehmer eine *short distance limit* von sieben zu benutzen, da dieses die beste Performance zeigte. *Short distance bee agents* durften dementsprechend nur einen Weg zurücklegen der nicht länger als sieben war.
Im Gegensatz zu den *short distance bee agents* sammeln und verbreiten *long distance bee agents* Informationen praktisch im ganzen Netzwerk. Sie haben auch eine obere Grenze von Hops, *long distance limit* genannt, die sie besuchen können. Typischerweise ist dieser Wert so hoch, dass alle Knoten im Netzwerk damit erreicht werden können. Somit werden Informationen über Knoten, die von den *short distance bee agents* nicht erreicht werden, gesammelt.
Bienenagenten die vom gleichen Knoten gesendet werden bilden eine verwandte Gruppe die gegenseitiges Interesse aneinander zeigen. Kommen zwei Agenten der gleichen Gruppe am gleichen Knoten aber über unterschiedliche Pfade an, greifen sie auf die Routinginformationen die ihre Verwandten zuvor gesammelt haben zu. Sie entscheiden sich die Reise nicht mehr fortzuführen, nachdem sie ihre Informationen in der Routingtabelle gespeichert haben, wenn ein Verwandter den gleichen Knoten schon vorher besucht hat.

Foraging region

In BeeHive wird das Netzwerk in feste Partitionen von Knoten eingeteilt, die man *foraging regions* nennt. Die Knoten des Netzwerks werden vom sogenannten *BeeHive Flooding Algorithmus* in Abhängigkeit der Besonderheiten des Netzwerks und der Anzahl an Hops die ein *short distance bee agent* zurücklegen kann in Regionen eingeteilt. Lässt man diesen Algorithmus einmal laufen werden alle Knoten eindeutig einer Region zugeordnet. Zusätzlich legt der Algorithmus für jede Region einen eindeutigen Knoten, der als *repräsentativer Knoten* (*representative node*) der Region betrachtet wird, fest. Dieser Knoten darf als einziger Knoten der Region *long distance bee agents* versenden. Ein Pseudocode des *BeeHive Flooding Algorithmus* sowie detailliertere Angaben zur Einteilung wurden im Forschungsbericht 801 an der Universität Dortmund veröffentlicht [19].
Ein Beispiel eines Netzwerks wird in Abbildung 2.1 dargestellt. Das Netzwerk besteht aus acht Knoten die untereinander verbunden sind. Auf diesem Netzwerk wurde der *BeeHive Flooding Algorithmus* einmal laufen gelassen. Es entstanden dadurch zwei Regionen von jeweils unterschiedlicher Grösse die durch die gestrichelte Linie abgegrenzt sind. Als repräsentative Knoten wurden die Knoten 1 und 6 ausgewählt.

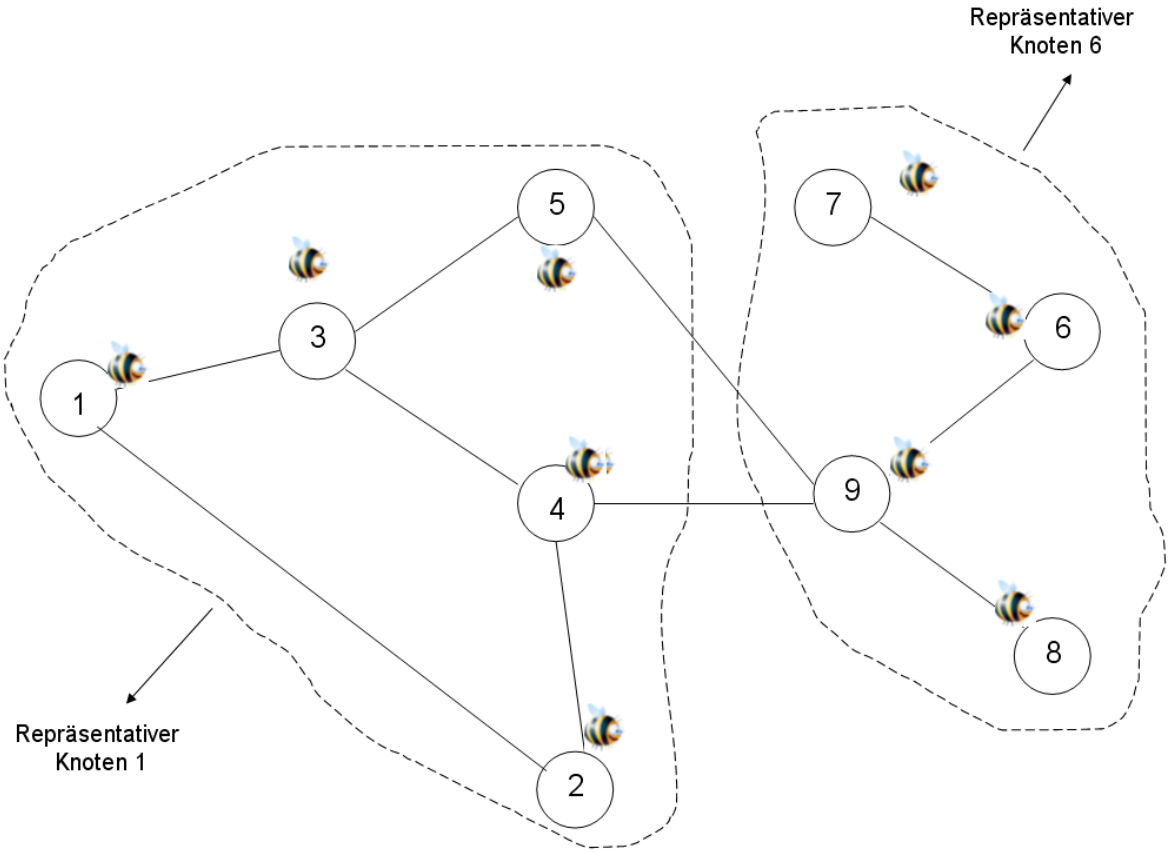


Abbildung 2.1: *Foraging regions* in BeeHive

Foraging zone

Die *foraging zone* (FZ_i) eines Knotens *i* wird als Menge aller Knoten die ein *short distance bee agent*, ausgehend vom Knoten *i*, erreichen kann, definiert. Eine Zone (*foraging zone*) kann sich dabei über mehrere Regionen (*foraging regions*) erstrecken.
Im Beispiel aus Abbildung 2.1 ist die *foraging zone* des Knotens 2 die Menge $\{1, 3, 4, 9\}$, bei einer *short distance limit* von zwei. Zu beachten ist dabei dass der Knoten 5 in der *foraging region* des Knotens 2 liegt, jedoch nicht in seiner *foraging zone*. Umgekehrt ist Knoten 9 in der *foraging zone* des Knotens 2, jedoch nicht in seiner *foraging region*.
Die Motivation die hinter diesen beiden Definitionen steht, sind die Vorteile eines flachen Routingschemata, in welcher alle Router äquivalent sind, und einer hierarchischen Routingschemata, in welcher repräsentative Knoten mehrere Funktionen als normale Router im Clusternetzwerk besitzen. In dem hybriden Schema von BeeHive besitzt jeder Knoten Routinginformationen um Knoten in seiner *foraging zone* zu erreichen und um repräsentative Knoten anderer Regionen zu erreichen. Dieser Mechanismus erlaubt es Knoten, Datenpakete, dessen Ziel ausserhalb der eigenen *foraging zone* liegen, über einen Pfad *in Richtung des repräsentativen Knoten* der zugehörigen *foraging region* zu verschicken. Folglich, braucht der BeeHive Algorithmus eine Routingtabelle die in der gleichen Grössenordnung wie die der OSPF [13, 14] liegt, jedoch hat ein repräsentativer Knoten keine Besonderen Aufgaben ausser *long distance bee agents* zu verschicken. Im Gegensatz dazu werden in einer hierarchischen Routingschemata, Datenpakete, die ausserhalb des eigenen Clusters liegen, zum Cluster Head verschickt. Dieser verschickt das Paket zum Cluster Head des Zielknotens, ehe dieser dann letztlich das Paket zum Zielknoten verschickt. Das Konzept der *foraging regions* und *foraging zones* bringt die Vorteile von kleineren Routingtabellen aber ohne den zusätzlichen Aufwand durch Cluster Heads zu routen.
Algorithmen die ein Netzwerk in Clusters einteilen haben alle die gleiche Besonderheit, dass der Cluster Head (oder der repräsentative Knoten) in der Mitte des Clusters liegt. Bei BeeHive hat dieses Konzept des *Centroids* keine signifikanten Fortschritte gebracht. Es ist nicht zwingend notwendig dass sich der repräsentative Knoten in der Mitte der *foraging region* befindet. Im Beispiel aus Abbildung 2.1 ist dies ersichtlich, da der repräsentative Knoten 1 nicht im Zentrum der Region liegt sondern am Rand.

2.3.2 Bewertungsmodell für Agenten

Im Folgenden wird das Bewertungsmodell für Agenten von BeeHive vorgestellt. Die Zeit t_{in} die ein Datenpaket braucht um einen Nachbarknoten n zu erreichen wird wie folgt berechnet:

$$t_{in} = q_{in} + tx_{in} + pd_{in} + pr_i + pr_n \tag{2.1}$$

wobei q_{in} die Laufzeit beim Schlangestehen (*queueing delay*) für den Nachbar n beim Knoten i darstellt. tx_{in} und pd_{in} sind die Laufzeiten der Sendung (*transmission delay*) bzw. der Übertragung (*propagation delay*) zwischen dem Knoten i und seinem Nachbarn n. pr_i und pr_n sind die Protokollverarbeitungszeiten (protocol processing delay) beim Knoten i bzw. n. Im Allgemeinen sind die Protokollverarbeitungszeiten im Vergleich zu der Summe der anderen Laufzeiten sehr klein, so dass man sie vernachlässigen kann. Somit ergibt sich für die Zeit t_{in} Gleichung 2.2.

$$t_{in} \approx q_{in} + tx_{in} + pd_{in} \tag{2.2}$$

Dabei bezeichnet tx_{in} die Verzögerung die ein Datenpaket wegen der begrenzten Bandbreite erfährt, während pd_{in} die Laufzeit, die ein Datenpaket, um vom Knoten i zum Nachbarn n zu gelangen, braucht, bezeichnet. tx_{in} ist von der Grösse des Datenpaketes und von der Bandbreite der Verbindung abhängig. pd_{in} ist von der Entfernung der beiden Knoten abhängig. Für eine bestimmte Verbindung ändert sich keiner dieser Werte mit der Belastung der Verbindung. Im Gegensatz dazu verändert sich q_{in} sehr wohl mit der Belastung der Verbindung. Man approximiert q_{in} wie folgt:

$$q_{in} \approx \frac{ql_{in}}{b_{in}} \tag{2.3}$$

wobei ql_{in} die Grösse der Queue (in Bits) für den Nachbar n beim Knoten i bezeichnet. b_{in} bezeichnet die Bandbreite der Verbindung zwischen dem Knoten i und dem Nachbarn n.

2.3.3 Güte eines Nachbarn

In BeeHive wird die Güte eines Nachbarn wie folgt definiert:

$$g_{jd} = \frac{\frac{1}{p_{jd}+q_{jd}}}{\sum_{k=1}^{n} \frac{1}{p_{kd}+q_{kd}}} \tag{2.4}$$

Die Motivation die hinter dieser Definition steht ist das Verhalten eines realen Netzwerkes zu approximieren. Wenn im Netzwerk ein hoher Datenverkehr herrscht, dann spielt die Laufzeit beim Schlangestehen die wichtigste Rolle in der Laufzeit einer Verbindung. In diesem Fall ist es trivial, dass gilt $q_{jd} >> p_{jd}$ und die Güte wird dann wie folgt berechnet:

$$g_{jd} = \frac{\frac{1}{q_{jd}}}{\sum_{k=1}^{n} \frac{1}{q_{kd}}} \tag{2.5}$$

Ist der Datenverkehr im Netzwerk gering, dann spielt die Übertragungslaufzeit eine wichtige Rolle in der Laufzeit einer Verbindung. Somit ist $q_{jd} << p_{jd}$ und die Güte wird zu:

$$g_{jd} = \frac{\frac{1}{p_{jd}}}{\sum_{k=1}^{n} \frac{1}{p_{kd}}} \tag{2.6}$$

2.3.4 BeeHive - ein *packet switching* Algorithmus

Datenpakete werden in BeeHive immer in Richtung des Zielknotens gesendet. Zwei Datenpakete die den selben Ausgans- und Zielknoten haben müssen dabei nicht die gleiche Route wählen. Somit wird verhindert, dass ein guter Pfad von allen Paketen benutzt wird und die Route entlastet. An jedem Knoten wird jeweils nur der nächste Knoten ausgewählt. Um den nächsten Hop aus der Liste der Nachbarn auszuwählen wird ein stochastisches Selektionsverfahren benutzt. Dieses Prinzip besagt dass ein Nachbar j mit Güte g_{jd} mit der Wahrscheinlichkeit ϕ^i_{jd} als nächsten Hop zum Zielknoten d ausgewählt wird. Konkret wird die Wahrscheinlichkeit, um den Nachbar j als nächsten Hop zum Zielknoten d, beim Konten i, wie folgt berechnet:

$$\phi^i_{jd} = \frac{g_{jd}}{\sum_{k=1}^{n} g_{kd}} \tag{2.7}$$

Um Datenpakete in Richtung des Zielknotens senden zu können besitzt jeder Knoten i drei Arten von Routingtabellen. *Intra Foraging Zone (IFZ)*, *Inter Foraging Region (IFR)* und *Foraging Region Membership (FRM)*. Auf den Aufbau und Inhalt der Tabellen wird hier nicht weiter eingegangen da diese ähnlich zu den in Kapitel 2.4.5 beschriebenen Tabellen sind. Einige Beispiele von BeeHive Routingtabellen sowie detaillierte Angaben zu diesen sind aus [20] zu entnehmen.

2.4 BeeJamA - Verkehrsrouting auf Straßennetzwerke

2.4.1 Systemmodell

Modelliert wurde ein Straßensystem, welches Verkehrswege unterschiedlichster Kapazität berücksichtigt und in einem geeigneten Graph darstellt, angefangen bei mehrspurigen Autobahnen, die das Wechseln einzelner Spuren erlauben, über Land- und Umgehungsstraßen, die einspurig und über weite Strecken ungeregelt sind, bis hinzu innerstädtischen Straßen, die verschieden geschwindigkeits- und ampelreguliert sind. Kanten des Graphen entsprechen dabei den Verkehrswegen und Knoten den Punkten an denen Routing-Entscheidungen getroffen werden können, zum Beispiel Kreuzungen, Auf- und Abfahrten etc.. Als Vorbild dient das eng verknüpfte Straßennetz des Ruhrgebiets. Das Systemmo-

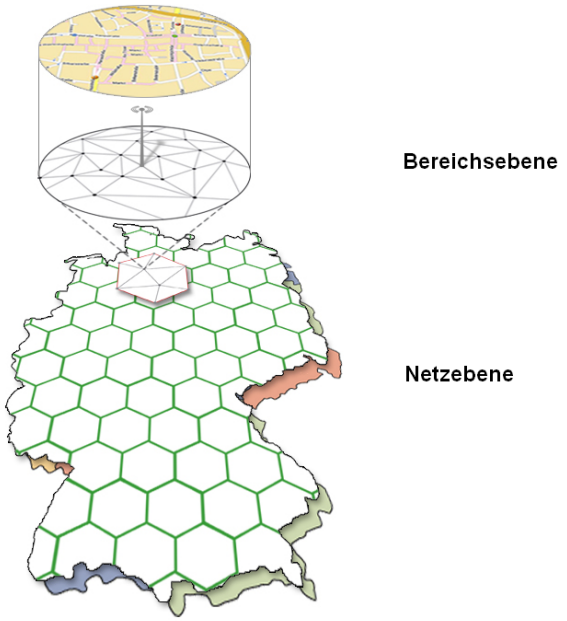


Abbildung 2.2: Ebenen im Systemmodell

dell besteht aus zwei unterschiedlichen Ebenen. Die tatsächlich vorhandene, geographische Straßentopologie wird in Bereiche eingeteilt und auf der ersten Ebene auf einen Verkehrsgraphen abgebildet, bei dem die Kanten den Straßen und die Knoten möglichen Abzweigungen entsprechen. Jeder Bereich wird von einem Navigator verwaltet, der Informationen über vorhandene Fahrzeuge sammelt und bearbeitet und Routingvorschläge an diese Fahrzeuge zurücksendet. Das Routing zwischen einzelnen

Bereichen erfolgt auf der Netzebene. Auf dieser Ebene werden als Knoten die Bereiche, dargestellt durch ihre Navigatoren, als Knoten abgebildet. Die Kanten stellen die logischen Kommunikationskanäle zwischen den Navigatoren dar. Auf der Netzebene sind zwei Knoten (Bereiche) nur durch eine Kante miteinander verbunden, wenn sie direkte Nachbarn sind. In diesem Fall kann man von dem aktuellen Bereich über einen Bordernode direkt in den benachbarten Bereich gelangen. Fahrzeuge werden also zuerst *in Richtung* ihres Zielbereichs geroutet. Sobald sie sich innerhalb dieses Bereiches befinden, erfolgt das Routing auf Bereichsebene und wird vom zuständingen Navigator verwaltet. Auf der Netzebene kommunizieren die Navigatoren untereinander, um so ein beschränktes Bild ihrer Umgebung zu bekommen.

2.4.2 Prozessmodell

Die Qualität von Verkehrsnetzen soll durch eine dynamische, dezentrale Steuerung der Fahrzeuge verbessert werden. Die lokalen Navigatoren treffen in Realzeit Entscheidungen, die die Gesamtqualität des Netzes verbessert. Im Rahmen dieser Entscheidungen wird die nächste Teilstrecke rechtzeitig vor der nächsten Kreuzung oder Abzweigung gewählt. Das Routing erfolgt dezentral und autonom, unter Berücksichtigung der lokalen Informationen, ohne dass eine Notwendigkeit von Informationen über das Gesamtnetzwerk besteht. Ziel ist, unter Betrachtung der umgebenden Stausituationen, für einzelne Fahrzeuge die günstigste Route zu bestimmen. Mit Hilfe einer Bewertungsfunktion werden zu diesem Zweck einzelnen Straßen verschiedene Kostenwerte zugeordnet, die für das Routing relevant sind. Diese Werte werden regelmäßig aktualisiert und Routingentscheidungen werden dynamisch geändert. Die Funktion, die diese Kosten berechnet, hat als Ziel die Minimierung der durchschnittlichen Fahrtzeit. Diese Berechnung betrachtet mehrere relevanten Kriterien und wird später in 2.5 ausführlich erläutert.

Routing von Punkt zu Punkt

Um Routingempfehlungen zu berechnen, ben¨otigen die Navigatoren die relevanten Informationen. Diese beinhalten, für jedes Fahrzeug:

- die aktuelle Position
- das Endziel
- die aktuelle Geschwindigkeit

Mit Hilfe dieser Informationen kann der Navigator die zum Routing benötigten Werte (Verkehrsdichte, Qualität) der Verkehrswege berechnen. Fahrzeuge teilen regelmaeßig diese Daten dem Navigator, in deßen Verwaltungsbereich sie sich befinden, mit. Je nachdem, ob sich der Endziel in demselben Bereich befindet, wird das Routing auf der Bereichsebene oder auf der Bereichs- und Netzebene durchgeführt. Sollte das Endziel in dem Bereich sein, in dem sich das Fahrzeug aktuell befindet, sind die Routingtabellen der Knoten ausreichend und das Routing kann auf der Bereichsebene komplett durchgefuehrt werden. Abhängig von den Kosten die aktuell in den Routingtabellen eingetragen sind, wird das Fahrzeug so gesteuert, daß seine Fahrtzeit unter den gegebenen Bedingungen minimiert wird. Wenn sich das Endziel in einem anderen Bereich befindet, wird zuerst festgelegt, welcher der Zielbereich ist. Danach wird das Routing auf der Netzebene durchgeführt und bestimmt, in Richtung von welchem Nachbar des aktuellen Bereiches das Fahrzeug gesteuert werden soll. Auf Bereichsebene

wird dann bis zur Grenze dieses Nachbarbereichs geroutet. Sobald sich das Fahrzeug in dem Nachbarbereich befindet, wird es von dem Navigator, der für diesen Bereich zuständig ist, verwaltet. Ist der Zielpunkt in diesem Bereich enthalten, so wird das Fahrzeug dahin, sonst zum Grenzpunkt des nächsten Bereichs gesteuert. Das Routing auf der Netzebene bestimmt also grob, durch welche Bereiche das Fahrzeug gesteuert werden soll. Innerhalb der einzelnen Bereiche wird auf der Bereichsebene unter der Verwaltung des Navigator geroutet.

Threshold

Wie vorher schon erwähnt, benutzt *BeeJamA* eine Kostenfunktion zur Bewertung der Verkehrswege. Diese soll für jede Kante die aktuelle Stausituation beschreiben. Diese Kosten werden in den Routingtabellen gespeichert und regelmässig aktualisiert. Um die Straßen bester Qualität immer zu bevorzugen, wurde der *Threshold* Parameter eingeführt. Dieser kann Werte zwischen 0 und 1 annehmen. Mit Hilfe des Thresholds kann man sicherstellen, dass Straßen, die eine schlechte Qualität bzw. hohe Kosten haben gar nicht ausgewählt werden können, wenn diese über einem bestimmten Grenzwert liegen. Wenn der Thresholdparameter auf 1 gesetzt wird, werden die Fahrzeuge immer auf die beste mögliche Route gesteuert. Sobald sich aber zu viele Fahrzeuge auf einer Kante befinden, die als beste Möglichkeit eingeschätzt wurde, steigen die Kosten dieser Kante. Die Routingtabellen werden aktualisiert und eine andere Route wird als kostengünstiger bewertet. Ab diesem Zeitpunkt werden auch die Fahrzeuge auf diesen alternativen Weg geschickt. In einem eng gekoppelten Verkehrsnetz besteht aber auch oft die Möglichkeit, mehrere gute Routen zu haben. In diesem Fall ist es sinvoll den Thresholdparameter herunterzusetzen, zum Beispiel auf 0.9 oder 0.8. Damit würde nicht nur die beste Route gewählt werden, sondern auch Wege, die eine unwesentlich kleinere Qualität haben und die eine ähnliche Fahrtzeit anbieten. So würde man sicherstellen, dass mehrere akzeptable Routen in Betracht gezogen werden, und nicht der ganze Verkehr auf einer einzigen konzentriert wird.

2.4.3 Routing auf der Bereichsebene

Die erste Art von Routing die in BeeJamA eingesetzt wird ist das Routing auf der Bereichsebene. Dieses Verfahren ist eine veränderte Version des BeeHive Routing Algorithmus. Es wurde speziell an den Anforderungen eines Routings in einem einzigen Bereich angepasst. Im folgenden werden zuerst die Eigenschaften des Routings auf der Bereichsebene beschrieben um danach die Routingtabellen des Verfahrens vorzustellen.

Eigenschaften

Beim Routing auf der Bereichsebene wurden gewisse Eigenschaften des Routings und der Bereiche festgelegt.

- Die erste und auch wichtigste Eigenschaft ist dass die Zonen der unterschiedlichen Knoten, im Gegensatz zum normalen BeeHive, alle gleich sind. In BeeHive wurde eine Zone als Menge aller Knoten die von einem *short-distance bee agent* erreicht werden kann definiert. Beim Routing auf der Bereichsebene wird die Zone *(foraging zone)* als Menge aller Knoten die im selben

Tabelle 2.1: IFZ_{area}-Tabelle in BeeJamA

IFZ_{area}	$D_1(i)$	$D_2(i)$	...	$D_d(i)$
$N_1(i)$	c_{11}	c_{12}	...	c_{1d}
$N_2(i)$	c_{21}	c_{22}	...	c_{2d}
...	...	...	$\ddots$	...
$N_n(i)$	c_{n1}	c_{n2}	...	c_{nd}

Bereich liegen definiert. Somit stimmt die *foraging zone* jedes Knotens mit dem Bereich überein. Desweiteren sind auch alle *foraging zones* der Knoten in einem Bereich gleich.
Mit dieser Eigenschaft wird erreicht dass jeder Knoten in einem Bereich weiss wie er einen anderen Knoten im selben Bereich erreichen kann. Wege zu einem anderen Bereich werden durch das Routing auf der Netzebene realisiert.

- Die zweite Eigenschaft ist dass die Regionen des Netzes auch festgelegt sind. Sie stimmen ebenfalls mit den Bereichen überein.
  Somit wird erreicht dass jeder Knoten, der einen Knoten aus einer anderen Region erreichen will, diesen mittels des Routings auf der Netzebene erreichen kann. Der zu erreichende Knoten liegt dann auch in einem anderen Bereich der auf der Netzebene als Knoten dargestellt wird. Das Routing auf der Netzebene gibt dann den nächsten Hop, bzw. den nächsten Bereich, zurück und somit auch die nächste Region die zu erreichen ist.

Entsprechend dieser zwei Eigenschaften ergeben sich 3 Arten von Routingtabellen die in den nächsten Unterkapiteln beschrieben werden.

Intra Foraging Zone Tabelle - IFZ_{area}

Die erste Routingtabelle ist die Intra Foraging Zone Tabelle, kurz IFZ_{area}. Jeder Knoten besitzt eine eigene IFZ_{area}-Tabelle. Die Tabelle hat eine Grösse von $|N(i) \times D(i)|$, wobei $N(i)$ die Menge der Nachbarn des Knotens i ist und $D(i)$ die Menge aller Knoten in dem Bereich von i ist. Jedes Element der Matrix ist eine Zahl c_{jd} die die Kosten repräsentiert, um den Zielknoten d über den Nachbar j zu erreichen. Die Kosten werden dabei, genau wie in der IFZ_{net}, als Zeit dargestellt.
Tabelle 2.1 zeigt ein Beispiel einer IFZ_{area}-Tabelle in BeeJamA. In der obersten Zeile werden alle Knoten aus dem Bereich des Knotens i eingetragen und auf der ersten Spalte die Nachbarn des Knotens. Somit werden am Knoten i alle Kosten, um alle Knoten in seinem Bereich zu erreichen, gespeichert. Mittels dieser Kosten kann dann stochastisch einen Nachbar als nächsten Hop zum Ziel ausgewählt werden.

Inter Foraging Region Tabelle - IFR_{area}

Die zweite Routingtabelle ist die Inter Foraging Region Tabelle, kurz IFR_{area}. Im Gegensatz zum BeeHive ist diese Tabelle für jeden Knoten aus einem Bereich gleich. Deshalb muss sie auch nur einmal gespeichert und verwaltet werden.
Die IFR_{area}-Tabelle gibt an, über welche Knoten man zu den benachbarten Bereichen kommt. Die Grösse der Tabelle ist $|NB(i) \times G(i)|$, wobei $NB(i)$ die Menge der benachbarten Bereiche und $G(i)$ die Menge der Grenzknoten bezeichnen. Grenzknoten sind Knoten die Übergänge zu anderen Knoten

Tabelle 2.2: IFR_{area}-Tabelle in BeeJamA

IFR_{net}	$NB_1(i)$	$NB_2(i)$	...	$NB_d(i)$
$G_1(i)$	c_{11}	c_{12}	...	c_{1d}
$G_2(i)$	c_{21}	c_{22}	...	c_{2d}
...	...	...	$\ddots$	...
$G_n(i)$	c_{n1}	c_{n2}	...	c_{nd}

Tabelle 2.3: FRM_{area}-Tabelle in BeeJamA

Knoten	1	2	3	...	n
Bereich	A	B	B	...	N

aus anderen Bereichen haben.
Tabelle 2.2 zeigt ein Beispiel einer IFR_{area}. In der obersten Zeile werden alle Nachbarbereiche eingetragen und auf der ersten Spalte alle Grenzknoten des Bereichs. Somit werden am Knoten i die Kosten um Nachbarbereich über Grenzknoten zu erreichen gespeichert. Mittels der IFR_{area}-Tabelle können somit Fahrzeuge den nächsten Grenzknoten zum Nachbarbereich auswählen.

Foraging Region Membership Tabelle

Die dritte, und letzte, Routingtabelle ist die Foraging Region Membership Tabelle, kurz FRM_{area}. Diese stellt eine Abbildung der Knoten auf Bereiche bzw. Regionen dar. Da die Zuweisung der Knoten zu den Bereichen eindeutig ist, ist die FRM_{area}-Tabelle für alle Knoten und Bereiche gleich.
Die Grösse dieser Abbildung ist durch die Anzahl der Knoten im Netz beschränkt und ist mit der Grösse eines Postleitzahlenbuches zu vergleichen. Da die FRM_{area}-Tabelle für alle Knoten im Netz identisch ist (sie informiert nur über die Zugehörigkeit der Knoten zu Bereiche, die gleich für alle Knoten ist) muss sie auch nur einmal gespeichert und verwaltet werden. Die Agenten können dann jeweils Anfragen dieser Tabelle stellen.

2.4.4 Aktualisieren der Tabellen auf der Bereichsebene

Auf der Bereichsebene haben die Routingtabellen eine andere Struktur als beim BeeHive Algorithmus (2.3.1). Da sie den Anforderungen von Bereichen angepasst werden, musste auch die Aktualisierung der Tabellen angepasst werden. Hierzu wurde ein iteratives Verfahren gegenüber dem rekursiven Verfahren bevorzugt. Das rekursive Verfahren könnte mittels Algorithmen wie Dijkstra oder Distance Vector realisiert werden und hat den grossen Vorteil dass zu jedem Zeitpunkt die genauen Kosten in den Routingtabellen stehen würden. Somit wäre an jedem Knoten die genaue Situation des Bereiches bekannt. Der grosse Nachteil der rekursiven Methode ist allerdings die grosse Ausführungszeit. Einen solchen Algorithmus mit grosser Laufzeit in jedem Schritt laufen zu lassen ist sehr aufwändig. Deswegen wurde ein iteratives Verfahren ausgewählt das im folgenden vorgestellt wird.
Beim Start der Anwendung werden alle Einträge mit unendlich initialisiert. Einen Eintrag in einer Tabelle mit unendlich bedeutet dass die Kosten die Strecke zurückzulegen unendlich sind. Sind zwei Knoten nicht miteinander verbunden, haben aber wegen dem Aufbau der Tabelle trotzdem einen entsprechenden Eintrag, so wird dieser Wert auf unendlich bleiben und die Kante wird als mögliche

Variante nicht in betracht gezogen.
Algorithmus 1, ähnlich dem Algorithmus zum Aktualisieren der Tabellen auf der Netzebene (2.4.6), wird für das Aktualisieren der Tabellen auf der Bereichsebene verwendet.

Algorithmus 1 Aktualisieren der Tabellen auf der Bereichsebene

1: **Init:** Initialisiere alle Einträge mit ∞
2: **for all** iterations **do**
3: aktualisiere die Kosten zu den direkten Nachbarknoten
4: $c_D = kostenZuNachbar(D)$;
5: aktualisiere die Kosten zu den anderen Knoten aus der Tabelle
6: $c_{JD} = kostenZuNachbar(D) + kostenVonNachbarNach(D, J)$;
7: **end for**

Der Unterschied zu der Netzebene liegt hierbei ausschliesslich bei der Berechnung der Kosten zu nicht benachbarten Knoten, es gibt nähmlich keine Kosten um Bereiche zu überqueren, sondern die Kosten dabei lassen sich einfach aus der Summe der Kosten zum Nachbar und der Kosten vom Nachbar zum Zielknoten berechnet. Wird diese Iteration in jedem Aktualisierungschritt ausgeführt, so wird nach spätetens n Iterationen, bei einer Tabelle der Grösse $n \times n$ die komplette Tabelle ausgefüllt sein. Dieses Verfahren wurde bei der Implementierung des Simultators (4.11.1) erfolgreich eingesetzt und zeigte deutlich seine Vorteile (5) gegenüber dem Dijkstra Algorithmus. Desweiteren erlaubt das iterative Verfahren Aktualisierungsraten von unter einer Sekunde.

2.4.5 Routing auf der Netzebene

Die Netzebene in *BeeJamA* ist eine Abstraktionsebene um Fahrzeuge zu einem weiter entfernten Ziel zu routen. Müssen Fahrzeuge über mehrere Bereiche gerouted werden, dann müssen sie zuerst über die Netzebene geroutet werden bis sie in den Zielbereich gelangen. Sind sie im Zielbereich angekommen, dann werden sie weiter auf der Bereichsebene geroutet was im Kapitel 2.4.3 erläutert wird. Das Prinzip des Routings wird in Abbildung 2.3 nochmal kurz dargestellt. Die Motivation für diese Einteilung wird im folgenden erläurtert.

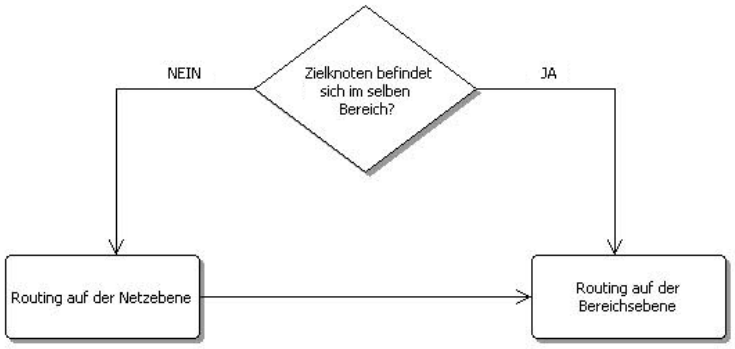


Abbildung 2.3: BeeJamA Routing auf mehreren Ebenen

Tabelle 2.4: IFZ_{net}-Tabelle in BeeJamA

IFZ_{net}	$D_1(i)$	$D_2(i)$	...	$D_d(i)$
$N_1(i)$	c_{11}	c_{12}	...	c_{1d}
$N_2(i)$	c_{21}	c_{22}	...	c_{2d}
...	...	...	$\ddots$	...
$N_n(i)$	c_{n1}	c_{n2}	...	c_{nd}

BeeJamA basiert auf den im Kapitel 2.3 beschriebenen BeeHive. Die Einteilung der Tabellen zum Routing wurde für die Netzebene übernommen da im Grunde keinen Unterschied zum normalen BeeHive existiert. Einzelne Bereiche werden als Knoten repräsentiert und einen Pfad zwischen den Bereichen wird als Kante repräsentiert. Somit ergibt sich ein Netzwerk das praktisch identisch mit einem Computernetzwerk sein kann. Es ergeben sich daraus folgende Tabellen für die Netzebene die Informationen für das Routing speichern:

- IFR_{net} - *Inter Foraging Region*
- IFZ_{net} - *Intra Foraging Zone*
- FRM_{net} - *Foraging Region Membership*

Die Routingtabellen haben das gleiche Format wie im BeeHive Algorithmus wobei jedes Element der Tabelle anstatt einem Paar von *queueing delay* und *propagation delay* die Kosten einer Kante speichert. Der Aufbau der Tabellen wird im folgenden kurz dargestellt.

Intra Foraging Zone Tabelle - IFZ_{net}

Die *Intra Foraging Zone* Routingtabelle IFZ_{net} ist eine Matrix der Grösse $|D(i) \times N(i)|$, wobei $D(i)$ die Menge von Knoten in der *foraging zone* des Knotens i ist und $N(i)$ die Menge der Nachbarn bezeichnet. Jedes Element der Matrix ist eine Zahl c_{jd} die die Kosten repräsentiert, um den Zielknoten d über den Nachbarn j zu erreichen. Die Kosten werden dabei als Fahrtzeit über die Kante dargestellt. Wie man diese Zeit bzw. diese Kosten abschätzt und wie sie berechnet werden wird im Kapitel (2.5) erläutert. Tabelle 2.4 zeigt ein Beispiel einer IFZ_{net} in BeeJamA. In der obersten Zeile werden alle Knoten in der *foraging zone* des Knotens i eingetragen und auf der ersten Spalte die Nachbarn des Knotens. Somit werden am Knoten i alle Kosten, um alle Knoten in seiner *foraging region* zu erreichen, gespeichert. Mittels diesen Kosten kann dann stochastisch ein Nachbar als nächsten Knoten zum Ziel ausgewählt werden.

Inter Foraging Region Tabelle - IFR_{net}

Die *Inter Foraging Region* Routingtabelle ist eine Matrix der Grösse $|Z(i) \times N(i)|$, wobei $Z(i)$ die Menge der repräsentativen Knoten aller *foraging regions* ist und $N(i)$ die Menge der Nachbarn. Die IFR_{net}-Matrix ist ähnlich der IFZ_{net}-Matrix aufgebaut und speichert Kosten um die repräsentativen Knoten anderer Regionen über die Nachbarn zu erreichen.
Tabelle 2.5 zeigt ein Beispiel einer IFR_{net}-Tabelle. Mittels der IFR_{net}-Tabelle können Fahrzeuge die ausserhalb der *foraging zone* des Knotens i liegen in Richtung des repräsentativen Kontens der jeweiligen Region geroutet werden.

Tabelle 2.5: IFR_{net}-Tabelle in BeeJamA

IFR_{net}	$Z_1(i)$	$Z_2(i)$	...	$Z_d(i)$
$N_1(i)$	c_{11}	c_{12}	...	c_{1d}
$N_2(i)$	c_{21}	c_{22}	...	c_{2d}
...	...	...	$\ddots$	...
$N_n(i)$	c_{n1}	c_{n2}	...	c_{nd}

Tabelle 2.6: FRM_{net}-Tabelle in BeeJamA

Knoten	1	2	3	...	n
Region	A	A	B	...	N

Foraging Region Membership Tabelle - FRM_{net}

Die *Foraging Region Membership* Routingtabelle liefert die Abbildung der bekannten Knoten zu den repräsentativen Knoten der Region denen die Knoten gehören. Die Grösse dieser Abbildung ist durch die Anzahl der Knoten im Netz beschränkt. Da die FRM_{net}-Tabelle für alle Knoten im Netz identisch ist (sie informiert nur über die Zugehörigkeit der Knoten zu Regionen, die gleich für alle Knoten ist) muss sie auch nur einmal gespeichert werden. Die Knoten stellen dann jeweils nur Anfragen an diese Tabelle.

Durch die Einteilung in kleineren Tabellen wird der Speicherplatzbedarf reduziert. Die jeweiligen Knoten besitzen Informationen nur über ihre Umgebung und nicht über das ganze Netz, was wesentlich aufwändiger wäre.

2.4.6 Aktualisieren der Tabellen auf der Netzebene

Die Routingtabellen auf der Netzebene speichern Informationen über eine abstrakte Ebene die der Speicherplatzreduzierung bei Fernrouting dient. Dabei werden Bereiche als Knoten repräsentiert und Übergänge zwischen den Bereichen als Kanten. Im realen Straßennetz allerdings gibt es diese Ebene nicht. Auch die Kanten zwischen Knoten gibt es als Straßen nicht, denn zwischen zwei Bereichen kann es mehrere Übergange geben, jedoch auf der Netzebene werden sie durch eine einzige Kante repräsentiert. Diese zusätzlichen Eigenschaften stellen gewisse Anforderung an das Aktualisieren der Tabellen. Strategien dazu gibt es mehrere mit unterschiedlicher Effizienz. Im folgenden wird jedoch die in der Implementierung des Simulators verwendete iterative Verfahren vorgestellt.
Die eigentliche Herausforderung in der Aktualisierung der Tabellen liegt nicht am Algorithmus selbst, sondern an den Kosten der Kanten bzw. der Übergänge. Dabei unterscheidet man zwei Arten von Übergängen zwischen zwei Bereichen:

- die zwei Bereiche sind benachbart und es gibt mindestens eine Übergangskante aus dem ersten Bereich in den Zweiten
- die zwei Bereiche sind nicht benachbart; sie sind jedoch durch einen Pfad miteinander verbunden

Fall 1: Sind die beiden Bereiche benachbart, so ergeben sich die Kosten zwischen den Bereichen als eine Funktion der Kanten die aus dem ersten Bereich in den zweiten Bereich führen. Dabei muss

beachtet werden, dass die Kanten des Graphs gerichtet sind.

Fall 2: Sind die beiden Bereiche nicht benachbart sondern nur durch einen Übergangsbereich zu erreichen, so werden die Kosten zwischen den beiden Bereichen, als Summe der Kosten zwischen dem ersten Bereich und einem seiner Nachbar (Fall 1) und den Kosten vom Nachbarn zum Zielbereich und zusätzlich noch dazu die Kosten um den Nachbarbereich zu überqueren berechnet.
Es ergibt sich Algorithmus 2 zum aktualisieren der Routingtabellen auf der Netzebene.

Algorithmus 2 Aktualisieren der Tabellen auf der Netzebene

1: **Init:** Initialisiere alle Einträge mit ∞
2: **for all** iterations **do**
3: aktualisiere die Kosten zu direkten Nachbarbereichen
4: $c_N = kostenZuNachbar(D)$;
5: aktualisiere die Kosten zu den anderen Knoten aus der Tabelle
6: $c_JN = kostenZuNachbar(N) + kostenVonNachbarNach(N, J) + Transitkosten(N, J)$;
7: **end for**

Die Transitkosten um einen Bereich zu überqueren werden hierbei aus den Tabellen der Bereichsebene, die im Kapitel (2.4.3) vorgestellt werden, ausgelesen. Sie repräsentieren dabei die Kosten zwischen den jeweiligen Grenzknoten.

2.5 Vorüberlegungen zur BeeJamA Bewertungsfunktion

2.5.1 Einführung

Eine der Hauptproblemstellungen bei der Aufgabe den BeeHive-Algorithmus auf den Bereich des Rotuings von Fahrzeugen anzuwenden ist die Entwicklung einer Bewertungsfunktion. Die Bewertungsfunktion ist eine Funktion, die jeder Kante in dem vom System verwalteten Straßennetz einen Wert zuweist, der die aktuelle Verkehrssituation widerspiegelt.
Im klassischem BeeHive-Algorithmus wird diese Funktion mit Hilfe gemessener Übertragungs- und Warteschlangenverzögerungen berechnet und wird verwendet, um den Datendurchsatz des gesamten Netzes zu erhöhen. Der BeeAdHoc-Algorithmus verwendet den gemessenen Energieverbrauch von verschiedenen Routen in einem Ad-hoc-Netz in der Berechnung der Bewertungsfunktion mit dem Ziel, die Batterielebenszeit von mobilen Geräten zu maximieren. Da der Fokus des BeeJamA-Algorithmus auf der Minimierung von Staus aber gleichzeitig auch der Wahl des schnellsten Weges jedes einzelnen Fahrzeugs liegt, wurde bei der Bewertungsfunktion ein vollständig anderer Ansatz gewählt.
Die Verhinderung von dichtem Verkehr kann im Gegensatz zu den Zielen des einzelnen Fahrers stehen. Zusätzlich dürfen einzelne Fahrer nicht mit zu langen Umwegen strapaziert werden, da der Erfolg des System letzlich immernoch von der Akzeptanz der Fahrer abhängt, die letztlich doch entscheiden, ob sie den Empfehlungen des System Folge leisten. Daher muß eine Bewertungsfunktion entwickelt werden, die in der Simulation angepasst werden kann, um diese Anforderungen ausgeglichen zu erfüllen.

2.5.2 Verwendbare Kenngrößen

Der erste Schritt in der Entwicklung der Bewertungsfunktion war es, die Annahmen der technischen Möglichkeiten des Fahrzeugsubsystems, das in einem fertigen System in jedem Fahrzeug installiert würde, genauer zu spezifizieren. Jedes Fahrzeug sollte mit einem GPS-Empfänger ausgerüstet sein, um die aktuelle Position, Geschwindigkeit und Fahrtrichtung zu berechnen und zu den Navigatorknoten übertragen zu können. Mit Hilfe dieser Daten ist es den Navigatorknoten möglich, für die Straßen im Zuständigkeistbereich des Navigators Größen wie Verkehrsdichte und Durchschnittsgeschwindigkeit zu berechnen (s. Darstellung 2.4 dynamisch), die bei der Bewertung der Verkehrssituation hilfreich sind.

Zusätzlich kann der Navigator statische Informationen aus der digitalen Repräsentation des Straßennetzes zur Berechnung der Bewertungsfunktion verwenden. Bei dem im Simulator verwendeten Datenformat werden die Länge, der Straßentyp (z.B. Autobahn) und ein sogenannter *road level*-Wert für jede Straßenkante zur Verfügung gestellt. Da dieses Datenformat keine Informationen zur erlaubten Höchstgeschwindigkeit oder Spurenanzahl einer Strasse bietet, diese Werte allerdings auch nützlich sein könnten, müsste man sie manuell zu den Daten hinzufügen oder heuristisch ermitteln. Die Anzahl der Spuren einer Strasse ist beispielsweise nicht im Datenformat als Information vorhanden, ist aber für die Bewertung der Verkehrsdichte wichtig, da dieser Wert auf einer dreispurigen Autobahn wesentlich höher sein darf als auf einer einspurigen Landstrasse, ohne den Verkehrsfluss zu beeinträchtigen.

Da die Länge der Kanten in manchen Fällen mehrere Kilometer beträgt, gibt es Probleme im Fall, dass ein Stau auftritt, dieser sich aber in den variablen Kenngrößen wie Verkehrsdichte oder Durchschnittsgeschwindigkeit erst widerspiegelt, wenn sich der Stau auf die Kante ausgebreitet hat. Hierfür kann die Berechnung der variablen Kenngrößen bei langen Straßenkanten in Segmente aufgeteilt werden. Der Wert der gesamten Straßenkante wird in diesem Fall auf das Minimum der Durchschnittsgeschwindigkeiten bzw. auf das Maximum der Verkehrsdichten auf den Segmenten der Straßenkante gesetzt. Die Kenngrößen geben so auch lokale Beeinträchtigungen des Verkehrsflusses wieder. Problematisch ist diese Lösung allerdings im Zusammenhang mit Tempolimits, da die Werte hierfür wie bereits erwähnt nicht zur Verfügung stehen und nicht differenziert werden kann, ob es sich bei einer streckenweisen niedrigeren Durchschnittsgeschwindigkeit der Verkehrsteilnehmer um einen Verkehrsstauung oder lediglich um das ordnungsgemäße einhalten der erlaubten Höchstgeschwindigkeit handelt.

Um dem Navigator eine bessere Bewertung der Kenngröße Durchschnittsgeschwindigkeit zu ermöglichen könnten die Fahrzeugsubsysteme einen für jedes Fahrzeug voreingestellten Wert der Höchstgeschwindigkeit übermitteln. Hiermit könnte der Navigator erkennen, ob eine niedrige Durchschnittsgeschwindigkeit durch eine Verkehrsbehinderung oder durch einen erhöhten Anteil an Lastkraftwagen und Fahrzeugen mit limitierter Höchsgeschwindigkeit verursacht wird.

2.5.3 Zusätzliche Kenngrößen

Es ist nicht anzunehmen, dass in einem sich im Einsatz befindlichen Verkehrsroutingsystem kurz nach der Installation durch die in den Fahrzeugen eingebauten Subsysteme ausreichende Informationen

über den Verkehrsfluss auf den Straßen bereitgestellt werden können. Bis eine gewisse Marktdurchdringung erreicht ist, werden zusätzliche Informationsquellen benötigt, damit die Bewertungsfunktion die Verkehrssituation auf den vom System überwachten Straßen adäquat abbilden kann. Wie groß der Anteil der Fahrzeuge, die mit zum BeeJama-Verkehrsrouting kompatiblen Geräten ausgerüstet sind, am Verkehr sein muß, um alleine durch die Kenngrößen, die von diesen Fahrzeugen bereitgestellt werden, das Verkehrsrouting ohne Störungen und wie geplant durchführen zu können, ist Gegenstand weitergehender Arbeiten. Weiterhin ist zu untersuchen, wie gut das System funktioniert, wenn ein Teil der Fahrer die von BeeJamA vorgeschlagene Route nicht befolgt.
Das oberste Ziel der Simulation und der Tests der Implementierung von BeeJamA war der Nachweis, dass der Algorithmus den Verkehrsfluss verbessern und Verkehrsstauungen verhindern kann. Infolgedessen wurde bei den im folgenden vorgestellten Bewertungsfunktionen ein Szenario angenommen, in dem alle Fahrzeuge auf den vom System verwalteten Strecken mit kompatiblen Fahrzeugsubsystemen ausgerüstet sind und die Fahrer die Vorschläge ohne Ausnahme befolgen.

Als zusätzliche Informationsquelle für die Bewertungsfunktion könnte das TMC bzw. TMCpro-System dienen, ein digitaler Radiodienst, der Informationen über Verkehrsbeeinträchtigungen über das RDS-Signal herkömmlicher Radiosender überträgt. Die TMC-Meldungen enthalten allerdings lediglich einen Ereigniscode, der eine grobe Beschreibung (z.B. „zähfließender Verkehr“ oder „Stau“) der Verkehrssituation auf einem Straßenabschnitt repräsentiert.

Zur Messung des Verkehrsaufkommens sind an vielen Autobahnen und sonstigen Strassen hauptsächlich Ultraschalldetektoren und Induktionsschleifen zum Einsatz. Diese beiden Messeinrichtungen können feststellen ob und wie lange sich ein Fahrzeug im Erfassungsbereich befindet, woraus sich die Geschwindigkeit der Fahrzeuge berechnet werden kann. Aus dem Bruttoabstand der Fahrzeuge kann die Verkehrsdichte, wie von Kienzle beschrieben [12], bestimmt werden.

Abbildung 2.4: Kenngrößen

	statisch	dynamisch
gegeben	• Kantenlänge • Straßentyp • „road level“	• Geschwindigkeit (der am System teilnehmenden Fahrzeuge • Position (der am System teilnehmenden Fahrzeuge)
heuristisch bestimmt(statisch) / berechnet (dynamisch)	• erlaubte Höchstgeschwindigkeit • Anzahl Spuren	• Durchschnittsgeschwindigkeit • Verkehrsdichte

2.6 BeeJamA Dichte-/Geschwindigkeits-Bewertungsfunktion

2.6.1 Einleitung

In der im folgenden beschriebenen ersten Strategie zur Berechnung der Kantenqualitäten sollen Faktoren, die zu einer Verkehrsstauung führen können, miteinbezogen werden. In diesem Zusammenhang wird die Verkehrsdichte als zentraler Einfluss betrachtet und weitere Faktoren wie die Durchschnittsgeschwindigkeit der Fahrzeuge und die Straßeneigenschaften berücksichtigt.

2.6.2 Beschreibung

Wie bereits erwähnt hat die Verkehrsdichte einen grossen Einfluss auf die Entstehung von Verkehrsstauungen. Die Straßenkanten des vom Routingsystem verwalteten Strassennetzes werden in Segmente aufgeteilt, wobei die *globale Dichte* ρ_{global} die Verkehrsdichte auf einer Kante und die *lokale Dichte* ρ_{lokal} die Verkehrsdichte auf einem einzelnen Segment der Kante beschreiben. Die lokale Dichte wird verwendet, um Verkehrsstauungen über die erhöhte Verkehrsdichte in einem Segment zu erkennen und die Qualität der ganzen Kante herabzusetzen. Die lokale Dichte berechnet sich wie folgt:

$$\rho_{lokal} = \frac{f}{l_s}$$

wobei f die Anzahl von Fahrzeugen auf dem Straßensegment und l_s die Länge des Segments angibt. Der Parameter β dient als Exponent, der zu Testzwecken den Einfluss der lokalen Dichte auf die Kantenqualität erhöhen bzw. absenken kann. Die lokale Dichte ist umgekehrt proportional zur Qualität einer Kante. Daher lässt sich folgende Formel für die lokale Qualität einer Kante ableiten:

$$Q_{lokal} = \frac{(\frac{\overline{v}}{v_{max}})^{\alpha}}{\rho_{lokal}^{\beta}}$$

Eine weitere Kenngröße, die bei dieser ersten Strategie zur Berechnung der Kantenqualtitäten verwendet wird, ist die Durchschnittsgeschwindigkeit der Fahrzeuge in einem Straßensegment. In der obigen Gleichung stellt die Variable v_max die erlaubte Höchstgeschwindigkeit auf dem betrachteten Straßensegment dar. Hierbei wird ein ideales Modell der Strassendaten angenommen, in dem für die verwalteten Strassen die Höchstgeschwindigkeit in den Daten vorhanden ist. Für eine spätere Verwendung mit erweiterten Daten sollten die Segmente unbedingt so gewählt werden, dass in jedem Straßensegment nur genau eine erlaubte Höchstgeschwindigkeit gilt. Die Variable $\overline{v}$ repräsentiert die durchschnittliche Geschwindigkeit der Fahrzeuge im betrachteten Straßensegment.
Der Quotient $\frac{\overline{v}}{v_{max}}$ beschreibt damit also die Ausnutzung der Höchstgeschwindigkeit durch die Fahrzeuge auf dem Straßensegment. Durch die Verwendung dieses Werts sind keine Unterscheidungen von verschiedenen Straßentypen wie Autobahn oder Stadtstraßen hinsichtlich der gefahrenen Geschwindigkeiten bei der Bewertungsfunktion nötig.

Im Fall einer zeitweilig von keinem Fahrzeug befahrenen Strasse wird also Durchschnittsgeschwindigkeit $\overline{v}$ die Maximalgeschwindigkeit auf dieser Strecke v_{max} eingesetzt. Der berechnete Wert soll widerspiegeln, dass die Geschwindigkeit, die auf dieser Strasse zu erwarten ist, der erlaubten Maxi-

malgeschwindigkeit entspricht.

Für jede Straßenkante gibt es nun eine Anzahl von lokalen Qualitäten, wobei eine schlechte lokale Qualität eines Segments der Kante die Qualität der gesamten Kante beinflussen soll, da eine Verkehrsstauung auf einem Segment den Verkehr auf der Strasse insgesamt blockiert. Daher wird für die Bewertung einer Kante das Minimum aller lokalen Qualitäten der Segmente dieser Kante verwendet:

$$Q_{Edge} = min\,\{Q_i\}$$

2.7 BeeJamA Statistik-Bewertungsfunktion

2.7.1 Einführung

In der zweiten Strategie zur Berechnung der BeeJamA-Bewertungsfunktion werden statistische Werte zu Verkehrsaufkommen und Verkehrsfluss zur Berechnung der Qualität der Straßenkanten verwendet. Die Grundlage bildet dabei [15]. Das Paper bietet Informationen zu den Zusammenhängen des Aufkommens von Verkehrsstaus und der Verkehrsdichte auf verschiedenen Straßentypen.

2.7.2 Beschreibung

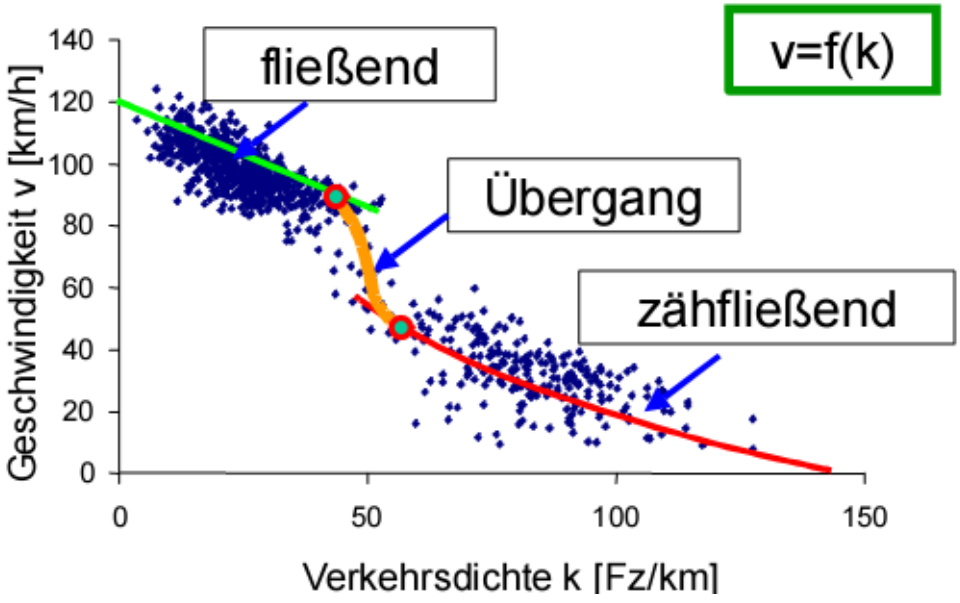
Zweispurige Autobahnen

Nach [15] zeigt Abbildung 2.5 die Bildung von Verkehrsstaus abhängig von der Verkehrsdichte auf einer zweispurigen Autobahn:

Der Ansatz von [15] ist es, die Abhängigkeit zwischen Verkehrsdichte und Durchschnittsgeschwindigkeit der Fahrzeuge in verschiedene Zustände einzuteilen. Gemäß der Darstellung ist ein fließender Verkehr mit Durchschnittsgeschwindigkeiten über 80 km/h nur möglich, wenn die Verkehrsdichte unter 45 Fahrzeugen pro Kilometer liegt. Zwischen 45 und 55 Fahrzeugen pro Kilometer gibt es den sogenannten Übergangsbereich, in dem die Durchschnittsgeschwindigkeit stark abfällt. Im Übergangsbereich liegen laut der statistischen Daten aus [15] die maximalen Verkehrsstärken. Für eine zweispurige Autobahn liegt die Kapazität bei etwa 4000 Fahrzeugen pro Stunde, wobei die Durchschnittsgeschwindigkeit dann bei ungefähr 80 km/h liegt. Der Ansatz, beim Verkehrsrouting eine Strasse möglichst immer bei genügendem Verkehr auf dem Straßennetz bis zu ihrer maximalen Verkehrstärke auszunutzen wurde verworfen, da der Übergangsbereich zu instabil ist. Bei über 55 Fahrzeugen pro Kilometer kommt es zu zähfließendem Verkehr und Verkehrsstaungen.

Die sogenannte Statistik-Bewertungsfunktion soll nun diese Erkenntnisse umsetzen. Daher sollte eine Straßenkante mit einer Verkehrsdichte unter 45 Fahrzeugen pro Kilometer eine sehr gute Qualität ausgewiesen bekommen. Über diesem Wert von 45 Fahrzeugen pro Kilometer sollte der Qualitätswert der Kante stark abnehmen, um die Kante zu entlasten und so Verkehrsstaus zu vermeiden. Analog zu Abbildung 2.5 wird die Qualitätsfunktion in drei Teile entsprechend der Verkehrszustände aufgeteilt.

Abbildung 2.5: Durchschnittsgeschwindigkeit auf zweispuriger Autobahn



Zwei Parameter α und β repräsentieren den Anfang und das Ende Übergangsbereich, um die Art der Straße zu charakterisieren. Im Fall von zweispurigen Autobahnen ist $\alpha = 45$ and $\beta = 55$. Die Parameter A und B können frei gewählt werden, um die Qualitäten in den Verkehrszuständen anzupassen. A steht dabei für die Qualität in der Schnittstelle zwischen fließendem Verkehr und Übergangsbereich. Im Folgenden wird $A = 0,8$ verwendet, da die Qualität im Bereich des fließenden Verkehrs mit Werten zwischen $0,8$ und $1,0$ als sehr gut betrachtet wird. Der zweite Parameter wird hier mit $B = 0,3$ verwendet, da die Qualitäten zwischen $0,3$ und $0,8$ einen teilweise gestörten Verkehrsfluss kennzeichnen sollen, während Qualitäten unter $0,3$ darauf hinweisen, dass auf der Straßenkante zähfließender Verkehr herrscht und diese Strecke vermieden werden sollte. Da Autobahnen auf Grund der höheren Kapazität und höheren erlaubten Höchsgeschwindigkeiten anderen Straßen immer bevorzugt werden sollten, wird ein Parameter q_{max} definiert, der die maximale Qualität einer Kante beschreibt. Der Wert der maximalen Qualität wird dann für Autobahnen höher angesetzt als für Bundesstraßen und dieser wiederum höher als der Wert für innerstädtische Straßen. Für eine zweispurige Autobahn wird $q_{max} = 1,0$ angenommen.
Die Qualitätsfunktion ist wie folgt definiert:

a) $0 < \rho < \alpha$
Der erste Fall tritt ein, wenn der Wert der Verkehrsdichte niedriger als α liegt, also im Bereich des fließenden Verkehrs. In diesem Fall soll der Kante eine sehr gute Qualität ausgewiesen werden, da noch mehr Fahrzeuge diese Strecke befahren können, ohne den Verkehr erheblich negativ zu beeinflussen. Die Qualität der Kante berechnet sich dann wie folgt:

$$Q_{Edge} = \frac{(A - q_{max})}{\alpha^2}\rho^2 + q_{max}$$

b) $\alpha < \rho < \beta$

Wenn die Verkehrsdichte größer als der Wert von α aber noch unterhalb des Werts von β liegt, tritt der zweite Fall ein. Hierbei liegt der Wert der Verkehrsdichte im Übergangsbereich zu zähfließendem Verkehr. Eine geringe Erhöhung der Verkehrsdichte führt schon zu einer beachtlichen Beeinträchtigung des Verkehrsflusses.

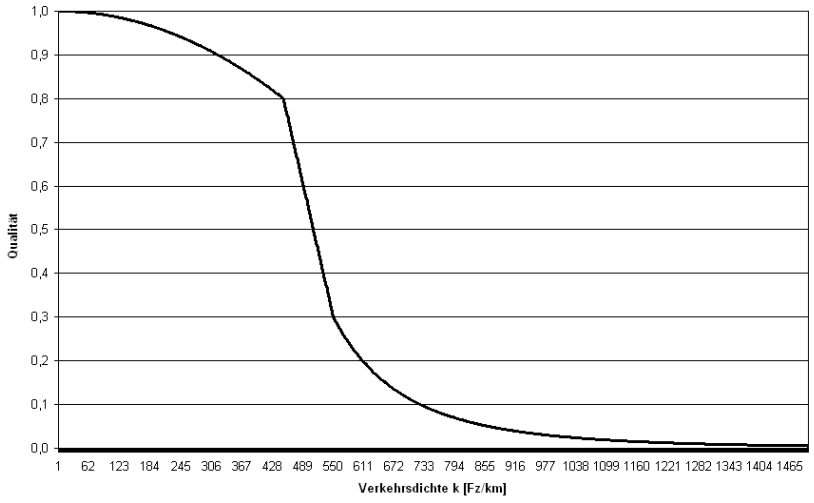
$$Q_{Edge} = \frac{(\rho-\alpha)(B-A)}{\beta-\alpha} + A$$

c) $\rho >= \alpha$
Der letzte Fall tritt ein wenn die Anzahl an Fahrzeugen pro Kilometer grösser als β und somit lediglich zähfließender Verkehr auf der zugehörigen Straßenkante möglich ist. Eine Straßenkante, bei dieser Fall eintritt sollte bei Routingentscheidungen vermieden werden und infolgedessen eine sehr niedrige Qualität ausgewiesen bekommen.

$$Q_{Edge} = \frac{\beta^4}{\rho^4} B$$

Die endgültige Funktion für eine zweispurige Autobahn ist in der folgenden Abbildung gezeigt. Die zugehörigen Parameter sind:
$\alpha = 45$
$\beta = 55$
$A = 0.8$
$B = 0.3$
$g_{max} = 1$

Abbildung 2.6: Qualitätsfunktion von zweispurigen Autobahnen



Einspurige Bundesstraßen und dreispurige Autobahnen

Analog zur Berechnung der Kantenqualitäten für zweispurige Autobahnen werden für dreispurige Autobahnen und einspurige Bundesstraßen statistische Werte zu Grunde gelegt. Die vorgestellte Qualitätsfunktion wird weiterhin verwendet, wobei die Parameter α, β und q_{max} an die Art der Strasse angepasst werden. Die statistischen Werte sind wiederum [15] entnommen.

Abbildung 2.7: Durchschnittsgeschwindigkeit auf einspurigen Bundesstraßen

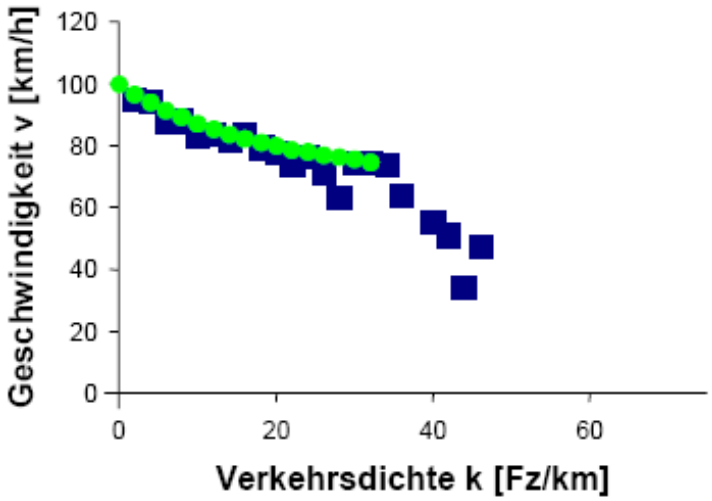


Abbildung 2.7 zeigt die Durchschnittsgeschwindigkeit auf einer einspurigen Landstrasse in Abhängigkeit von der Verkehrsdichte. In Übereinstimmung mit diesen statistischen Größen werden die Werte zur Einteilung der Verkehrszustände $\alpha = 35$ und $\beta = 40$ gesetzt. Da Autobahnen normalerweise die schnellere und komfortablere Alternative zu Bundesstraßen sind, sollte der maximale Qualitätswert einer einspurigen Bundesstrasse niedriger sein als der Qualitätswert einer zweispurigen Autobahn mit fließendem Verkehr. So sollte bei der Routingentscheidung eine Autobahn mit ausreichend flüssigem Verkehr mit einer höheren Wahrscheinlichkeit gewählt werden als eine leere Bundesstrasse.
Wie oben abgebildet ist es durch die dritte Fahrspur einer dreispurigen Autobahn möglich, eine höhere Verkehrsdichte mit mehr Fahrzeugen pro Kilometer als bei zweispurigen Autobahnen zu erlauben, ohne in den Bereich des zähfließenden Verkehrs zu kommen. Selbst bei einer Verkehrsdichte von 50 Fahrzeugen pro Kilometer liegt die Durchschnittsgeschwindigkeit noch bei etwa 80 km/h. Für die Qualitätsfunktion für dreispurige Autobahnen werden die Parameter $\alpha = 50$ und $\beta = 70$ gesetzt. Die Graphen der Qualitätsfunktionen aller drei Typen von Straßen werden in Abbildung 2.9 gezeigt.
Für zweispurige und dreispurige Autobahnen gilt wie vorher $A = 0,8$ und $B = 0,3$. Die maximale Qualität auf Bundesstraßen ist 0,8 und die Parameter werden mit $A = 0,5$ und $B = 0,3$ gesetzt für die Qualitätsfunktion bei Bundesstraßen gesetzt.

Abbildung 2.8: Durchschnittsgeschwindigkeit auf dreispurigen Autobahnen

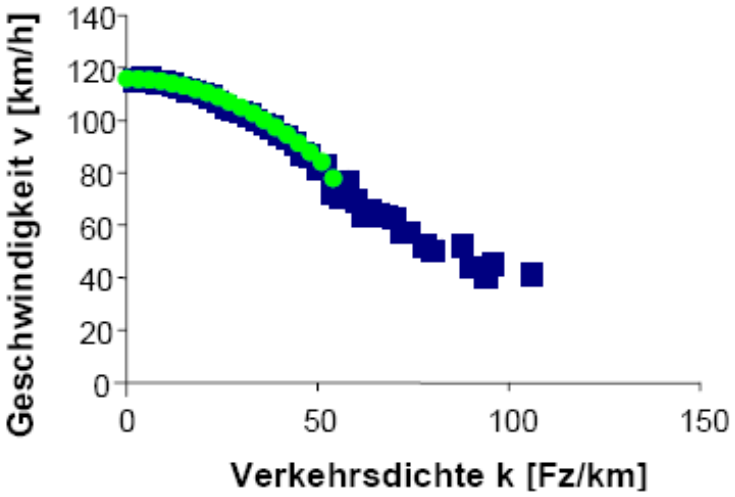
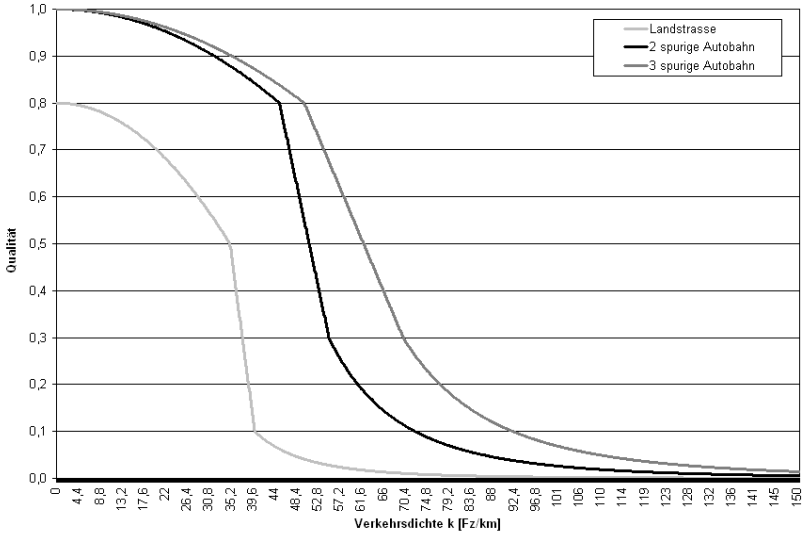


Abbildung 2.9: Qualitätsfunktionen



Kapitel 3

Simulationsmodell

3.1 Datenbasis

Damit ein realitätsnahes Verkehrsgeschehen mithilfe des Simulators überhaupt erst simuliert werden kann, ist es nötig, das reale Straßennetz als vektorielle Daten vorliegen zu haben, um daraus einen Graphen bilden zu können. Allerdings gibt es nur einen einzigen, kommerziellen Anbieter, der solche Daten mit weltweiter Abdeckung zur Verfügung stellen kann: die Firma *AND Automotive Navigation Data* aus den Niederlanden. Aus diesem Grund und aufgrund der Tatsache, dass das Format der *AND Global Road Data* relativ einfach zu handhaben ist, hat sich die PG dazu entschlossen, ein zu dem der Firma AND kompatibles Format für den Simulator einzusetzen und eine Reihe eigener Straßennetze zu erstellen.

3.1.1 Straßennetze

Ein Straßennetz wird mithilfe eines gerichteten Graphs dargestellt. Es besteht also immer aus einer Knotendatei (Endung .vrt), einer Kantendatei (Endung .sgm) und einer optionalen Intermediatedatei (Endung .int), die der Unterteilung von Kanten zur schöneren grafischen Darstellung dient. Dabei ist jede dieser drei Dateien eine einfache Textdatei: jede Zeile enthält einen einzelnen Datensatz, dessen Felder durch Kommata voneinander getrennt sind. Die Felder haben folgende Bedeutungen:

Knotendatei (Endung .vrt)

Beispiel eines Datensatzes:
`982924,51.493,7.3679,0,49,4,0,,`

Feld 0 Eindeutige Knoten-ID, auf die sich die Kanten- und Intermediatedateien beziehen.

Feld 1 Geographische Breite des Knotens in Grad als Dezimalzahl mit dem Punkt als Dezimaltrennzeichen. Nördliche Breiten werden als positive, südliche als negative Zahl dargestellt.

Feld 2 Geographische Länge des Knotens in Grad als Dezimalzahl mit dem Punkt als Dezimaltrennzeichen. Westliche Längen werden als negative, östliche als positive Zahl dargestellt.

Felder 3 bis 8 Werden vom Simulator nicht verwendet, können aber aus Kompatibilitätsgründen zum AND-Format vorhanden sein.

Kantendatei (Endung .sgm)

Beispiel eines Datensatzes:
`983672,983654,1,2,1,49,1,0,,`

Feld 0 ID des Start-Knotens (siehe Knotendatei, Feld 0).

Feld 1 ID des End-Knotens (siehe Knotendatei, Feld 0).

Feld 2 Länge der Kante in Hektometern.

Feld 3 Richtungsangabe für die Kante. Eine `0` bedeutet, dass die Straße in beide Richtungen, eine `1`, dass die Straße nur vom Start- zum Endknoten und eine `2`, dass die Straße nur vom End- zum Startknoten befahren werden kann.

Feld 4 Straßentyp (siehe unten)

Felder 5 bis 10 Werden vom Simulator nicht verwendet, können aber aus Kompatibilitätsgründen zum AND-Format vorhanden sein.

Dabei existieren foldende Straßentypen:

1 Autobahn

2 autobahnähnliche Bundesstraße

3 Bundesstraße

4 Regionalstraße

5 Stadtstraße

6 andere Straßen

Intermediatedatei (Endung .int)

Die Intermediatedatei dient dazu, einzelne Kanten nochmals in kleinere Abschnitte zu unterteilen. Dieses führt zu einer schöneren graphischen Darstellung von stark gekrümmten Kanten, hat auf die eigentliche Verkehrssimulation aber ansonsten keine Auswirkungen.

Beispiel eines Datensatzes:
`983014,983592,51.4923,7.3592,51.4924,7.3601,51.4925,7.3607`

Feld 0 ID des Start-Knotens (siehe Knotendatei, Feld 0).

Feld 1 ID des End-Knotens (siehe Knotendatei, Feld 0).

Felder 2,4,6... Geographische Breite einer Unterteilung in Grad als Dezimalzahl mit dem Punkt als Dezimaltrennzeichen. Nördliche Breiten werden als positive, südliche als negative Zahl dargestellt.

Felder 3,5,7... Geographische Länge einer Unterteilung in Grad als Dezimalzahl mit dem Punkt als Dezimaltrennzeichen. Westliche Längen werden als negative, östliche als positive Zahl dargestellt.

Die Felder ab Feld 2 enthalten also eine Liste von Intermediates. Die Anzahl der Intermediates bzw. ihrer Koordiantenpaare ist beliebig.

3.1.2 Probleme und Grenzen

Fehlende Informationen

Durch die in Abschnitt 3.1.1 beschriebenen Dateien wird ein gerichteter Graph dargestellt, dessen Kanten durch Länge und Typ gewichtet sind. Dieses reicht für eine realitätsnahe Simulation des Verkehrsgeschehens allerdings nicht aus. Es müssen weitere Daten anhand von Länge, Typ und Intermediates heuristisch ermittelt werden, z.B.:

- die Höchstgeschwindigkeit einer Kante
- die Anzahl der Fahrspuren einer Kante
- Kanten, die Autobahnauffahrten sind, müssen als solche markiert werden

Diese Aufgabe übernehmen die Operatoren aus der *Data and Heuristic Component (DHC)*, siehe Abschnitt 4.12 .

Problematische Netzsituationen

Bei der Erstellung der Straßennetze und der dazugehörigen Dateien ist darauf zu achten, dass es einige Sonderfälle innerhalb der Straßennetze geben kann, die zwar theoretisch möglich, aber nicht sehr realitätsnah sind und für ein ordentliches Funktionieren der Routingalgorithmen vermieden werden sollten. Es könnte zum Beispiel sein, dass zwar in Teile des Straßennetzes hinein, aufgrund von Einbahnstraßen aber nicht wieder heraus gefahren werden könnte. Auch sind Gebiete möglich, zu denen keine Straße führt, sodass diese Gebiete überhaupt nicht zu erreichen sind.

All diese Situationen können dann auftreten, wenn schon vorhandene Datensätze räumlich eingeschränkt (zum Beispiel bei der Beschränkung auf eine einzelne Stadt) oder anderweitig gefiltert werden (zum Beispiel beim Herausfiltern von Autobahnen oder Stadtstraßen). Es ist also immer Voraussetzung, dass der zu bildende Graph *stark zusammenhängend* ist, das heißt, von jedem beliebigen Knoten muss jeder andere Knoten des Graphs erreichbar sein. Entweder ist schon beim Erstellen der Straßennetze darauf zu achten, dass diese Bedingung erfüllt ist oder aber der Graph muss im Nachhinein geeignet angepasst werden.

3.2 Verkehrsmodell

Um Routing-Algorithmen simulieren zu können muss ein möglichst realitätsnahes Verkehrsmodell existieren, welches als Grundlage dient. In diesem Abschnitt soll das im Simulator benutzte Verkehrsmodell vorgestellt werden und es wird einen kurzen Einblick geben, warum dieses Modell bzw. diese

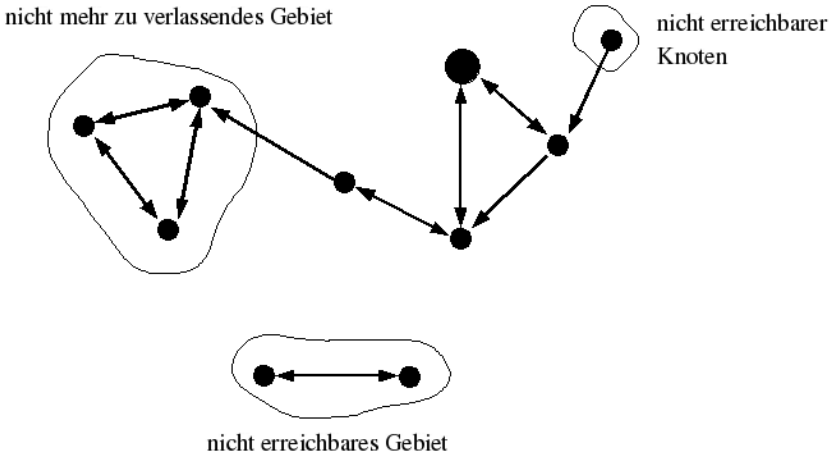


Abbildung 3.1: Problematische Netzsituationen

Modelle ausgewählt wurden. Verkehr und dessen Beeinflussung ist ein komplexes Thema, was man anhand der Vielzahl von Modellen aus [18] erkennen kann. Die Modelle beschreiben im Großen und Ganzen zwei Arten von Verkehr, einmal den Verkehr bzw. die Bewegung des Verkehrs auf einer Spur und zum anderen die Verkehrsbewegung auf mehrspurigen Straßen. Um mehrspurigen Verkehr zu simulieren benötigt man also zwei Modelle, eines für die Fortbewegung auf der Spur auf der sich der Verkehrsteilnehmer befindet und eines um zwischen verschiedenen benachbarten Spuren zu wechseln. Die Einspur-Modelle regeln den Verkehrsfluss auf einer Spur unter Berücksichtigung von Bremsen, Beschleunigen und Trödeln. Im Grunde wurde das VDR Modell (Velocity Dependant Randomization, Abschnitt 3.2.2) nach Nagel-Schreckenberg verwendet und durch einige Details verfeinert, um den Verkehr realitätsgetreuer abbilden zu können. Das VDR Modell wurde erweitert indem Bremslichter benutzt werden, um das Verhalten des rückwärtigen Verkehrs in Bremssituationen zu verbessern. Durch diese Erweiterungen wird aus dem VDR Modell das Bremslicht Modell nach Knospe (Abschnitt 3.2.2), welches im Simulator umgesetzt werden soll, um das Einspurverhalten der Fahrzeuge zu simulieren.
Um nun das Mehrspurverhalten zu simulieren, musste eine Entscheidung getroffen werden, welches der verschiedenen Modelle in der Simulation genutzt werden soll. Die Auswahl fiel auf das Mehrspurmodell nach Nagel (Abschnitt 3.2.3), da es eine gute Grundlage für die Simulation von Verkehr auf mehreren Spuren Darstellt. Es wird zwischen symmetrischem[1] (Amerika) und asymmetrischen[2] (Europa) Spurwechseln unterschieden und es gilt das Rechtsfahrgebot, wie man es aus Deutschland kennt. Nähere Informationen zu den Modellen folgen in Abschnitt (3.2.2).

[1] Bei symmetrischen Spurwechselregeln wird ein Wechsel von Links nach Rechts und ein Spurwechseln von Rechts nach Links gleich behandelt. Es ist weiterhin nicht vorgeschrieben nach Möglichkeit die rechteste Spur zu benutzen, sich also an das Rechtsfahrgebot zu halten.

[2] Bei asymmetrischen Wechselregeln gilt das Rechtsfahrgebot, dementsprechend werden auch Spurwechsel von Rechts nach Links und entgegensetzt unterschiedlich behandelt.

3.2.1 Zellularautomat

Grundlage des Verkehrsmodells bildet ein Zellularautomat welcher das Verkehrsnetz in Spuren mit einer festen Anzahl von Zellen unterteilt. Fahrzeuge belegen dabei eine Menge von aufeinander folgenden Zellen und blockieren diese, so dass kein anderes Fahrzeug sie nutzen kann.
Formal ist ein Zellularautomat ein räumlich und zeitlich diskretes System, dass durch folgende Größen vollständig beschrieben wird.

- Raum der Zellen R
- Nachbarschaft N
- Menge der Zustände Q
- Zustandsüberführungsfunktion δ

Der Raum R ist in der Regel ein- oder zwei-dimensional. Für die Simulation eines Straßennetzes ist R zweidimensional. Die Nachbarschaft N beschreibt die Zellen, die sich in der Nähe einer bestimmten Zelle befinden und einen Einfluss auf ihren Zustand haben können. Bei der Verkehrssimulation wird über die Nachbarschaft festgelegt, bis auf welche Entfernungen andere Fahrzeuge, Verkehrsbeeinflussungsanlagen[3] oder sonstige Einflussfaktoren relevant für das Fahrverhalten eines Fahrzeuges sind. Die Menge der Zustände beschreibt alle möglichen Belegungen der einzelnen Zellen. Die Zustandsüberführungsfunktion wird durch die im folgenden beschriebenen Regeln definiert.

3.2.2 Einspurmodell

Im folgenden sollen die Grundlegenden Überlegungen für die Umsetzung von Einspurverkehr erläutert werden.

Das Nagel-Schreckenberg Modell Das Nagel-Schreckenberg-Modell (im folgenden als NaSch-Modell bezeichnet) stellt einen einfachen Regelsatz dar, um das Verhalten von Fahrzeugen auf einer einspurigen Straße zu simulieren. Der Zellraum des Zellularautomaten ist in diesem Fall eindimensional. Um eine Fahrspur auf einen diskreten Zellraum abbilden zu können, wird sie in $7,5\ m$ lange Teilstücke unterteilt. Diese $7,5\ m$ sollen dem Platz entsprechen, den ein stehendes Auto inklusive der Abstände zum Vorder- und Hintermann benötigt. Der Zustandsraum stellt sich dadurch dar, dass jede Zelle zu einem Zeitpunkt von einem Fahrzeug besetzt sein kann oder nicht. Fahrzeuge werden durch ihre Geschwindigkeit $v = 1, 2, ..., v_{max}$ charakterisiert. Dabei entspricht v der Anzahl an Zellen die das Auto innerhalb eines diskreten Zeitschrittes (Step) überwindet.
Im NaSch-Modell werden vier Regeln definiert, welche das Verhalten der Fahrzeuge festlegen und die Überführungsfunktion des Automaten darstellen.

1. **Beschleuigung**
   Jedes Fahrzeug n das zum aktuellen Zeitpunkt t noch nicht die Maximalgeschwindigkeit v_{max} erreicht hat, beschleunigt um eine Einheit.
   $WENN(v_{n,t} < v_{max})\{v_n(t+1) := v_n(t) + 1\}$

[3] Technische Einrichtungen an Straßen , zur Beeinflussung des Verkehrsflusses (bspw. Zuflussregelungen an Autobahnauffahrten, dynamische Geschwindigkeitsanzeigen)

2. **Sicherheitsabstand**
   Befinden sich vor einem Fahrzeug n d_n freie Zellen und ist die Geschwindigkeit des Fahrzeugs größer als d_n, wird die Geschwindigkeit auf d_n reduziert.
   $v_{n,t+1} := min\{v_n(t+1), d_n\}$

3. **Trödeln**
   Die Geschwindigkeit wird zufällig mit einer Wahrscheinlichkeit p um eine Einheit vermindert, wenn sie nicht bereits null war.
   $WENN(p\&v_n(t) < v_{max})\{v_n(t+1) := v_{n,t+1} - 1\}$

4. **Fahren**
   Nachdem in den Schritten 1 bis 3 die neuen Geschwindigkeiten v_n für jedes Fahrzeug n ermittelt wurden, werden die Positionen x_n entsprechend aktualisiert.
   $x_n(t+1) := x_n(t) + v_n(t+1)$

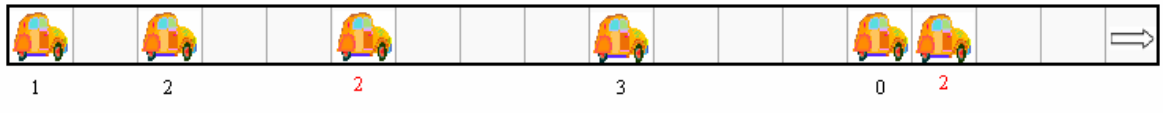
Der erste Schritt simuliert das Bestreben eines jeden Fahrzeugführers die erlaubte Maximalgeschwindigkeit zu erreichen. Im zweiten Schritt wird die Interaktion zwischen dem betrachteten Fahrzeug und dessen nächsten Nachbarn ausgewertet, hierbei ist die Nachbarschaft in der andere Fahrzeuge Einfluss auf das Fahrverhalten nehmen abhängig von der Geschwindigkeit. Der dritte Berechnungsschritt stellt das nicht ideale Verhalten eines menschlichen Fahrzeugführers nach, welcher nicht immer den Idealvorstellungen entsprechend beschleunigt oder abbremst.
Bei der Abarbeitung dieser Schritte ist es wichtig, dass während der Berechnungen in den Schritten 1 bis 3 keine Zustandsänderungen am Simulator vorgenommen werden. Es darf kein Fahrzeug umgesetzt werden, da es eine elementare Bedingung ist, dass alle Autos den selben Zustand als Grundlage für ihre Berechnungen benutzen. Erst durch das Umsetzten der Fahrzeuge im vierten Schritt darf der Zustand des Zellularautomaten geändert werden. Dieses Verkehrsmodell wird als minimales Verkehrsmodell bezeichnet, da das Weglassen eines Schrittes kein realistisches Verkehrsverhalten mehr ermöglicht.
Am folgenden Beispiel soll der Ablauf dieser vier Berechnungschritte kurz erläutert werden. In Abbildung 3.2(a) ist der Zustand des Automaten vor den Berechnungschritten zu sehen. Die Zahlen unter den Autos stellen die Geschwindigkeit $v_{n,t}$ für jedes Fahrzeug zum Zeitpunkt t dar. In Abbildung 3.2(b) ist die Geschwindigkeit $v_{n,t+1}$ nach dem Beschleunigungsschritt dargestellt. Alle Fahrzeuge die noch nicht ihre Endgeschwindigkeit (in diesem Beispiel 5) erreicht haben, haben ihre Geschwindigkeit um eins erhöht. In Abbildung 3.2(c) ist zu sehen, dass alle Fahrzeuge ihre Geschwindigkeit auf die Anzahl freier Zellen vor sich reduziert haben. Im vierten Bild ist dann zu sehen, dass das dritte und fünfte Fahrzeug ihre Geschwindigkeit um eins veringert haben, was auf das Trödeln zurück zuführen ist. Letztendlich werden die Fahrzeuge in Abbildung 3.2(e) um die ermittelte Geschwindigkeit fortbewegt.

Das VDR Modell Das VDR Modell (Velocity Dependant Randomization) ist eine Erweiterung des NaSch Modells. Zusätzlich zu den vier erwähnten Schritten wird ein weiterer Schritt vor den anderen Berechnungsschritten eingefügt. In diesem Schritt werden die im folgenden beschriebenen Wahrscheinlichkeiten ausgewürfelt. Dabei ist p_n die Wahrscheinlichkeit für das Trödeln eines Fahrzeugs n. Diese Trödelwahrscheinlichkeit wird entsprechend der Situation, in der sich das Fahrzeug befindet, berechnet Falls das Fahrzeug n im Stau ($v_n = 0$) steht, gilt Wahrscheinlichkeit p_0. Die Wahrscheinlichkeit p_b wird verwendet falls das vorausfahrende Fahrzeug sein Bremslicht aktiviert hat. In allen anderen Fällen wird die Wahrschienlichkeit p verwendet. Für die individuelle Trödelwahrscheinlich-

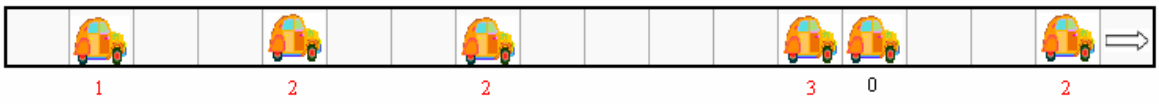
(a) Situation zum Zeitpunkt t

(b) Beschleunigen

(c) Abbremsen



(d) Trödeln



(e) Situation zum Zeitpunkt t+1

keit eines jeden Fahrzeugs n gilt dann:

- **Wahrscheinlichkeit berechnen**

$$p_n = \begin{cases} p_b & WENN(b_{n+1} == 1\, UND\, t_h < t_s) \\ p_0 & WENN(v_n == 0) \\ p & sonst \end{cases}$$

Damit ist Schritt 3 nicht mehr von der allgemeinen Trödelwahrscheinlichkeit p, sondern von der in Schritt 0 berechneten Wahrscheinlichkeit p_n anhängig.

- **Trödeln**
  Die Geschwindigkeit wird zufällig mit einer Wahrscheinlichkeit p um eine Einheit vermindert, wenn sie nicht bereits null war.
  $WENN\ (p_n\ \&\ v_n(t) < v_{max})\ \{v_n(t+1) := v_n(t+1) - 1\}$

Wenn ein Fahrzeug steht, also die aktuelle Geschwindigkeit null ist, nimmt die Trödelwahrscheinlichkeit demnach einen anderen Wert an als bei einem fahrenden Auto. Im Regelfall wird eine höhere Trödelwahrscheinlichkeit für stehende Autos verwendet. Durch den neuen Trödelparameter ergibt sich ein vollkommen anderer Verkehrsfluss. Während beim NaSch Modell oft kleine Staus entstehen die sich schnell wieder auflösen, kommt es durch das VDR-Modell zu weniger aber größeren Staus. Das ist dadurch zu begründen, dass beim NaSch-Modell die Fahrzeuge am Stauanfang eine freie Fahrbahn haben und maximal Beschleunigen können. Der Stau kann sich also schnell lösen. Beim VDR Modell hingegen wird mit höherer Wahrscheinlichkeit getrödelt. Der Stau löst sich folglich nur langsam auf.

Erweiterungen durch das Bremslichtmodell von Knospe Die bisher beschriebenen Modelle sind nicht in der Lage, das Bestreben eines Fahrzeugführers nach einer vorausschauenden Fahrweise zu berücksichtigen. Diese Lücke soll durch das Bremslichtmodell geschlossen werden. Während bei den bisher beschriebenen Modellen immer bis auf die letzte Zelle hinter einem vorausfahrenden Fahrzeug aufgefahren wurde um dann abrupt zu bremsen, wird jetzt ein vorausschauender Faktor hinzugefügt. Zur Umsetztung dieses Modells werden folgende Variablen benötigt.

p : Trödelwahrscheinlichkeit aus dem NaSch Modell

p_0 : Trödelwahrschienlichkeit für stehende Fahrzeuge

p_b : Trödelwahrscheinlichkeit für den Fall, dass das vorausfahrende Fahrzeug bremst

$b_n(t)$: Bremslicht des Fahrzeuges n zum Zeitpunkt t

t_h : benötigte Zeit um die Position des Vordermannes bei der aktuellen Geschwindigkeit zu erreichen

t_s : Einflussbereich des Vorausfahrenden Fahrzeugs $t_s = \frac{\Delta x_n(t)}{v_n(t)}$

Δx_{safety} : Sicherheitsabstand

$v_{anti,front}(t)$: geschätzte Geschwindigkeit des vorausfahrenden Fahrzeugs

$x_{eff}(t)$: effiektiver Abstand zum vorausfahrenden Fahrzeug unter Beachtung dessen geschätzter Geschwindigkeit

Es gibt drei Situationen die abhängig vom Abstand zum vorausfahrenden Fahrzeug eintreten können.

1. großer Abstand zum Vorausfahrenden: $v_n(t+1) = f(v_{max}, p_n)$
   Das Bremslicht des Vorausfahrenden ist nicht relevant.
2. mittlerer Abstand zum Vorausfahrenden: $v_n(t+1) = f(b_{n+1})$
   Das Bremslicht des vorausfahrenden Fahrzeugs $n+1$ hat Einfluss auf das Fahrverhalten von Fahrzeug n.
3. sehr geringer Abstand: $v_n(t+1) = f(\Delta x_{safety})$
   Das Fahrzeug ist ausschließlich bestrebt den Sicherheitsabstand herzustellen.

Die geschätzte Geschwindigkeit wird in einfachen Implementierungen aus der aktuellen Geschwindigkeit des vorausfahrenden Fahrzeugs ermittelt. Der zur Verfügung stehende effektive Abstand setzt sich dann aus der geschätzten Geschwindigkeit und einem Sicherheitsabstand Δx_{saftey} zusammen.

Das verwendete Modell Für das zu realisierte Verkehrsmodell werden die oben beschriebenen Modelle folgendermaßen zusammen gefasst.

- **Schritt 0 - Geschwindigkeits- oder Bremslichtabhängige Fluktuation**

  $$p_n = \begin{cases} p_b & WENN(b_{n+1} == 1\, UND\, t_h < t_s) \\ p_0 & WENN(v_n == 0) \\ p & sonst \end{cases}$$

  Wenn das Bremslicht des vorderen Fahrzeugs an ist, ist die Wahrscheinlichkeit p_b.
  Wenn das Fahrzeug steht, ist diese p_0,
  sonst p_d.
  Zu Beginn jeder Berechnung ist das Bremslicht aus, so ist b_n(t+1)=0 .

- **Schritt 1 - Beschleunigung**

  Ist $b_n(t) = b_{n+1}(t) = 0\, ODER\, t_h \geq t_s \Rightarrow\ v_n(t+1) = min\{v_n(t)+1, v_{max}\}$

  Wenn das eigene Bremslicht und das des Vordermannes aus ist oder wenn sich das Fahrzeug nicht im Interaktionsbereich des Vordermannes befindet, wird um eine Einheit beschleunigt, bis die maximale Geschwindigkeit erreicht ist.

- **Schritt 2 - Bremsen**

  $v_n(t+1)$ = min$\{v_n(t+1), x_{eff,n}(t)\}$
  gilt zusätzlich $v_n(t+1) < v_n(t) \Rightarrow\ b_n(t+1) = 1$

  Wenn die Geschwindigkeit größer ist als die effektive Anzahl freier Zellen zwischen dem Fahrzeug und dem vorausfahrenden Fahrzeug,wird die effektive Anzahl der freien Zellen als Geschwindigkeit übernommen. Ist dem nicht so, wird mit gleicher Geschwindigkeit weiter gefahren. Die effektive Anzahl freier Zellen berechnet sich aus dem Abstand zwischen dem Fahrzeug und seinem Vordermann, veringert um die geschätzte Geschwindigkeit des Vorausfahrenden. Zusätzlich wird noch ein Sicherheitsbereich x_{safety} addiert.

- **Schritt 3 - Fluktuation**

rand() $< p \Rightarrow\ v_n(t+1)$ = *max*$\{v_n(t+1) - 1, 0\}$
gilt zusätzlich $p = p_b \Rightarrow\ b_n(t+1) = 1$

Wenn getrödelt wird, verringere die Geschwindigkeit um 1 und wenn das Bremslicht des Vordermanns erkannt wurde, wird das Bremslicht angeschaltet.

- **Schritt 4 - Fahren**

  x_n(t+1) = x_n(t) + v_n(t+1)

  Das Fahrzeug wird um die ermittelte Geschwindgkeit vorwärts bewegt.

3.2.3 Mehrspurmodell

Beim Mehrspurverkehr muss das Modell für das Einspurverhalten auf alle Spuren angewendet werden und es muss den Verkehrsteilnehmern möglich sein zwischen benachbarten Spuren zu wechseln. Es wird als erstes das Wechselverhalten ausgewertet, um danach die Bewegung des Verkehrsteilnehmers durch das Einspurmodell zu berechnen. Dieses Wechseln schafft für den Verkehr neue Freiheiten, die durch entsprechende Regeln gelenkt werden müssen. Wie in Abschnitt 3.2 bereits erwähnt, wird das Mehrspurmodell nach Nagel (Abschnitt 3.2.3) Anwendung finden, da es einige Vorteile gegenüber den Modellen nach Rickert und Wagner bietet. Die beiden zuletzt genannten Modelle werden kurz vorgestellt und dessen Nachteile erläutert.

Modell nach Rickert Das Modell nach Rickert gibt einen Regelsatz bestehend aus vier Reglen vor. Die erste Regel überprüft, ob ein Vordermann existiert. Wird diese Regel für den Wechsel von rechts nach links sowie von links nach rechts ausgeführt, so kann nach den symmetrischen Spurwechelregeln gewechselt werden (amerikanisches System). Im Fall des verwendeten Verkehrsmodells darf diese Bedingung nur für Wechsel von rechts nach links gelten, da ein asymmetrisches Wechselverhalten benötigt wird (europäisches System). In Regel 2 wird verglichen, ob die Situation auf der Wunschspur günstiger ist als auf der momentan befahrenen Spur. Regel 3 prüft ob durch den Wechsel ein anderer Verkehrsteilnehmer behindert wird. Die letzte Regel führt eine stochastische Bedingung ein. Diese Bedingung bewirkt, dass ein möglicher Wechsel nur mit einer bestimmten Wahrscheinlichkeit ausgeführt wird. Durch Bedingung 4 wird Nichtdeterminismus erzeugt, um den Verkehr realistischer nachzubilden. Nachteile dieses Modells sind, dass asymmetrische Spurwechsel von Verkehrsteilnehmern meist von den vor ihnen fahrenden Teilnehmern ausgelöst werden. Das bedeutet, dass wenn der Vordermann die Spur wechselt, es der nachfolgende Teilnehmer in den meisten Fällen ebenfalls tut. Weiterhin wird das in der Realität beobachtbare Verhalten, das ab einem bestimmten Fahrzeugfluss die Dichte auf der linken Spur höher ist als auf der rechten, durch dieses Modell nicht nachgebildet. Für detailiertere Informationen wird auf [18] verwiesen.

Modell nach Wagner Im folgenden Modell nach Wagner gibt es 5 Bedingungen die abgearbeitet werden müssen bevor ein Verkehrsteilnehmer die Entscheidung zum Wechseln treffen kann.

- Bedingung 1 setzt voraus, dass auf der Spur auf die gewechselt werden soll kein Fahrzeug durch den eventuellen Wechsel zum Bremsen gezwungen wird. Der Extremfall ohne diese Bedingung

würde bedeuten, dass zum Beispiel Fahrzeug 1 wechselt, obwohl sich ein Fahrzeug 2 mit einer Geschwindigkeit von etwa 20 Zellen/Step nähert. Sollte Fahrzeug 2 nach dem Wechsel nur fünf Zellen hinter Fahrzeug 1 stehen, so müsste Fahrezug 2 extrem stark bremsen, was in der Realität eventuell nicht möglich wäre und einen Unfall verursachen würde. Die nächsten beiden Bedingungen betreffen den Wechselwillen des Fahrzeugs von links nach rechts.

- Bedingung 2 prüft, ob die gewünschte Geschwindigkeit (also v_{max}) auf der momentanen Spur erreichbar ist.
- Bedingung 3 kontrolliert, ob eventuell eine höhere Geschwindigkeit auf der Nachbarspur erreichbar ist. Analog dazu berechnen die letzten beiden Bedingungen den Wechsel von links nach rechts.
- Bedingung 4 überprüft, ob ein Vordermann auf der momentanen Spur existiert.
- Bedingung 5 prüft das selbe Kriterium für die rechte Spur.

Ein grober Nachteil dieses Regelsatzes ist die schlechte Auslastung der rechten Spur bei hoher Verkehrsdichte. Für dieses Problem wurde per stochastischer Entscheidung ein alternativer Regelsatz benutzt, welcher aus einer modifizierten Bedingung 1 und 5 besteht. Weiterhin stimmen die Geschwindigkeitsdifferenzen zwischen den Fahrzeugen mit der Realität nicht überein [18].

Mehrspurmodell nach Nagel Aufgrund der überwiegdenen Nachteile der beiden anderen Modelle ist die Entscheidung zu Gunsten des Modells nach Nagel gefallen. Es gibt nur zwei Bedingungen und eine einfache Struktur. Bedingung 1 beschreibt eine Sicherheitsüberprüfung, die erfüllt sein muss, so dass Bedingung 2 berechnet werden kann. Bedingung 2 prüft dann ob sich der Wechsel für das Fahrzeug lohnt. Der formale Regelsatz ist im folgenden aufgeführt.

Parameter

Δx^o_- *: Anzahl der freien Zellen auf der Zielspur hinter dem wechselnden Fahrzeug*

Δx^o_+ *: Anzahl der freien Zellen auf der Zielspur vor dem wechselnden Fahrzeug*

d - Vorausschaudistanz

v_r */* v_l *: Geschwindigkeit des Fahrzeugs auf der rechten Spur (*v_r*) / auf der linken Spur (*v_l*)*

Δ *: "Anpassungs Parameter"*

Bedingungen

Bedingung 1

Sicherheitsbedingung

Relativ zur eigenen Position darf sich im Bereich $\left[-\Delta x^o_-, \Delta x^o_+\right]$ kein Fahrzeug auf der Zielspur befinden.

Bedingung 2 (modifiziert)
Wechselbedingung

für $v > 0$
$(R \rightarrow L) : v \geq v_r OR v \geq v_l$
$(L \rightarrow R) : v < v_r - \Delta$ AND $v < v_l - \Delta$

für $v = 0$
$(R \rightarrow L) : v < v_l$
$(L \rightarrow R) : v < v_r$

Die Sicherheitsbedingung ist folgendermaßen zu verstehen. Wenn sich in einem bestimmten Bereich vor oder hinter dem betrachteten Fahrzeug ein Fahrzeug auf der Zielspur befindet wird nicht gewechselt, um einen Unfall zu vermeiden. Die Parameter Δx^o_- bzw. Δx^o_+ werden dabei mit der Maximalgeschwindigkeit der Straße bzw. der aktuellen Geschwindigkeit des Fahrzeugs belegt.
Bedingung 2 ist eine Modifikation der eigentlichen Wechselbedingung. Der Unterschied besteht darin, dass Fahrzeuge eine verkürzte Vorausschaudistanz d haben und erheblich langsamer fahren müssen als die Fahrzeuge auf der eigenen Spur und auf der Zielspur. Die ursprüngliche Bedingung 2 sieht wie folgt aus:

nicht-modifizierte Bedingnung 2

- **Bedingung 2**
  Wechselbedingung

  für $v > 0$
  $(R \rightarrow L) : v \geq v_r$ OR $v \geq v_l$
  $(L \rightarrow R) : v < v_r$ AND $v < v_l$

  für $v = 0$
  $(R \rightarrow L) : v < v_l$
  $(L \rightarrow R) : v < v_r$

Der Parameter d, welcher die Vorausschaudistanz beschreibt, wird zur Identifizierung des vorausfahrenden Fahrzeugs benutzt (Fahrzeuge im Bereich d des betrachteten Fahrzeugs gelten als vorausfahrendes Fahrzeug). Ohne Vorausfahrenden ist ein Wechsel nach links unnötig. Existiert ein vorausfahrendes Fahrzeug ist ein Wechsel nach rechts ggf. unnötig, da man rechts nicht überholen darf. Der Wert $d = 16$ in Verbindung mit der nicht modifizierten Bedingung 2 führt zu einer Dichteinversion (die linke Spur wird wesentlich mehr beansprucht als die rechte) bei geringerer Fahrzeugdichte, was so in der Realität nicht zu beobachten ist. Um dem vorzubeugen wurde Bedingung 2 modifiziert und der Wert für $d = 16$ auf $d = 7$ angepasst. Der Parameter Δ bekommt dabei den Wert 3.
Damit ergibt sich aus der modifizierten Bedingung 2 das man nur nach links wechselt, wenn man mindestens so schnell fahren kann wie auf der momentanen Spur oder auf der linken Spur. Und es wird nur nach rechts gewechselt, wenn man entsprechend langsamer ist als der Verkehr auf der rechten und momentanen Spur, abzüglich dem Anpassungs Parameter Δ. Das Fahrzeug muss also wesentlich

Tabelle 3.1: Modellparameter

Beschreibung	**Variable**	**Wert**
Zellenlänge	x	1,5m
Trödelwahrscheinlichkeit	p_d	0,1
Trödelwahrscheinlichkeit (Stau)	p_0	0,5
Trödelwahrscheinlichkeit (Bremslicht)	p_b	0,9
Maximal Geschwindigkeit der Pkw	$v_{max,vehicle}$	25 Zellen/Schritt
Maximal Geschwindigkeit der Lkw	$v_{max,truck}$	18 Zellen/Schritt
Sicherheitsabstand	x_{safety}	5 Zellen
Freie Zellen beim Überholen hinter dem Fahrzeug	x_-^0	v_{max}
Freie Zellen beim Überholen vor dem Fahrzeug	x_+^0	$v_n(t)$
Vorausschaudistanz	d	7 Zellen
Anpassungs Parameter	Δ	3
Anteil Lkw von allen Fahrzeugen		0,1
Einflussbereich des Bremslichts	h	6-11 Sekunden

langsamer fahren wenn es nach rechts wechseln will, da Δ den Wert 3 (Abschnitt 3.2.4) zugewiesen bekommt.
Ein Nachteil dieser Modifizierung von Bedingung 2 ist das die Fahrzeuge das Verkehrsverhalten nun realer nachbilden, aber dafür die Spuren in bestimmten Situationen (z.B Stau) nicht mehr realitätsnah ausnutzen. Aus diesem Grund werden im Fall $v = 0$ (also im Stau) eine symmetrische Wechselbedingung benutzt, welche besagt dass das Fahrzeug nur noch vergleicht, ob es auf der Zielspur schneller/langsamer fahren kann als auf der momentanen Spur und dann nach links/rechts unter Berücksichtigung der Sicherheitsbedingung wechselt. Ein weiterer Nachteil besteht darin, dass der Spurwechsel zu schnell vollzogen wird. Wenn man davon ausgeht das pro Berechnung für alle Fahrzeuge 1 diskreter Schritt (also 1 Sekunde vergeht) gemacht wird, so dauert ein kompletter Spurwechsel nur 1 Sekunde. Als Abhilfe wurde deswegen das Wechseln von links nach rechts nur in geraden und das Wechseln von rechts nach links nur in ungeraden Schritten zugelassen, damit entspricht ein Spurwechsel 3 Sekunden, was eindeutig Realitätsgetreuer ist [18].

3.2.4 Parameter

Wie schon vorgestellt, werden das Einspurmodell nach Knospe und das Mehrspurmodell nach Nagel benutzt. Die Modelle benötigen beide eine Vielzahl von Variablen und Parametern, die im folgenden Kontext vorstellt werden. Die Auswahl der Parameter beruhen im wesentlichen auf den Informationen aus [18].

3.2.5 Erweiterungen

Das Mehrspurmodell soll in Verbindung mit dem Einspurmodell Verkehr auf mehrspurigen Straßen möglichst realitätsnah simulieren. Allerdings werden im Kontext der Modelle keine Angaben dazu gemacht, wie das Verhalten an Kreuzungen aussieht, wie ein Fahrzeug eine Autobahnauffährt benutzt oder was an Engstellen passiert, an denen eine Spur wegfällt. Diese Probleme mussten möglichst realitätsnah gelöst werden. Die entsprechenden Lösungsansätze werden im folgenden vorgestellt. Wei-

terhin werden auch einige Verbesserungen aus [18] vorgestellt, die es den Modellen erlauben den Verkehr noch realer darzustellen.

Detailgrad Einspurmodell Um den Detailgrad des Modells zu erhöhen, wurde die Zellengröße auf $1,5m$, statt $7,5m$ welche in Abschnitt 3.2.2 genannt wurden, festgelegt. Dadurch ist ein realistischeres Beschleunigungsverhalten von $1,5\frac{m}{s^2}$ gegeben. Ein Fahrzeug belegt dann nicht mehr genau eine Zelle, sondern mehrere. Als durchschnittliche Fahrzeuglänge wurden $6m$ angenommen und damit werden 4 Zellen belegt. Außerdem ist es vorgesehen verschiedene Fahrzeugtypen, wie beispielsweise Lkw und Pkw, mit unterschiedlichem Brems- und Beschleunigungsverhalten zu realisieren. Lkw werden im Moment als längere Pkw ($15m$ Länge) dargestellt, allerdings mit gleichem Brems- und Beschleunigungsverhalten.

Antizipierte Geschwindigkeit im Einspurmodell Die antizipierte Geschwindigkeit [18] dient der Schätzung der Geschwindigkeit anderer Verkehrsteilnehmer. Im einfachsten Fall wird diese durch die aktuelle Fahrzeuggeschwindigkeit geschätzt. Dieses Detail wird im Moment nicht verwendet. Die antizipierte Geschwindigkeit kann nicht ohne Weiteres in das verwendetete Straßenmodel übertragen werden, da dieses aus vielen komplex, miteinander verbundenen Kanten mit unterschiedlich vielen Spuren zusammengesetzt ist. Erst ein relativ hoher Wert für den Sicherheitsabstand führt zu unfallfreiem Fahren. Dadurch wird aber die eigentliche Idee der antizipierten Geschwindigkeit nicht mehr sinnvoll umgesetzt. Die Realisierung des gleichmäßiger fließenden Verkehrs wird also einzig durch die Bremslicherweiterung angenähert.

Kreuzungen Das Verhalten an Kreuzungen ist ein Problem, da alle Fahrzeuge separat betrachtet werden und keiner gemeinsamen Kontrollinstanz unterliegen. Als erster Ansatz wird die Steuerung des Verkehrs durch Ampelanlagen in Erwägung gezogen. Ampeln haben in Deutschland vier Phasen (grün, gelb, gelb-rot und rot) und werden innerhalb des Einspurmodells in die Berechnung integriert. Bei der grünen Phase ist es erlaubt die Kreuzung zu überqueren, bei gelb, gelb-rot und rot wird gehalten.
Um Vorfahrtsregeln zu integrieren musste abstrahiert werden. Straßen werden unterschiedlich genutzt (z.B. Autobahn, Landstraße, etc.) und haben somit unterschiedliche Eigenschaften (Geschwindigkeitslimit, Spurbreite, etc.) die beachtet werden müssen, deshalb wird ihnen eine Priorität zugeordnet. Die erste Regel an Kreuzungen lautet, dass ein Auto von einer höherwertigeren Straße Vorrang hat, vor einem Auto das von einer niedriger eingestuften Straße kommt. Fahren nun zwei Autos gleicher Priorität auf die Kreuzung zu, welche beide die höchst priorisierten Fahrzeuge sind, erhält das zuerst berechnete Fahrzeug den Vorrang. Es entsteht dadurch das Problem, dass auf Kreuzungen nicht der maximale Durchsatz erzielt wird, da immer nur ein Fahrzeug die Kreuzung passieren kann. Allerdings entsteht ein akzeptabler Fehler, da dadurch keine Verhaltensfehler auf Autobahnen passieren. Autobahnenwerden nicht durch Kreuzungen sondern durch Auf- und Abfahrten befahren und verlassen. Die korrekte Umsetzung aller Vorfahrtsregeln erfordert eine exakte Information darüber, welche Straßen von rechts und links einmünden und wie die Abbiegespuren gestaltet sind.

Auffahrten auf Autobahnen Auffahren auf eine Autobahn ist weitaus weniger komplex als das Problem der Kreuzungen, allerdings ergeben sich auch hier zwei Probleme. Zum Einen existieren in

den geographischen Daten keine gesonderten Informationen zu den Auffahrten, sie sind als einfache Kreuzungen in den Daten enthalten und sehen eine Spur für die Auffahrt nicht vor (Beschleunigungsspur). Zum Anderen ist eine Auffahrt irgendwann zu Ende und bis dahin muss das Fahrzeug von der Beschleunigungsspur gewechselt haben, da innerhalb der Daten auch keine Standstreifen existieren. Das erste Problem wurde anhand der Informationen aus den Intermediatedaten gelöst. Näheres dazu ist im Abschnitt (3.1.2) zu finden. Das zweite Problem wurde im ersten Ansatz so gelöst, dass Auffahrten durch einen extra Straßentyp gekennzeichnet sind und Fahrzeugen, die sich auf der Autobahn befinden das Wechseln auf diesen Typ von Straße verwehren. Fahrzeuge die auffahren wollen, beachten die Wechselbedingung aus dem Mehrspurmodell aus Abschnitt 3.2.3 nicht, sie prüfen nur den Sicherheitsbereich. Der Sicherheitsbereich wurde verkleinert (nach hinten zehn Zellen, nach vorne eine Zelle), so dass ein Wechsel schneller vollzogen wird.

Spurverengung Unter Spurverengung versteht man das Wegfallen einer Spur im Straßennetz. Es kann beispielsweise passieren das sich drei Spuren hinter einer Kreuzung auf 2 Spuren verengen, da z.b. eine Baustelle, ein Unfall oder Sonstiges die dritte Spur blockiert. Bei Spurverengungen wurde der Ansatz verfolgt, dass immer die rechteste Spur wegfällt und nur die Fahrzeuge der rechtesten Spur auf die Nachbarspur wechseln müssen. Es wird wie im Mehrspurmodell (Abschnitt 3.2.3) ein Sicherheitscheck auf der linken Spur durchgeführt und gewechselt sobald eine Lücke groß genug ist damit das Fahrzeug wechseln ohne ein nachfolgendes Fahrzeug zu behindern (Sicherheitsabstand des nachfolgenden Fahrzeugs zum betrachteten Fahrzeug entspricht der Geschwindigkeit des nachfolgenden Fahrzeugs). In der Realität kommen Spurverengungen willkürlich vor (linke Spur fällt weg, Rechte Spur fällt weg oder beiden münden in eine mittige Spur) und können nicht so einfach gehandhabt werden wie es dieser erste Ansatz tut. Eine realitätsnahere Umsetzung dieses Problems kann als zukünftiges Ziel festgehalten werden.

Kapitel 4

Implementierung

4.1 Einleitung

Die zentrale Aufgabe der PG ist es den BeeHive-Algorithmus an die vorliegende Problemdomäne der Stauvermeidung im Straßenverkehr anzupassen, geeignet umzusetzen und mit klassischen Verfahren zu vergleichen. Da sich das anzugehende Problem der Stauvermeidung bei genauerer Betrachtung als ein mehrkriterielles, kombinatorisches und NP-schweres Optimierungsproblem mit dynamischer Zielfunktion herausstellt, verwehrt es sich einer einfachen analytischen Lösungsstrategie. Stattdessen ist aus praktischer Sicht eine Simulation die einzige Möglichkeit die Wirksamkeit des aus dem BeeHive- abgeleiteten BeeJamA-Algorithmus aussagekräftig zu untersuchen. Somit motiviert sich die Forderung nach einem Verkehrssimulator-Framework, dessen, von der PG vorgenommene, Implementierung im vorliegenden Kapitel vorgestellt werden soll. Dieser Simulator diente der PG um den im Abschnitt 2.4 vorgestellten BeeJamA-Algorithmus zu testen und ihn mit weiteren Routingalgorithmen vergleichen zu können. Die Ergebnisse dieser Vergleichsstudien finden sich in Kapitel 5. Das vorliegende Kapitel strukturiert sich wie folgt. Zunächst werden die ermittelten Anforderungen an den Simulator und die wesentlichen Designentscheidungen vorgestellt, welche zusammen die Rahmenbedingungen der durchgeführten Implementierungsarbeit darstellen. Wie weiter unten gezeigt wird, unterteilt sich der Simulator in mehrere Komponenten, welche im Folgenden in Funktion und Umsetzung einzeln erläutert werden. Für einen Überblick wie der Simulator programmatisch an neue Bedürfnisse in weiterführenden Arbeiten angepasst werden kann, sei auf den Anhang hingewiesen.

4.2 Anforderungen

Die hier bündig aufgelisteten Anforderungen an den Simulator ergeben sich zum einen aus der Aufgabenstellung der PG und zum anderen aus einer Anforderungsanalyse zu Beginn des ersten Semesters der Bearbeitungszeit. Es kristallisieren sich insgesamt sechs Funktionsgruppen herraus, in welche sich die gewünschten Anforderungen klassifizieren lassen. Im einzelnen ergeben sich die folgenden Funktionsgruppen:

Entitätenmodellierung: Grundbaustein einer Simulation stellen die zu simulierenden Entitäten aus der realen Welt dar. Zentrale Anforderung ist es somit, die für das in Kapitel 3 dargestellte

makroskopische Verkehrssimulationsmodell, relevanten Objekte, wie Straßen, Fahrzeuge und andere staurelevenate Elemente, wie z.B. Ampeln, zu modellieren und eine effiziente und intuitive programmatische Nutzung dieser Modelle zu ermöglichen.

(Heuristische) Datenaufbereitung: Um die Effektivität der zu entwickelnden Algorithmen aussagekräftig testen zu können, sollen die Simulationsläufe auf realitätsnahen Straßenmodellen durchgeführt werden. Konkret wird gefordert, dass als Straßenmodell das Ruhrgebiet, als der größte europäische Ballungsraum, verwendet wird. Dieses Straßenmodell muss aus vorgegebenen Datensätzen heraus aufgebaut werden können und ggf. fehlende staurelevante Angaben heuristisch oder interaktiv vom Benutzer unterstützt hinzugefügt werden können. So sind Ampeln zweifelsohne eine staurelevante Entität, werden in den üblichen Datensatzformaten allerdings nicht berücksichtigt. Eine Heuristik der Form „Innerorts steht an großen Kreuzungen zu 90% eine Ampel“, kann hier helfen realitätsnähere Straßenmodellinstanzen zu generieren. Gleiches gilt für die Generierung von Verkehr. Detaillierte Daten zum Verkehrsaufkommen auf einzelnen Straßen sind entweder nicht öffentlich oder gar nicht vorhanden, weswegen auch hier heuristische Regelsätze gefordert sind. Folgende Regel gelte als Beispiel: „Jeden Morgen verlassen X Fahrzeuge die Wohnsiedlung Y, jeden Nachmittag kehren sie zurück”.

Verkehrsmodellierung: Makroskopische Verkehrssimulationskonzepte sehen vor jedes einzelne Fahrzeug als eigenständige Entität zu modellieren und die diskrete Position eines Fahrzeugs zu jedem diskreten Zeitpunkt anhand von Regelsätzen, die versuchen menschliches Verhalten zu imitieren, zu bestimmen. Gemäß Aufgabenstellung sollen Regelsätze umgesetzt werden die auf Erweiterungen des Schreckenberg-Verkehrsmodells basieren und in Abschnitt 3.2 erläutert sind.

Routing: Die zentrale Anforderung für den Simulator in diesem Zusammenhang ist die Umsetzung des in Abschnitt 2.4 beschriebenen BeeJamA-Algorithmus. Es ist gefordert, dass die Implementierung auch unter Effizienzbetrachtungen geschieht, damit die Grundanforderung umgesetzt werden kann Systeme in der Größenordnung des Ruhrgebiets simulieren zu können.

Visualisierung: Weiterhin ist es zu Debugging- und Anschauungszwecken notwendig die simulierte Welt graphisch aufzubereiten. Nur so ist es möglich Erfolge intuitiv zugänglich zu machen, aber auch Misserfolge und Fehler in den Implementierungen der Simulationsverfahren und des BeeJamA-Algorithmus ausfindig zu machen.

GUI: Der Simulator soll weiterhin eine GUI besitzen, um eine komfortable Konfiguration und das Starten von Simulationen zu ermöglichen.

Auf eine vollständige Auflistung aller Anwendungsfälle, die daraus resultierenden Anforderungen und deren Zuordnung zu den einzelnen Funktionsgruppen sei hier verzichtet. Wie weiter unten gezeigt wird, ist ein Großteil der oben definierten Anforderungen während des vergangenen Jahres implementiert worden, noch offene Punkte werden in Abschnitt 6.1 aufgezeigt. Wie bereits erwähnt, ist im PG-Antrag darüberhinaus als Minimalanforderung definiert, dass das Ruhrgebiet mit einer realistischen Anzahl von Fahrzeugen simuliert werden kann und darüber hinaus eine Vergleich zwischen dem entwickelten BeeJamA- und einem Vergleichsroutingalgorithmus durchgeführt wird. Daraus folgt, dass der entwickelte Simulator viele zehntausende Fahrzeuge effizient auf großen Verkehrsszenarien simulieren können muss. Da das ganze Simulationsframework durch Folgearbeiten erweitert werden soll, besteht zudem eine (nicht-funktionale) Anforderung darin, dass die oben definierten Funktionsgruppen derartig in Softwarekomponenten verkapselt werden, dass eine spätere Erweiterung oder gar vollständgier Austausch einer einzelnen Komponente leicht durchführbar ist. Für die Implementierung der Komponenten selber wird Java als Zielplattform gefordert. Diese Anforderung resultiert aus

der Tatsache, dass die Mitglieder der PG durch das Curriculum des Fachbereichs Informatik der Universität Dortmund seit Beginn des Studiums die Programmiersprache Java erlernt haben und demnach über genügend Erfahrung bei der Erstellung von Software für diese Zielplattform verfügen. Zudem eignet sich Java hervorragend dazu, wie im vorliegenden Fall verlangt, schnell eine prototypische Implementierung der zu testenden Algorithmen und des Simulatorframeworks zu schaffen, welche zeitgleich flexiblel und wartbar ist um in nachfolgenden Arbeiten stetig erweitert zu werden.

4.3 Designentscheidungen

Zu Beginn des ersten PG-Semesters wurde zunächst eine Evaluierung der frei verfügbaren Verkerssimulatoren durchgeführt, mit obigen Anforderungen verglichen und eine Aufwandsanalyse für die Umsetzung dieser erarbeitet. Das Ergebnis dieser Vorüberlegungen und zugleich wichtigste Designentscheidung ist, dass das benötigte Simulationsframework vollständig neu entwickelt werden sollte. Die weiteren wesentlichen Designentscheidungen die verbindlich getroffen wurden, um die oben genannten Anforderungen in dem neu zu entwickelnden Simulator umzusetzten, seien im Folgenden genannt. Wie im Abschnitt 2.4 ausführlich dargestellt, handelt es sich bei dem entwickelten BeeJamA-Algorithmus um einen inherent dezentralen Routingalgorithmus. Somit würde eine Implementierung als verteiltes System nahe liegen, jedoch wurde aus zeitgründen entschieden, dass zunächst ein monolithischer Simulator entwickelt wird. Für geplante Weiterentwicklungen sei auf Abschnitt 6.1 verwiesen. Die grundsätzliche Arbeitsweise der Simulation ergibt sich aus der Art des verwendeten Verkehrssimulationsprinzips. Schreckenberg-basierte Modelle arbeiten, wie in Abschnitt 3.2 dargestellt, diskret in den Dimensionen Ort und Zeit. Vor diesem Hintergrund ist die Designentscheidung, den Simulator nach dem gleichen Paradigma zu konstruieren, leicht nachvollziehbar. Im Gegensatz dazu stünde ein kontinuierliches Simulationsmodell, welches beispielsweise mittels Differentialgleichungen modelliert werden könnte. Statt kontinuierlichen Ortsangaben werden also die Straßen in diskrete Zellen konstanter Größe eingeteilt und statt kontinuierlichen Zeitangaben wird die Simulation in diskrete Zeitschritte eingeteilt. Diese Vorgehensweise sichert auf der einen Seite eine im Vergleich zu kontinuierlichen Modellen einfache Implementierung und schnelle Laufzeit, ist bei genügend klein gewählten Orts- und Zeiteinheiten aber auch hinreichend präzise. Als Standard für den Simulator wurde für die zeitliche Dimension eine Auflösung von einer Sekunde und für die Ortsdimension eine Auflösung von $1,5$ Meter festgelegt, wobei beide Parameter selbstverständlich frei konfigurierbar sind. Bei einer angestrebten Zielgröße der Simulation von vielen zehntausend Fahrzeugen auf vielen hundert Kilometern Straßennetz, sollte dadurch eine ausreichend hohe Genauigkeit garantiert sein. Als Datenformat für die Straßendaten wurde, wie in der Aufgabenstellung gefordert, das AND-Format gewählt und als Routing-Algorithmus neben dem BeeJamA-Algorithmus wurde der Dijkstra-Algorithmus als Vergleichsalgorithmus bestimmt. Im Sinne der oben geforderten Kapselung und flexiblen Implementierung, wurden zunächst den einzelnen Funktionsgruppen jeweils eine Softwarekomponente zu geordnet. Folgende Auflistung gibt einen Überblick über die Komponenten, deren abstrakte Aufgabe sowie deren aktuelle Umsetzung. Die abstrakte Aufgabe und das Zusammenspiel der Komponenten bleibt dabei gleich, die aktuelle Umsetzung kann jedoch je nach Art der Weiterentwicklung unterschiedliche Ausprägungen annehmen.

Entity Component (EC): Beinhaltet die grundlegenden Entitäten der Verkehrsdomäne. Aktuell wird eine makroskopische Auswahl an simulierten Entitäten getroffen.

Data and Heuristic Component (DHC): Übernimmt das Einlesen und Aufbereiten der Straßen- und

Verkehrsdaten. Aktuell wird das AND-Datenformat verwendet und gegebenfalls heuristisch ergänzt.

Routing Algorithm Component (RAC): Berechnet eine Route für jedes Fahrzeug vom Start- zum Endpunkt. Aktuell exisitieren zwei Umsetzungen, zum einen für den BeeJamA-Algorithmus und zum anderen für den Dijkstra-Algorithmus.

Traffic Model Component (TMC[1]): Bestimmt die Position und Geschwindigkeit der einzelnen Fahrzeuge auf dem von der RAC vorgeschriebenen Weg zu jedem Zeitpunkt. Aktuell wird das in Abschnitt 3.2 beschriebene Schreckenberg-basierende Modell umgesetzt.

Visualisation Component (VC): Stellt den Zustand der Simulation makroskopisch graphisch dar. Aktuell wird eine JoGL-Ansicht unterstützt.

GUI Component (GC): Ermöglicht das intuitive Konfigurieren und Starten einer Simulation. Aktuell existiert eine Swing-GUI über welche die wichtigsten Parameter konfiguriert werden können, sowie die Simulation gestartet und gestoppt werden kann.

Über diese Komponenten hinaus existieren drei weitere, welche die grundlegende Simulatorumgebung zur Verfügung stellen:

Chain Component (CC): Kapselt die technische Grundlage für den zeitdiskreten und modularen Ablauf der Simulation. Das verwendete Operator-Chain-Konzept wird in Abschnitt 4.6.2 erläutert.

Simulator Component (SC): Stellt die zentrale Schnittstelle zwischen den einzelnen Komponenten dar und ermöglicht das Starten der Simulation. Die oben beschriebene GC stellt ein graphisches Frontend für die Funktion dieser Komponente zur Verfügung.

Metasimulator Component (MSC): Der Metasimulator ermöglicht das Erstellen und statistische Auswerten von kompletten Testreihen sowie die automatische Parameteroptimierung.

4.4 Übersicht und Status der Implementierung

Im Folgenden soll ein grober Überblick über die Umsetzungen der oben definierten Anforderungen und Designentscheidungen gegeben werden, die einzelnen Komponenten werden dann weiter unten ausführlich dargestellt. Die Implementierungen sind soweit vorangeschritten, dass die aufgestellten Anforderungen größtenteils realisiert werden konnten. Für darüber hinaus geplante Weiterentwicklungen sei auf Abschnitt 6.1 verwiesen. Der erstellte Simulator steht unter der GNU General Public License (vgl. [7]) und ist unter [4] frei verfügbar. Neben den selbst erstellten Softwarekomponenten, werden einige Bibliotheken zusätzlich verwendet, welche ebenfalls unter einer OpenSource-Lizenz frei verfügbar sind (näheres in Abschnitt 7.1). Somit steht eine vollständige OpenSource-Toolchain zur Simulation von Straßenverkehr zur Verfügung, die es leicht ermöglicht innovative Routingalgorithmen, wie den BeeJamA-Algorithmus, in realitätsnahnen Modellen simulativ zu testen. Struktuiert ist der vollständig in Java 6 (Standard Edition) geschriebene Quelltext in Packages, welche gemäß den *Java Code Conventions* (vgl. [8]) unterhalb der Hierarchie *edu.udo.cs.pg502* angeordnet sind. Die Abbildung 4.1 gibt einen Überblick über die wichtigsten Packages. Wie ersichtlich ist jedem der oben genannten Komponenten ein eigenes Package zugeordnet:

Darüber hinaus existieren einige zusätzliche Packages:

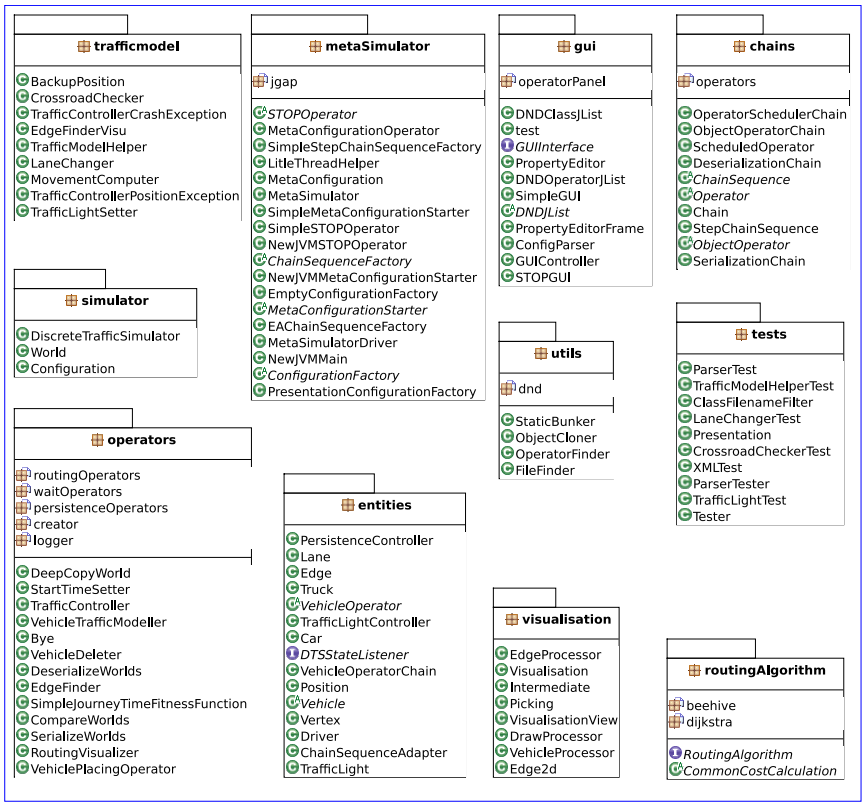


Abbildung 4.1: Die wesentlichen Packages im Überlick.

operators: Sammelbecken einiger allgemeiner Operatoren.

utils: Hilfsklassen, u.a. für die Persistenz.

tests: JUnit-Tests.

Für grundlegende Aufgaben wurde z.T. auf zusätzlichen Java-Bibliotheken zurückgegriffen. Diese Bibliotheken sind für das Kompilieren (genauer gesagt, das Linken) des Quelltextes des Simulators zwingend notwendig und sind im „lib"-Verzeichnis zu finden. Folgende Bibliotheken werden verwendet:

Jdom: Zum Parsen von XML-Konfigurationsdateien

Jgap: Eine Framework für Evolutionäre Algorithmen (EAs). Wird momentan zur Optimierung einiger Parameter durch den Metasimulator (vgl. 4.8) verwendet.

jGraphT: Eine Bibliothek mit klassischen Graphenalgorithmen. Der dort implementierte Dijkstra-Algorithmus dient als Ausgangspunkt eines Vergleichsalgorithmus für den BeeJamA-Algorithmus.

JoGL: Java-Bindings für OpenGL. Wird von der Visualisierungskomponente verwendet.

JUnit: Dient als Grundlage für die erstellten Unit-Tests.

log4j: Eine Bibliothek, welche umfangreiche Logging-Funktionen bietet.

In den folgenden Abschnitten seien nun die oben eingeführten Komponenten und anderweitig relevante Softwareelemente vorgestellt, wobei insbesondere Wert auf eine umgangssprachliche und intuitive Darstellung gelegt wird, um die verwendeten Konzepte dem Leser möglichst leicht zugänglich zu machen. Aus diesem Grund sei für Implementierungsdetails und andere technische Informationen, wie beispielsweise die Signaturen der jeweiligen Methoden, verzichtet und stattdessen auf die vorhandene JavaDoc-Kommentare verwiesen.

4.5 Entity Component

Die Entity Component (EC) bildet die Grundlage des Simualtionsframeworks. Sie beinhaltet die Java-Modellierung der zu simulierenden Entitäten aus der Verkehrsdomäne, wie z.B. Straßen, Fahrzeuge, Ampeln und ähnliches. Im Folgenden soll die Abbildung der Entitäten aus der Verkehrsdomäne in objekt-orientierte Modelle dargestellt werden. Jede dieser Entitäten ist in einer eigenen Klasse im „entities"-Package implementiert. Hierbei handelt es sich jeweils, sofern abgeleitete Klassen existieren, um die Basisklassen. Die abgeleiteten Klassen finden sich in den jeweiligen Packages der Komponenten, welche die erweiterte Funktionalität benötigen.

Straßennetz

Das reale Straßennetz wird durch einen gerichteten Graphen G modelliert[2]. Vereinfacht dargestellt, entsprechen Straßen in diesem Modell Kanten des Graphen, wobei jede Kante einen Streckenabschnitt einer Straße mit gleichen Attributen, wie Maximalgeschwindigkeit und Anzahl der Spuren entspricht.

[2]Mit $G = (V, E), E \subseteq V \times V, \forall v \in V : outEdges(v) := \{e \mid e \in E : \forall w : v \xrightarrow{e} w\}, \forall v \in V : inEdges(v) := \{e \mid e \in E : \forall w : w \xrightarrow{e} v\}$

Verbunden sind Kanten jeweils über Knoten, was impliziert, dass ein Knoten nicht wie man intuitiv annehmen könnte, nur Kreuzungen symbolisieren, sondern auch ganz allgemein zwei Streckenabschnitte mit unterschiedlichen Attributen verbinden können. Konsequenterweise verbietet sich eine geographische Interpretation eines Graphen. Die Abbildung 4.2 zeigt die beiden Klassen *Edge* und *Vertex* die zur Umsetzung solch eines Graphen. Zusätzlich ist jede Kante in solch einem Graphen implizit gerichtet. Falls eine Kante e_1 zu einer Menge von ausgehenden Kanten (*outEdges*) vom Knoten v_1 gehört und zu der Menge der eingehenden Kante (*inEdges*) von Knoten v_2, dann ist es offensichtlich, dass die Kante e_1 von v_1 nach v_2 gerichtet ist. Die Abbildung 4.3 illustriert ein Beispiel. Um eine Situation zu modellieren, indem eine Straßensektion in zwei Richtungen befahrbar ist, muss eine zweite Kante hinzugefügt werden. Ein realistisches Modell muss darüber hinaus auch einzelne Fahrspuren unterstützen, um Abbiegevorgänge simulieren zu können. Dabei gehört jede Spur genau zu einer Kante und jede Kante hat mindestens eine Spur. Abb. 4.4 zeigt ein Beispiel. Sei l_1 eine Spur die zur Kante e_1 gehört und Spur l_2 zur zweiten Kante e_2. Zusammen bilden sie den unteren Teil der „Sesame Street". Verbunden sind Fahrspuren durch die *PreLane*- und *PostLane*-Referenzen. Eine PreLane ist dabei eine Fahrspur von der man auf die aktuelle Spur fahren kann und eine PostLane demnach eine Spur auf die man von der aktuellen Spur wechseln kann. Über diese Verkettung ist es möglich realitätsnahe Abbiegevorgänge zu modellieren, dazu betrachte man als Beispiel die Abbildung 4.5. Ohne diese Verkettung wäre es möglich von allen Spuren einer Kante auf alle Spuren einer Nachfolgekante zu wechseln, welches in realen Straßennetzen zu Unfällen führen würde. Setzt man im Beispiel allerdings $post(l_1) = \{l_4, l_7\}$ und $post(l_2) = \{l_5\}$, sowie die anderen Pre- und Postmengen entsprechend, dann ist es nur möglich von Spur l_2 geradeaus auf Spur l_5 zu fahren statt zusätzlich auf Spur l_7, wie man es bei Betrachtung des entsprechenden Graphen zunächst vermuten könnte. Für eine vollständige Umsetzung des Beispiel in Quelltext sei auf den Anhang verwiesen. Wie oben bereits erwähnt, arbeiten Schreckenberg-basierte Verkehrsmodelle zeit- und ortsdiskret. Um die Ortsdiskretheit umzusetzten, werden Spuren in kleinere Einheiten, die sogenannten Zellen, unterteilt, wobei die Standardgröße 1,5m beträgt. Zellen werden nicht durch eigenständige Entitäten modelliert, da ihre schiere Anzahl einfach zu groß wäre. Sie werden stattdessen über das Integer-Attribut *Edge.numCells* angegeben, wodurch alle Fahrspuren einer Kante dieselbe Länge aufweisen.

Position

Die Klasse *Position* ist vergleichbar einfach strukturiert. Instanzen dieser Klasse werden benutzt um den Aufenthaltsort anderer Entitäten, insbesondere Fahrzeugen, zu speichern. Ein Aufenthaltsort ist durch ein 3-Tupel von Kante, Spur und Zelle voll qualifiziert. Abbildung 4.6 zeigt das zugehörige Klassendiagramm. Darüber hinaus wird diese Klasse genutzt um die belegten Zellen eines Fahrzeugs im Straßenmodell zu beschreiben. Dazu hält jedes Fahrzeug zwei Referenzen auf *Position*-Objekte, nämlich einmal auf eine *headPosition* und einmal auf eine *tailPosition*, um den Anfang und das Ende eines Fahrzeugs zu markieren.

Fahrzeuge und Fahrer

Das Konzept von simulierten Fahrzeugen ist zweigeteilt und in den beiden Klassen *Vehicle* und *Driver* umgesetzt. Abbildung 4.7 illustriert die Attribute dieser Klassen, es wird ersichtlich, dass jedes *Vehicle*-Objekt genau eine Referenz zu einem *Driver*-Objekt hält. So wird das Prinzip modelliert,

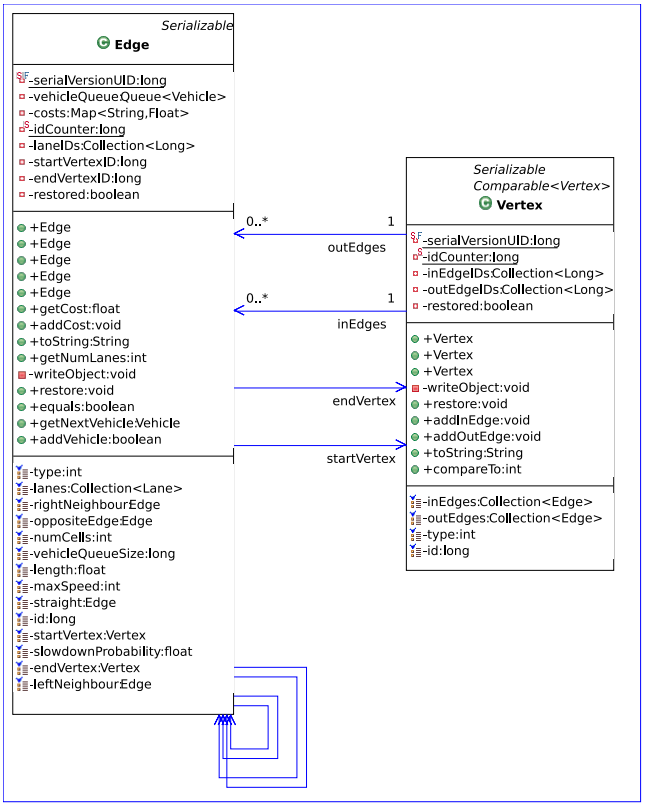


Abbildung 4.2: Die *Edge* und *Vertex* Entitäten.

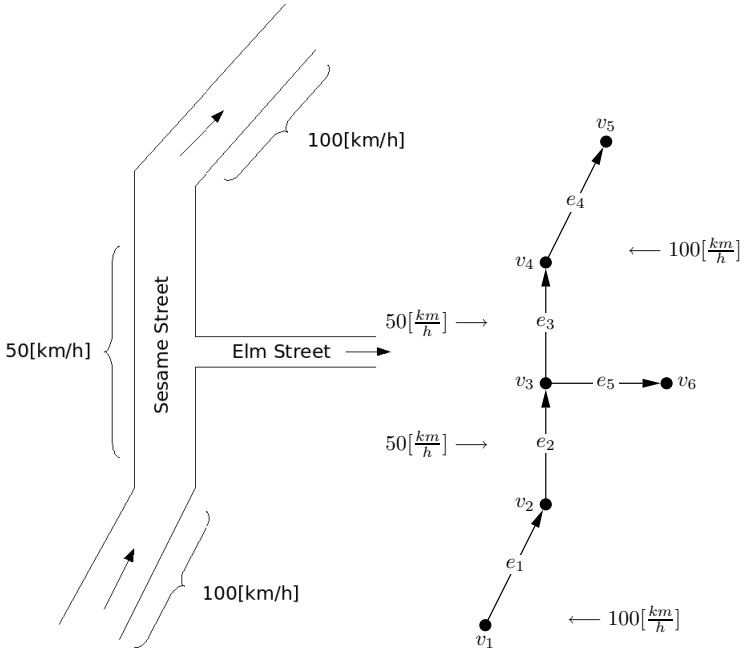


Abbildung 4.3: *(Links)* Zwei Straßen mit unterschiedlichen Maximalgeschwindigkeiten. *(Rechts)* Der entsprechende Graph. Straßensektionen werden zu Kanten, welche durch Knoten verbunden sind. Das Arragement der Graphen dient ausschließlich der besseren Lesbarkeit und impliziert demnach keine Informationen über das Aussehen des modellierten Straßennetzes, d.h. jeder isomorphe Graph repräsentiert das selbe Straßennetz (mit Ausnahme der Anzahl der Fahrspuren).

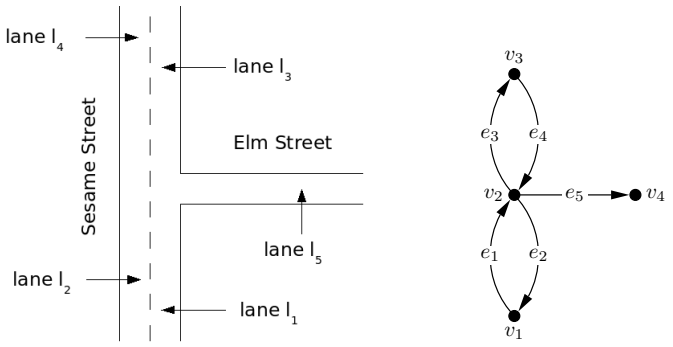


Abbildung 4.4: *(Links)* Ein einfaches Straßennetz mit Fahrspuren. *(Rechts)* Umsetzung als Graph mit einer Kante pro Fahrtrichtung.

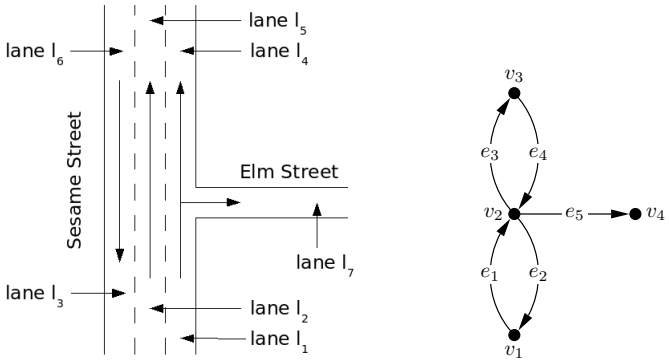


Abbildung 4.5: *(Links)* Ein Straßennetz mit mehreren Spuren pro Kante. *(Rechts)* Derselbe Graph wie im obigen Beispiel, jedoch jetzt mit zwei Spuren (l_1, l_2) auf der Kante e_1 und mit zwei Spuren (l_4, l_5) auf der Kante e_3.

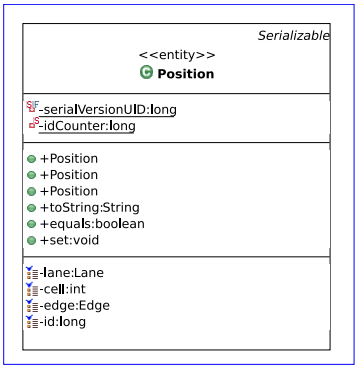


Abbildung 4.6: Die *Position* Entität.

dass jedes Fahrzeug genau von einem Fahrer gelenkt wird. Ein Fahrer kennt dabei insbesondere den Anfang und das Ende seiner Route, modelliert wird dies duch die Attribute *Driver.source* und *Driver.destination*. Durch die Separation von Fahrern und Fahrzeugen wird es leichter die Simulation von unterschiedlichen Fahrprofilen zu unterstützen. Man betrachte ein Szenario in dem jeden morgen 80% der Einwohner einer Siedlung 30km im Durchschnitt zur morgendlichen Rush-Hour zur Arbeit fahren und dass die restlichen 20% zufällige Routen fahren. Es ist unmöglich den Start- und das Zielort für diese Profile manuell festzulegen, über statistische Regeln hingegen ist dieses Verhalten modellierbar.

Die wichtige Funktion der beiden Klassen, *Driver* und *Vehicle*, liegt in der Eigenschaft, dass sie zwischen der TMC und der RAC vermitteln. Die TMC berechnet die exakten Positionen der Fahrzeuge und setzt das *headPosition*-Attribut der *Vehicle*-Objekte entsprechend. Die RAC hingegen nutzt diese globalen Informationen über die Positionen der Fahrzeuge und berechnet, sobald von einem Fahrzeug angefordert, die gewünschte Route als eine Liste von Kanten. Die RAC speichert die berechneten Kanten dann in der Queue des Fahrers (*Driver.route*). Die Semantik dieser Queue besteht darin, dass die TMC an jeder Kreuzung genau die Kante wählt, welche dem ältesten Elementes auf der Queue entspricht. Die TMC ruft diese Kante dann mittels der Methode *Driver.getNextRouteEdgeAndRemove()* ab, um die nächste Kante auf dem Weg vom Start zum Zielpunkt zu bekommen. Wie der Methodenname es andeutet wird die Kante gleichzeitig gelöscht, allerdings auch an die Liste der bereits besuchten Knoten angehängt (*visitedEdges*). Die Verwendung einer Queue garantiert, dass verschiedene Routing-Algorithmenarten als RAC implementiert werden können. Ein Offline-Algorithmus, wie der bekannte Dijkstra-Algorithmus zum Beispiel, kann die geforderte Route allein aufgrund von statischen Informationen über Start und Ziel schon vor der eigentlichen Simulationsphase berechnen und muss danach die Route-Queue nicht mehr verändern. Im Gegensatz dazu generieren Online-Algorithmen, wie der verwendete BeeJamA-Algorithmus, die Route Schritt-für-Schritt und fügt Kanten während der Simulation dynamisch zur Route-Queue. Abschließend sei an dieser Stelle darauf hingewiesen, dass, die oben erwähnte *tailPosition* eines Fahrzeugs, nicht explizit berechnet werden muss, sondern diese automatisch durch die *Vehicle.setHeadPosition(Position)* berechnet wird.

Ampeln

Wie dem geneigten Leser aus eigener Erfahrung bekannt sein dürfte, stellen Amplen durchaus eine potentielle Stauquelle dar, weswegen es sinnvoll ist Ampelsysteme mit in die Modellierung aufzunehmen. Umgesetzt wurde das deutsche 4-Phasen System, in dem Grün, Gelb, Rot und Gelb-Rot die gültigen Zustände darstellen. Die Abbildung 4.8 stellt das entsprechende Klassendiagramm dar. Jedem *TrafficLight* ist genau eine *Position* zugeordnet, zudem besitzt jede *Lane* eine Liste der sich auf ihr befindlichen Amplen, womit der TMC ein einfacher Zugriff ermöglicht wird. Da jede Ampel einer Zelle zugeordnet ist, hat sie nur lokal Auswirkungen auf die entsprechende Zelle. Demzufolge müssen z.B. auch zwei *TrafficLight*-Instanzen für zwei benachbarte Spuren eingesetzt werden[3]. Wie leicht ersichtlich müssen Ampeln an Kreuzungen synchronosiert werden, um Verkehrsunfälle in der realen Welt und unrealistisches Verhalten in der Simulation zu vermeiden. Beispielsweise sei die in Abbildung 4.9 dargestellte Situation gegeben. Hier ist es wichtig, dass die mit „A" markierten Ampel unbedingt rot sind, wenn die mit „B" markierten Ampelen grün sind. Da die manuelle Synchronisation von Ampeln eine fehlerträchtige Aufgabe darstellen würde, existiert eine Kontrollerklasse die dieses automatisch macht. Um dies umzusetzen, verwaltet die Klasse zwei disjunkte Mengen von sogenann-

[3]Es sei denn die implementierte TMC interpretiert das Vorhandensein einer Ampel auf andere Art.

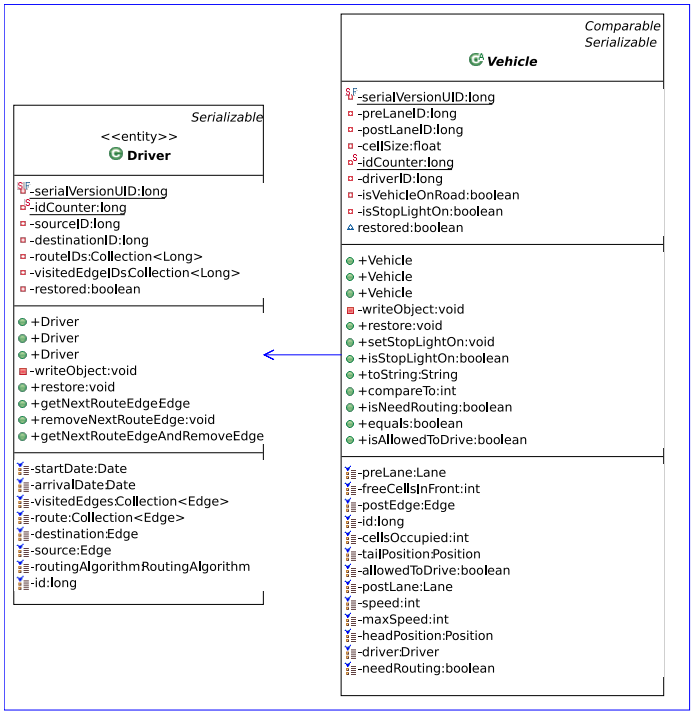


Abbildung 4.7: Die *Driver* und *Vehicle* Entitäten.

ten *positiven* und *negativen* Ampeln. Für jede Kreuzung bzw. andere Straßensituation in der Ampeln synchornisiert sein müssen, wird je eine Instanz dieser Kontrollerklasse benötigt. Dann ist für jede Instanz sichergestellt, dass sich Ampeln aus der *negativen* Menge stets im jeweilig gegenteiligen Zustand befinden als die Ampeln aus der Menge der *positiven* Ampeln. Seien beispielsweise die positiven Ampeln grün, so sind die negativen Ampeln auf rot geschaltet. Tabelle 4.1 listet die möglichen Zustände auf. In der oben beschrieben Situation aus Abbildung 4.9, wäre es eine realistische Annahme, dass die „A"-Ampeln mit den „B"-Ampeln synchronisiert werden. Wählt man die Menge A gleich der Menge der positiven Ampeln und die Menge B als Menge der negativen Ampeln, werden die Amplen synchronisiert geschaltet. Da jedoch die Daten, auf welche Weise Ampeln an Kreuzungen konkret geschaltet sind, öffentlich nicht verfügbar sind, muss diese Entscheidung, wie im Abschnitt 4.12.1 gezeigt, beim Initialisieren der Entitäten heuristisch getroffen werden.

Verkehrsschilder

Verkehrsschilder sind zum jetzigen Zeitpunkt noch nicht implementiert, da die derzeitig implementierte TMC keine Verkehrsschilder unterstützt. Eine Ausnahme stellen die in der Realität vorhandenen Schilder zu Höchstgeschwindigkeiten dar, die im verwendeten Modell jedoch nicht über Verkehrsschilder modelliert sind, sondern als eine Eigenschaft einer Kante angesehen werden.

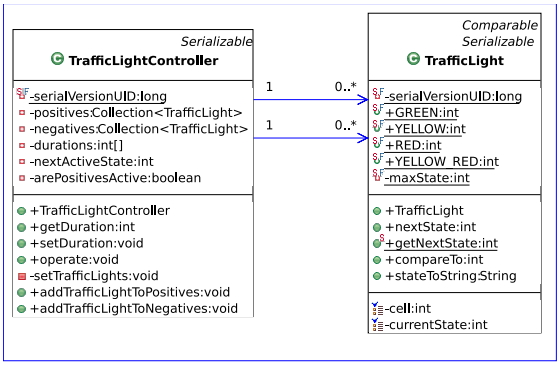


Abbildung 4.8: Die *TrafficLight* Entität und die *TrafficLightController* Klasse.

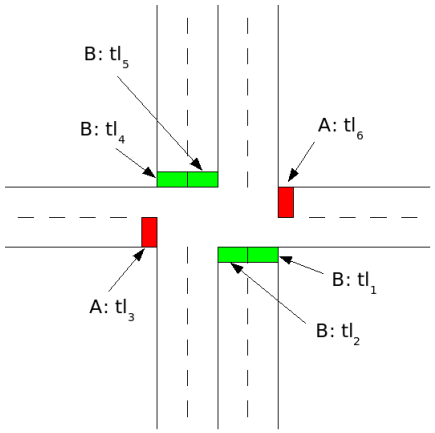


Abbildung 4.9: Ein Beispiel für die Einteilung von Ampeln in disjunkte Mengen A und B zur Synchronisation an Kreuzungen.

Tabelle 4.1: Zustandstransitionen von Ampeln.

Positiv	Negativ
rot	rot
gelb-rot	rot
grün	rot
gelb	rot
rot	rot
rot	gelb-rot
rot	grün
rot	gelb
rot	rot

4.6 Chain Component

Die Chain Component (CC) bildet das Fundament auf dem die Simulator Component (SC) und die Meta Simulator Component (MSC) aufbauen, um eine zeitdiskrete Simulation durchzuführen. Um höchste Modularität der Simulationsumgebung zu gewährleisten wird dazu ein *Operator-Chain*-Konzept umgesetzt. Diese beiden Begriffe seien im Folgenden erläutert.

4.6.1 Operatoren

Eine der wichtigsten nicht-funktionalen Anforderungen an den Simulator stellt sicherlich eine hohe Flexibilität und Modularität dar, soll doch der Quelltext Ausgangspunkt weiterer Arbeiten im Bereich der Stauvermeidung sein. Festzustellen ist, dass der Begriff der Verkehrssimulation im konkreten Sinne sehr heterogen ausgelegt werden kann. Ziel ist es den Simulator so flexibel und modular zu gestalten, dass pro Simulationslauf gänzlich unterschiedliche Anwendungsfälle ausgeführt werden können. So möchte der Benutzer des Simulators möglicherweise alle Simulationsschritte ohne weitere Unterbrechung auf einem Cluster ohne graphische Ausgabe auszuführen, um eine langwierige Vergleichsstudie zwischen zwei Routingalgorithmen durchführen zu können. Andererseits soll es aber auch zu Debugging- oder Vorführzwecken möglich sein, die Simulation Schritt-für-Schritt mit einer entsprechenden Visualisierung des aktuellen Simulationszustandes durchzuführen. Auch soll es vielleicht möglich sein, nach einem Schritt interaktiv einige Parameter anzupassen und die Simulation mit diesen neuen Parametern laufen zu lassen. Ein weiteres Beispiel stellt die Initialisierung dar. Der erste Benutzer möchte einen Datengenerator verwenden, der Fahrzeuge und andere Entitäten vollkommen zufällig generiert. Der zweite Benutzer hingegen will eventuell einen Datenparser für gemessene Daten benutzen und der dritte Benutzer möchte vielleicht diese gemessenen Daten noch heuristisch aufbereiten. Die Anzahl der denkbaren und sinnvollen Kombination aus Subroutinen aus Bereichen wie Initialisierung, Visualisierung, Analyse, Serialisation, Routing und Verkehrsmodellierung ist nahezu unbegrenzt. Der Simulator muss also flexibel genug sein, um all diese Kombinationen zu ermöglichen ohne dabei die Handhabbarkeit zu reduzieren. Um diesen Anforderungen gerecht zu werden, wird ein *Operator-Chain*-Konzept genutzt, um dem Nutzer die entsprechende Handlungsfreiräume einzuräumen einen Simulationslauf beliebig konfigurieren zu können. Ein *Operator* stellt dabei eine gekapselte Subroutine dar, deren Ausführungsreihenfolge durch die weiter unten beschriebenen *Chains* festgelegt ist. Abbildung 4.10 zeigt die abstrakte Klasse *Operator*, sowie die wichtigen Unterklassen *ScheduledOperator* und *ObjectOperator*. Die Basisklasse umfasst das allgemeine Attribut *Name*, sowie die Methode *operate()*, welche als Callback-Methode innerhalb des *Chain*-Konzepts dient (s.u.). Dies bedeutet, dass wenn eine Subroutine, d.h. ein Operator, während der Simulation ausgeführt wird, genau diese Methode angesprungen wird. Demzufolge muss die jeweilige Programmlogik über diese Methode verfügbar sein. Die Unterklasse *ObjectOperator* überschreibt die *operate()*-Methode und fügt einen Parameter vom Typ *Object* hinzu, wodurch der aufrufende Kontext der Methode Informationen über den aktuellen Simulationszustand übergeben kann. Dies wird momentan unter anderem dafür ausgenutzt, um Operatoren in der unten beschriebenen *StepVehicleChain* für jedes einzelne zu simulierende Fahrzeug separat mit dem aktuell betrachteten Fahrzeug aufrufen zu können. Die Klasse *ScheduledOperator* eignet sich für Operatoren, welche nicht in jedem Simulationsschritt ausgeführt werden müssen, sondern nur zu bestimmten Zeiten „aufgeweckt" werden müssen. Abschnitt 7.3.1 im Anhang gibt Beispiele und zeigt wie die drei Arten von Operatoren implementiert werden können.

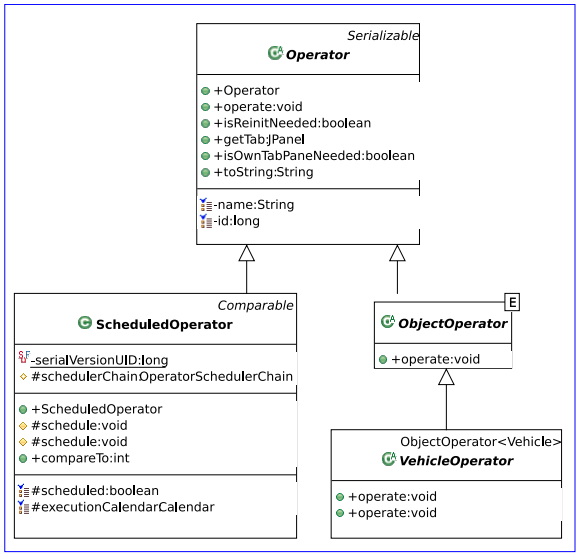


Abbildung 4.10: Die Operatorklassen im Überblick.

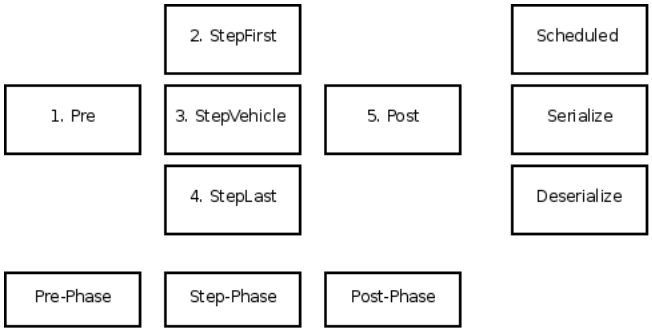


Abbildung 4.11: Übersicht über die Abfolge der *Chains*.

4.6.2 Chains

Eine Chain implementiert ein „*inversion of control*" Entwurfsmuster (vgl. [6]) entsprechend des „Hollywood"-Prinzips[4]. Im vorliegenden Fall, sind die asynchron aufgerufenen Subroutinen des IoC-Entwurfsmusters die oben dargestellten Operatoren. Um eine größtmögliche Flexibilität zu gewährleisten, aber gleichzeitig eine gut überschaubare Struktur zu schaffen, wurde der Simulationsprozeß in drei Phasen unterteilt: die Pre-, Step- und Post-Phase. Diesen drei Phasen sind insgesamt fünf Chains zugeordnet. Hinzu kommen Chains für das Serialisieren und Deserialisiern, sowie für terminierte (*scheduled*) Operatoren. Abbildung 4.12 gibt eine Übersicht. Die Semantik der einzelnen Phasen und der assoziierten Chains seien nun dargestellt:

Pre-Phase: Diese Phase wird genutzt um die simulierten Entitäten zu initialisieren und konfigurieren. Die zugehörige Operator-Chain ist die *preOperatorChain*. Es ist die zuerst ausgeführte Chain.

Step-Phase: Diese Phase repräsentiert die eigentliche Simulation, indem die einzelnen Simulationsschritte ausgeführt werden. Um hier weitere Flexibilität zu gewähren, ist ein einzelner Schritt

[4] „Don't call us, we call you."

wiederum in drei einzelne Subphasen unterteilt:

Step-First-Subphase: Diese erste Subphase ist für Operatoren bestimmt, welche Zugriff auf Entitäten benötigen die im aktuellen Simulationsschritt noch nicht geändert wurden. Die zugehörige *stepFirstOperatorChain* wird momentan hauptsächlich zu Logging-Zwecken benutzt.

Step-Vehicle-Subphase: In dieser zweiten Subphase wird jeder Operator in der *vehicleStepOperatorChain* genau einmal für jedes simulierte Fahrzeug aufgerufen. Folglich ist die implementierte TMC als Operator für diese Chain konzipiert.

Step-Last-Subphase: Analog zur ersten Subphase wird die *stepLastOperatorChain* am Ende eines jeden Simulationschrittes ausgeführt und ist somit gut geeignet für Operatoren die die Veränderungen zum vorhergehenden Schritt benötigen. VCs zum Beispiel können gut als Operator für diese Chain programmiert werden.

Post-Phase: Diese Phase ist das Analogon zur Pre-Phase und somit eignet sich die *postOperatorChain* folglich zum Freigeben von Ressourcen und z.B. zum Speicher von Logs.

(De-)Serialization-Chain: Wird der Simulator während der Ausführung unterbrochen, kann der aktuelle Simulationsstand abgespeichert werden, um zu einem späteren Zeitpunkt genau dort weiter zu simulieren. Die *serializeChain* wird genau in dem Fall der Unterbrechung aufgerufen. Die ggf. dort vorhandenen Operatoren können den aktuellen Simulationsstand dann auf einem beliebigen Medium speichern. Die Umkehrung geschieht durch die *deserializeChain*, wobei anzumerken ist, dass im Falle in dem Operatoren in diese Chain geladen werden die *preChain* nicht ausgeführt wird, um doppelte Initialisierungen zu vermeiden.

Scheduled-Chain: Kann von Operatoren genutzt werden die nicht jeden Schritt aufgerufen werden wollen, sondern nur zu bestimmen Zeiten.

Umgesetzt ist das Konzept durch die Klasse *Chain* und deren Ableitungen (vgl. Abb. 4.12). Die Basisklasse bietet Methoden mit welchen Operatoren in eine Chain eingefügt werden können und diese Operatoren in der Reihenfolge des Hinzufügens während der Simulation ausgeführt werden können. Die Klasse *ChainSequence* kapselt die oben genannten einzelnen Chains und bietet mit der Methode *execute()* die Möglichkeit die Chains gemäß obiger Reihenfolge auszuführen. Hinzu existiert die Methode *ChainSequence.isTerminationConditionReached()*, welche von Unterklassen überschrieben werden kann, um eine Abbruchbedingung für die Iterationen durch die Step-Phase zu beschreiben. Beispielsweise bietet die Unterklasse *StepChainSequence* die einfache Möglichkeit den oben erwähnten Simulationsparameter der maximalen Simulationsschritte als Abbruchbedingung zu verwenden.

4.7 Simulator Component

Die Simulator Component (SC) dient als zentrale Schnittstelle zwischen den einzelnen Komponenten und ermöglicht das Starten und Anhalten von Simulationsläufen. Im Folgenden Abschnitt sei dazu auf die Umsetzung der SC eingegangen und im Anschluss daran, sei auf die aufgetretene Persistenzproblematik eingegangen.

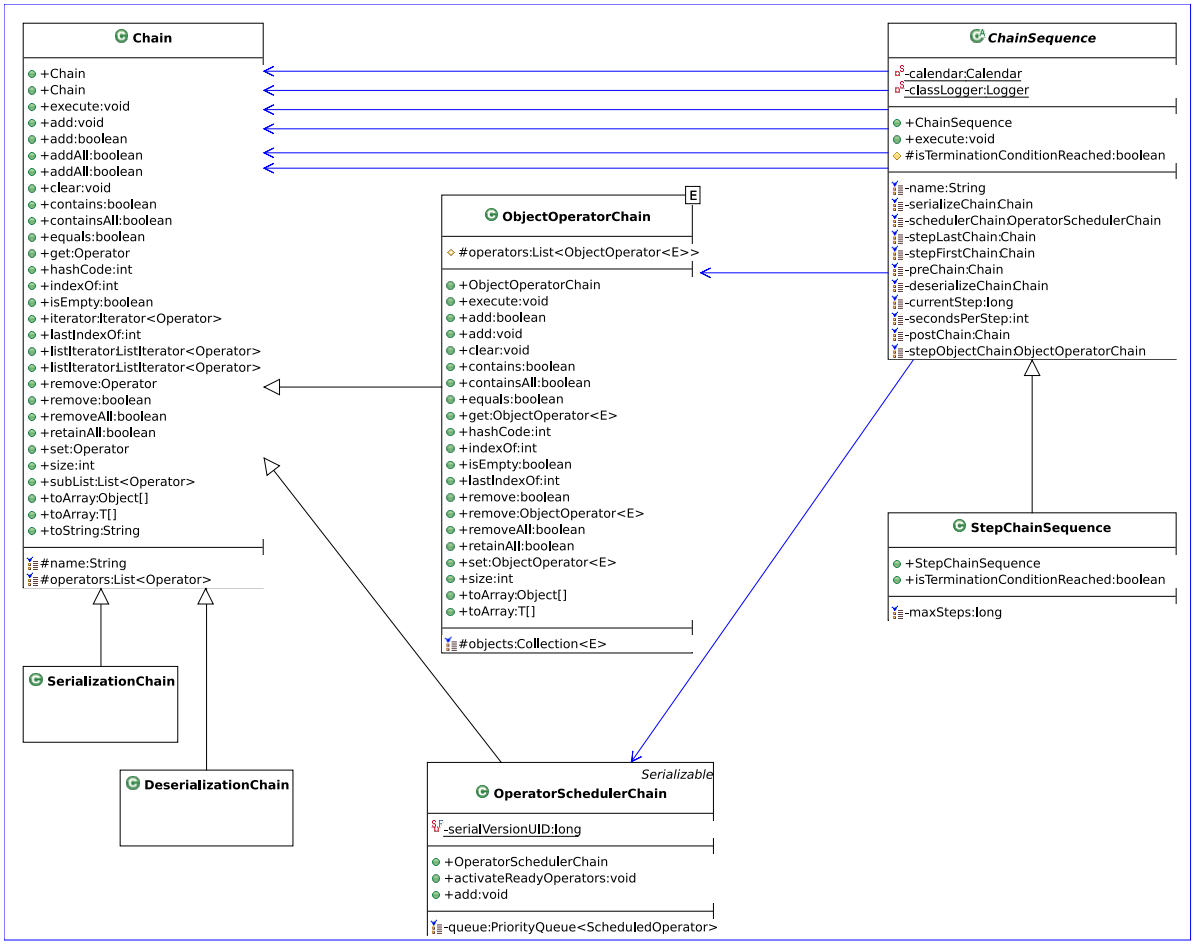


Abbildung 4.12: Die Chain-Klassen im Überblick.

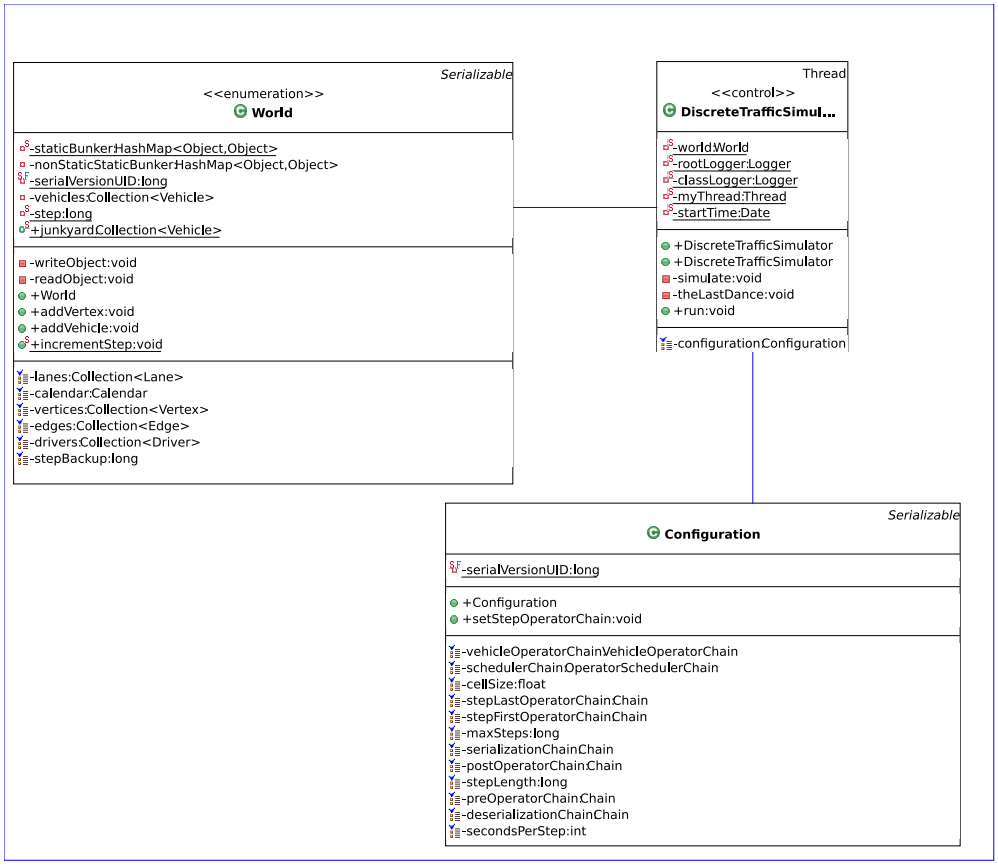


Abbildung 4.13: Die Hauptkontrollerklasse *DiscreteTrafficSimulator*, sowie die beiden Containerklassen *Configuration* und *World*.

4.7.1 Verkehrssimulator

Die zentrale Kontrollerklase *DiscreteTrafficController* ist in Abbildung 4.13 dargestellt und dient dem Starten und Stoppen von Simulationen. Bevor eine Simulation gestartet werden kann, werden jeweils eine Instanz der beiden Containerklassen *Configuration* und *World* benötigt. Erstere dient zum Einstellen der Parameter die sich aus der orts- und zeitdiskreten Arbeitsweise des Simulators ableiten, nämlich die maximale Anzahl der Simulationsschritte und verstrichene Zeit pro Schritt als Zeitparameter, sowie die (globale) Größe einer Zelle als einzigen Ortsparameter. Weiterhin werden in einer *Configuration*-Instanz die ChainSequence wie sie während der Simulation ausgeführt werden soll, gespeichert. Dazu wird dem Konstruktor eine *ChainSequenceFactory*-Instanz übergeben, mit der dann die jeweilige ChainSequence erzeugt wird. Eine *Configuration*-Instanz spiegelt also den statischen Anteil der zu simulierenden Umgebung wieder, eine *World*-Instanz hingegen den dynamischen. So wird die *World*-Kontainerklasse nämlich benutzt um Entitäten, wie Fahrzeuge, Straßennetze und verstrichene Zeit, im jeweils aktuellen Simulationszustand zu speichern und strukturiert zu kapseln. Die *World*-Instanz spiegelt also das aktuelle Weltbild der Simulation wieder, welches sich in der Regel von Simulationsschritt zu Simulationsschritt ändert. Zusammen fassend kann man sagen, dass i.d.R. gilt, dass eine Konfiguration vom Benutzer einmal vor dem Starten der Simulation angelegt wird und danach für den gesamten Simulationslauf unverändert bleibt. Die Eigenschaften der simulierten

Welt hingegen veränderen sich i.d.R. nach jedem einzelnen Simulationsschritt. Vor diesem Hintergrund kann man grob sagen, dass die Welt von der EC bereitgestellt wird, von der HAC initialisiert, verändert von der TMC, analysiert von der RAC und visualisiert wird von der VC. An dieser Stelle sei noch einmal auf die herausragende Stellung des Operator im verwendeten Simulationskonzepts hingewiesen. Jede Änderung die an der zentralen Datenstrukur, der *World*, zur Laufzeit vorgenommen wird, wird exklusiv durch die entsprechenden Operatoren verursacht. Ein entsprechendes Beispiel wie die erwähnten Klassen benutzt werden können, um manuell ein Straßennetz zu modellieren, und eine kleine Simulation mit wenigen Fahrzeugen durchzuführen wird in Abschnitt 7.3.1 im Anhang gezeigt.

4.7.2 Persistenz

Da beliebig lange Zeitspannen simuliert werden können sollen, ist es mitunter wichtig eine einzelne Simulation unterbrechen zu können, um sie später (möglicherweise auf einem anderen Computer) weiterlaufen lassen zu können oder aber um einmal „eingefrorene" Szenarien mit unterschiedlichen Parameter erneut zu simulieren. Die Klasse *DiscreteTrafficSimulator* bietet dazu die entsprechenden Methoden *start(), stop(), interrupt()* und *resume()*. Ziel ist es also den aktuellen Weltzustand abspeichern zu können, im vorliegenden Fall also das aktuelle *World*-Objekt. In Java-Terminologie ist also die die *World*-Instanz zu serialisieren und zu deserialisieren. Java bietet komfortable Möglichkeiten um dieses zu realisieren, jedoch scheitert der rekursive Algorithmus von SUN daran große, stark zyklisch verzeigerte Objektgraphen wie eine *World*-Instanz vom Ruhrgebiet mit realistischen Stackgrößen zu serialisieren (vgl. Bug-ID [17]). Stark zyklisch ist eine solche Instanz in vielerlei Hinsicht, so hält z.B. eine *Edge* eine Liste von *Lanes*, welche wiederum Referenzen auf die *Edge*s hält zu der sie gehören. Diese und weitere zyklische Referenzen sorgen dafür, dass zur Laufzeit eine „stack overflow" Ausnahme geworfen wird, wenn versucht wird eine Welt zu serialisieren die aus mehr als einigen wenigen Entitäten besteht. Das Problem das es zu lösen gilt, besteht also darin, einerseits weiterhin objektorientiert zu sein, wodurch sich ein Ersetzen der Objekte durch primitive ID-Referenzen[5] verbietet, als auch Persistenz überhaupt zu ermöglichen. Der gewählte Lösungsweg erhält die Objekte während der Simulation, markiert sie aber als *transient*[6] und fügt zusätzlich aber auch Stellvertreter-IDs ein. Aus

```
1         private Edge parent;
```

wird also beispielsweise

```
1
2         transient private Edge parent;
3
4         /** Serializable ID reference of parent. */
5         private long parentID;
6
7         //..
8
9         private void writeObject(ObjectOutputStream oos) {
10
11                if (parent != null)
12                        parentID = parent.getId();
13
14                // ...
15
16                try {
17                        oos.defaultWriteObject();
18                } catch (IOException e) {
19                        e.printStackTrace();
20                }
21        }
```

[5] Jede Entität verfügt von Hause aus über eine ID vom Typ *long*.

[6] Das Java-Schlüsselwort, um Variablen vom Serialisieren auszuschließen.

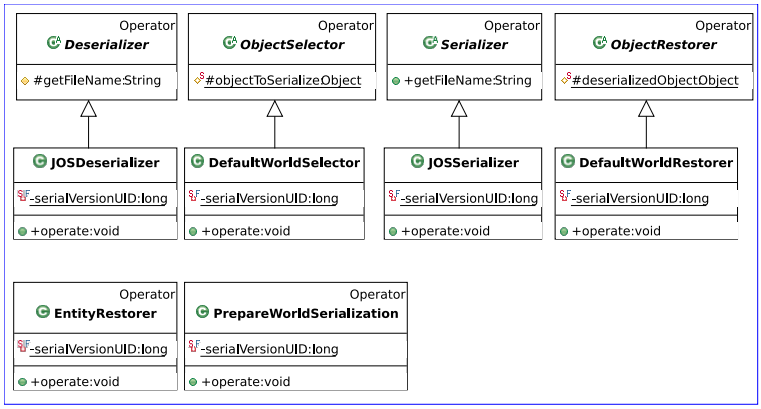


Abbildung 4.14: Die Persistenzklassen im Überblick.

Hinzu kommen die jeweiligen Getter- und Setter-Methoden. Im Falle des Serialisierens wird nun die *serializeChain* durchlaufen. Ist dort der *JOSSerializer*[7] registriert, wird die Methode *writeObject()* aller Entitäten durch das JOS-Framework aufgerufen (da die Klassen das Interface *Serializable* implementieren) und somit kurz vor dem Serialisieren der Entität die Referenzen durch primitive ID-Referenzen ersetzt. Beim Deserialisieren funktioniert der Mechanismus analog andersherum. Dabei sind alle Entitäten einzulesen, wiederherzustellen (bis auf ihre Objektreferenzen) und dann anhand der IDs die gewünschten Objekte herauszusuchen und die Objektreferenz auf sie zu restaurieren. Die Abbildung 4.14 zeigt die bisher existierenden Klassen für die Aufgaben rund um das Serialisieren und das Deserialisieren. Die Klassen bedienen sich dem Operator-Konzept und passen sich dank der Serialize- und Deserialize-Chain gut in das Gesamtkonzept ein. Mittels Vererbung können neue Serialisierungsverfahren leicht implementiert werden. Die Klassen der *ObjectSelector*-Hierarchie haben die Aufgabe das zu serialisierende Objekt auszuwählen. Das Default-Derivat wählt dabei naheliegenderweiße die aktuelle *World*-Instanz aus. Würde mittels eines Operators eine Historie der Welt angelegt, könnte beispielsweise die Historie durch einen entsprechenden Selektor statt nur des aktuellen Zustandes ausgewählt werden. So wäre es leicht möglich eine komplette Simulation aufzuzeichnen und später z.B. als Demonstration durch die VC abspielen zu lassen. Sinn würde dies z.B. in dem Fall machen, in dem eine Simulation unbeaufsichtigt auf einem Cluster ausgeführt wird und nur später bei erfolgreichen Ergebnissen näher untersucht werden soll. Das Serialisieren selbst übernehmen die Klassen der *Serialize*-Hierarchie, das Deserialisieren wird analog von den von *Deserializer* abgeleiteten Klassen durchgeführt und schlussendlich die oben beschriebene Restaurierung der Objektreferenzen durchgeführt. Zusammenfassend ist zu sagen, dass ein großer Aufwand, insbesondere durch die Modifikation aller Entitätenklassen, getätigt werden musste, um das Serialisieren mittels des JOS-Frameworks zu ermöglichen. Gemäß SUN-Bug-Datenbank-Einträgen ist leider auch in Zukunft nicht davon auszugehen, dass große zyklische Objektgraphen standardmäßig unterstützt werden.

Der Konfigurationsparser

Ein möglicher Ansatz zum Speichern und Laden von Konfigurationen besteht darin, die jeweilige Konfiguration in einer XML-Datei zu speichern und diese aus der XML-Datei zu rekonstruieren. Um dieses zu bewerkstelligen wird für jede Konfiguration eine eigene XML-Datei erzeugt, in der alle Ei-

[7] Für *Java Object Stream Serializer*.

genschaften der Konfiguration und der Operatoren gespeichert werden. Aus den XML-Dateien lassen sich durch das Auslesen der entsprechenden Eigenschaften die jeweiligen Ausgangskonfigurationen rekonstruieren. Eine Konfiguration enthält verschiedene Eigenschaften durch die eine Simulation beschrieben wird. Diese bestehen im Wesentlichen aus allgemeinen Eigenschaften wie zum Beispiel der Anzahl der Simulationsschritte und der Zugehörigkeit der Operatoren zu den jeweiligen Chains. Neben der Konfiguration enthalten aber auch die einzelnen Operatoren Eigenschaften, durch die die Simulation beeinflusst wird. Daher ist es notwendig sowohl die direkten Eigenschaften der Konfiguration als auch die operatorspezifischen Eigenschaften zu speichern. Eine Eigenschaft stellt im Wesentlichen eine vom Benutzer veränderbare größe dar, durch die sowohl das Simulationsverhalten, als auch das Verhalten verschiedener Komponenten beeinflusst wird. Wie bereits oben erläutert wurde, handelt es sich bei den Eigenschaften genau um die größen, die vor der Simulation vom Benutzer statisch festgelegt werden können und sich während der gesamten Simulation nicht ändern. Anhand des folgenden XML-Schemas wird der Aufbau der XML-Dateien festgelegt:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<xsd:element name="configuration" type="configurationType"/>

<xsd:complexType name="configurationType">
    <xsd:sequence>
        <xsd:element name="operators" type="operatorsType"/>
        <xsd:element name="configurationVariables" type="configurationVariablesType"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="operatorsType">
    <xsd:sequence>
        <xsd:element name="operator" type="operatorType" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="operatorType">
    <xsd:sequence>
        <xsd:element name="operatorType" type="xsd:string"/>
        <xsd:element name="chainPath" type="xsd:string"/>
        <xsd:element name="chainName" type="xsd:string"/>
        <xsd:element name="operatorVariables" type="operatorVariablesType"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="operatorVariablesType">
    <xsd:sequence>
        <xsd:element name="operatorVariable" type="operatorVariableType"
         maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="operatorVariableType">
    <xsd:sequence>
        <xsd:element name="operatorVariableName" type="xsd:string"/>
        <xsd:element name="operatorVariableType" type="xsd:string"/>
        <xsd:element name="operatorVariableValue" type="xsd:string"/>
    </xsd:sequence>
```

```
</xsd:complexType>

<xsd:complexType name="configurationVariablesType">
    <xsd:sequence>
        <xsd:element name="configurationVariable" type="configurationVariableType"
        maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="configurationVariableType">
    <xsd:sequence>
        <xsd:element name="configurationVariableName" type="xsd:string"/>
        <xsd:element name="configurationVariabletype" type="xsd:string"/>
        <xsd:element name="configurationVariableValue" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>
```

Um nach diesem XML-Schema eine XML-Datei erzeugen zu können, in der die Eigenschaften der Konfiguration und der Operatoren gespeichert werden, müssen die Variablen, welche die Eigenschaften darstellen, als solche kenntlich gemacht werden. Dafür muss es für jede Variable, welche eine Eigenschaft symbolisieren soll, entsprechende *Getter* und *Setter* Methoden geben. Beim schreiben in die Datei wird in der Konfiguartion und den Operatoren nach genau solchen Kombinationen aus Variable und *Getter* und *Setter* Methoden gesucht und der Wert einer Variablen, welche der Vorgabe für eine Eigenschaft entspricht, in die Datei geschrieben. Wie aus dem XML-Schema hervorgeht, enthält die XML-Datei neben den Werten der als Eigenschaft kenntlich gemachten Variablen die Zugehörigkeit der Operatoren zu den jeweiligen Chains, da auch diese eine Eigenschaft ist und sich von Konfiguration zu Konfiguration ändern kann.
Nachdem eine XML-Datei erzeugt wurde, kann aus dieser automatisch eine Simulation zu generieren. Um die Ausgangskonfiguration zu rekonstruieren wird zunächst eine neue Konfiguration erzeugt. Danach wird die entsprechende XML-Datei mittels *JDOM* eingelesen, und als erstes die Eigenschaften der Konfiguration gesetzt. Darauf hin werden die Chains und die Operatoren mittels Reflexion erzeugt, die Eigenschaften der Operatoren gesetzt und diese in die jeweiligen Chains eingefügt.
Die Hauptproblematik der Implementierung besteht darin, dass der Konfigurationsparser sehr unflexibel ist. So ist es zum Beispiel notwendig, dass jeder Operator direkt von der Klasse *Operator.java* erbt. Dieses Problem geht aus der Operator Instanziierung mittels Reflexion hervor, da alle zu instantiierenden Klassen die gleiche Vererbungshierarchie aufweisen müssen. Andernfalls ist eine Instantiierung mittels Reflexion nicht möglich, es sei denn es werden Fallunterscheidungen für bestimmte Operatortypen eingefügt, wodurch die Flexibilität sicherlich nicht erhöht wird. Selbiges gilt auch für die verschiedenen Arten der Chains, diese müssen direkt von der Klasse *Chain* erben.
Des weiteren lassen sich lediglich die statischen Eigenschaften der weiter oben beschriebenen *World*, in Form der Konfiguration, speichern. Damit stellt der Konfigurationsparser keine Lösung für das beschriebene Persistenzproblem dar.

4.8 Metasimulator Component

Mit dem oben vorgestellten Konzepten ist es möglich einen einzelnen Simulationslauf zu konfigurieren und durchzuführen. Sollen nun mehrere Simulationsläufe hintereinander gestartet werden ist dies

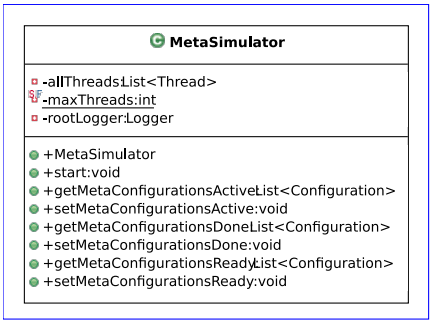


Abbildung 4.15: Hauptklasse des Metasimulators.

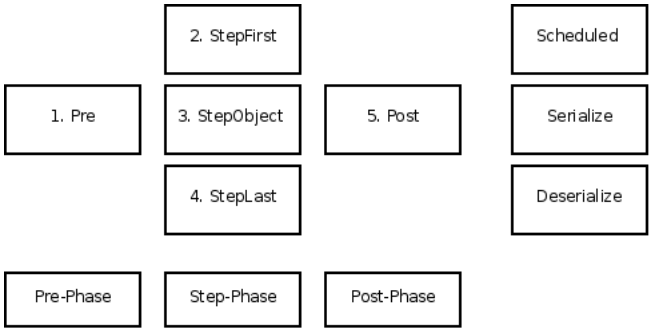


Abbildung 4.16: Übersicht über die Abfolge der *Chains* im Metasimulator.

nur manuell (bzw. durch Skripte) möglich. Der hier vorgestellte Metasimulator benutzt das Chain-Konzept, um einen verallgemeinerten Simulator zu konfigurieren, in dem ein einzelner Simulationsdurchlauf nur ein Operator in einer Chain ist und somit mehrere Durchläufe automatisch hintereinander durchgeführt werden können. Die Klassendiagramm der MSC-Hauptklasse ist in Abbildung 4.15 zu sehen. Dabei wird dieselbe Chain-Anordnung wiederverwendet wie sie oben bereits in Abbildung 4.12 dargestellt wurde, nun jedoch wird anstelle der spezifischen *StepVehicle*-Chain die allgemeine *StepObject*-Chain verwendet (siehe Abbildung 4.16). Damit ist es dann beispielsweise leicht möglich ein und dasselbe Simulationsszenario mit unterschiedlichen Parametern zu starten. Iterative Mehrfachausführung von Simulation, z.B. für statistische Auswertungen, lassen sich so ebenfalls leicht programmieren. Erleichtert wird dies durch den Batch-Modus des Metasimulators. Wird ein Job dem Batch-System zugeführt, wird er ausgeführt sobald eine CPU dafür frei ist. Jedem Job wird dabei genau eine CPU zugeordnet bis er abgearbeitet ist (vgl. Abb. 4.17). Mehrere CPUs pro Job würden keinen Geschwindigkeitsvorteil bringen, da die eigentliche Simulation monolithisch abläuft (siehe Abbildung 4.3). Genauer gesagt wird jedem Job ein eigenständiger Java-Thread zugeordnet und die maximale Anzahl der Threads entspricht der Anzahl der CPUs[8]. Ein Job besteht hierbei aus einer Instanz der Klasse *Configuration* wie bereits oben vorgestellt (vgl. Abschnitt 4.7). Das Hinzufügen von Operatoren wird wieder durch entsprechende Factory-Klassen durchgeführt. Beispielhaft sei hier der Aufbau der *ChainSequence* wie sie von der Klasse *EAChainSequenceFactory* generiert wird, erläutert. Abbildung 4.18 gibt einen Überblick. Ziel dieser Konfiguration ist es mittels Evolutionären Algorithmen Simulationsparameter zu optimieren. Durch die Verwendung der JGAP-Bibliothek, welche ein

[8]Gemeint sind hierbei die CPUs des lokalen Computers. Prinzipiell könnten leicht Operatoren erstellt werden, welche die eigentliche Simulation auf entfernten Rechner starten, dies ist allerdings bislang nicht implementiert.

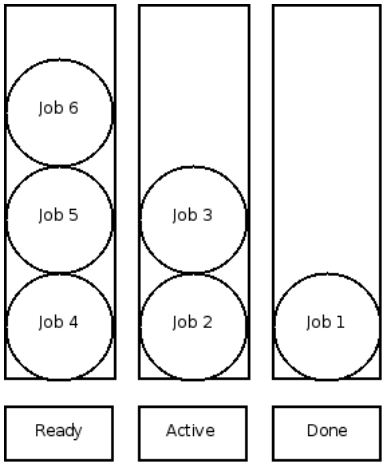


Abbildung 4.17: Beispiel für das Batch-Verhalten der MSC. Neue Jobs werden in die Ready-Queue aufgenommen, während der Abarbeitung befinden sie sich in der Active-Queue und schließlich in der Done-Queue.

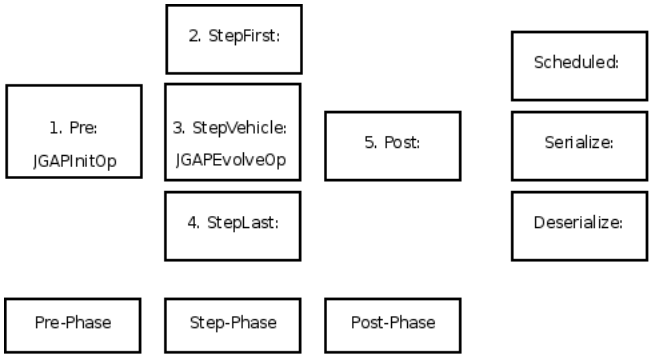


Abbildung 4.18: Verwendete Operatoren zur Parameteroptimierung.

vollständiges Framework zur Entwicklung von Evolutionären Algorithmen bereitstellt, werden insgesamt nur zwei Operatoren benötigt. Der erste Operator (*JGAPInitOperator*) aus der *preChain* initialisiert das gesamt JGAP-Framework und legt insbesondere die verwendete(n) Fitnessfunktion(en) fest. Im aktuellen Entwicklungsstand wird hierbei nur eine einzige Fitnessfunktion verwendet, d.h das mehrkriterielle Stauvermeidungsproblem wird der Einfachheit halber auf ein einziges Kriterium, und zwar der akkumulierten individuellen Reisezeit[9], reduziert. Optimiert werden soll im vorliegenden einfachen Fall der in Abschnitt 2.4.2 beschriebenen Thresholdwert, d.h. die Population besteht aus Individuen von Fließkommazahlen zwischen 0 und 1. Der *JGAPEvolveOperator* evoliert nun pro Simulationsschritt die aktuelle Generation genau einmal. Insbesondere heißt dies, dass die Fitnessfunktion für jedes Individuum genau einmal aufgerufen wird, d.h. für jeden Threshold in der aktuellen Population wird eine vollständige Simulation durchgeführt. Die Fitnessfunktion bewertet dann diejenigen Thresholds besser, welche zu einer niedrigeren akkumulierten Reisezeit führen. Da es sich hier ausschließlich um ein Beispiel zur Verdeutlichung der Einsatzmöglichkeiten des Metasimulators handelt und die Entwicklung des Metasimulators noch ganz am Anfang steht, sei hier nicht auf weitere Details der verwendeten EAs, wie Selektion und Mutation, eingegangen. Aus gleichen Gründen sei an dieser Stelle auch die Problematik ausgeklammert, dass aufgrund der stochastischen Natur der durchgeführten Simulation, eine wiederholte Fitnessfunktionsauswertung im Allgemeinen nicht zum selben Resultat führen muss, d.h. nicht robust ist. Es sei jedoch darauf hingewiesen, dass der Einsatz von Racing-Algorithmen an dieser Stelle lohnenswert sein kann.

4.9 Visualisierungskomponente

Um ein einfaches und direktes testen der in Kapitel 4.11 dargestellten Routingalgorithmen zu ermöglichen, wurde ein Werkzeug zur Unterstützung des Algorithmenentwurfs in Form der Visualisierungskomponente entwickelt. Durch die Darstellung des zugrunde liegenden Graphen und der Fahrzeuge, welche auf diesem geroutet werden, wird es dem Entwickler möglich, in einer einfachen und anschaulichen Weise Auswirkungen von Parameteränderung innerhalb eines Algorithmus während der Entwicklung zu bewerten und gegebenen falls Änderungen vorzunehmen.
Die folgende Darstellung der Implementierung der Visualisierungskomponente nimmt eine eher theoretischen Sichtweise ein, in der es vor allem um anfängliche Problematiken, deren Lösungen und die Strukturierung der Visualisierungskomponente geht. Zur eigentlichen Implementierung der Visualisierungskomponente sei lediglich gesagt, dass sie wie der gesamte Simulator, in Java programmiert ist. Für die graphische Darstellung der Straßen und Fahrzeuge (siehe Kapitel 7.2.2) wurde JoGL, eine Java OpenGL-Programmbibliothek, verwendet. Daher ist es unumgänglich JoGL vor der Benutzung der Visualisierungskomponente zu installieren. Informationen zur Installation von JoGL finden sich im Kapitel 7.1. Genauere Informationen zur Implementierung können dem package *edu.udo.cs.pg502.simulator.visualization* entnommen werden.

4.9.1 Aufbau und theoretische Aspekte der Visualisierung

Gegenstand dieses Kapitels ist sowohl die Darstellung der grundlegenden Ideen als auch die Erläuterung des Aufbaus der Visualisierungskomponente. Das in Abbildung 4.19 gezeigte Klassendiagramm der Visualisierungskomponente, welches sich im package *edu.udo.cs.pg502.simulator.visualization*

[9] $\sum_{i=0}^{n} RZ_i$, mit n als Anzahl der Fahrzeuge und RZ_i als Reisezeit des Fahrzeugs i.

befindet, stellt den Ausgangspunkt der Erläuterung dar, darüber hinaus dient es der Gliederung dieses Kapitels. Im Folgenden werden sowohl die Funktionalitäten der einzelnen Klassen als auch die

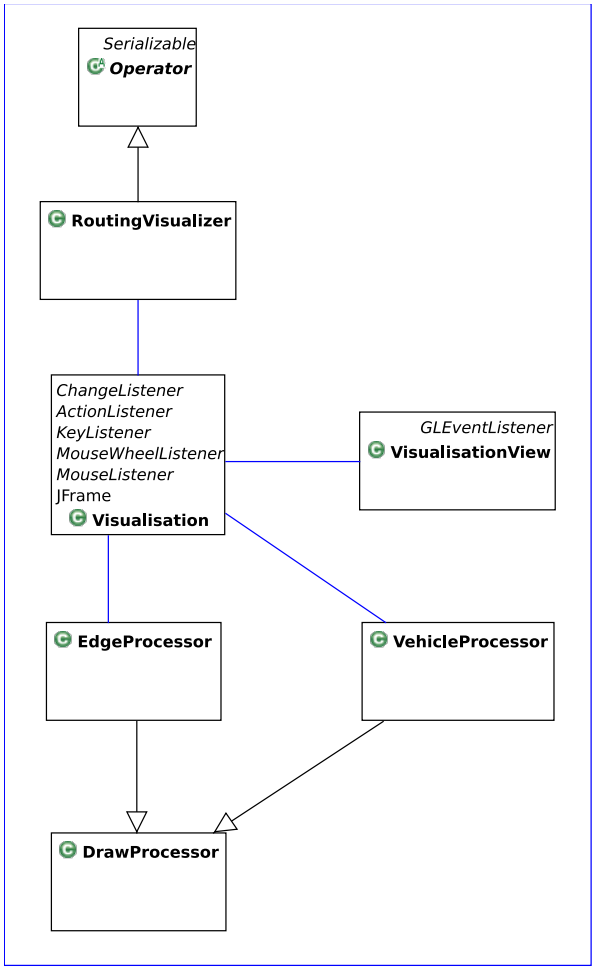


Abbildung 4.19: Klassendiagramm der Visualisierungskomponente

Abhängigkeiten zwischen diesen betrachtet, wobei die Operator Klasse nur der Vollständigkeithalber mit in das Diagramm aufgenommen wurde. Genauere Informationen bezüglich der Operator Klasse können Kapitel 4.6.2 entnommen werden. Da die Schwierigkeit der Visualisierung in der Berechnung der Kanten- und Fahrzeugpositionen im kartesischen Koordinatensystem liegt, macht die Darstellung der Funktionalitäten der entsprechenden Klassen (*EdgeProcessor* und *VehicleProcessor*) den Hauptteil dieses Kapitels aus.

RoutingVisualizer

Dem Operatorkonzept folgend, steht auch für die Visualisierungskomponente ein Operator zur Verfügung, welcher durch die Klasse *RoutingVisualizer* definiert ist. Der Operator der Visualisierung ist entweder in der *StepFirstOperatorChain* oder der *StepLastOperatorChain* zu platzieren (vgl. Kap. 7.3.1), da der Operator in jedem Simulationsschritt aufgerufen werden muss, um die Berechnung der Fahrzeugpositionen und Kantenfarben zu veranlassen. Des weiteren werden alle für die Visualisierungskomponente

benötigten Klassen durch den Operator instantiiert, daher genügt es den Operator in eine der genannten Chains einzufügen, um die Visualisierungskomponente zu benutzen.

Visualization

Die Klasse *Visualization* erzeugt die im Kapitel 7.2.2 dokumentierte graphische Benutzerschnittstelle. Dabei kommen lediglich Java-Swing Komponenten zum Einsatz; auf eine genauere Darstellung der Implementierung der GUI wird an dieser Stelle verzichtet und auf den Quellcode der Klasse Visualization im oben genannten package verwiesen. Des Weiteren dient die Klasse *Visualization* als Schnittstelle zwischen den berechnenden Klassen *VehicleProcessor* bzw. *DrawProcessor* und der zeichnenden Klasse *VisualizationView*.

VisualizationView

Die Klasse *VisualizationView* beinhaltet verschiedene OpenGL Methoden zum Zeichnen des Straßennetzes und der Fahrzeuge. Im Wesentlichen werden durch diese Klasse lediglich vorher erzeugte Datenstrukturen ausgelesen und entsprechend graphisch dargestellt. Aufgrund der Trennung von Darstellung und Berechnung finden in dieser Klasse keine Berechnungen statt. Alle in der Klasse VisualizationView verwendeten OpenGL-Methoden sind ausführlich in [11] dokumentiert, daher wird auf eine genauere Darstellung verzichtet.

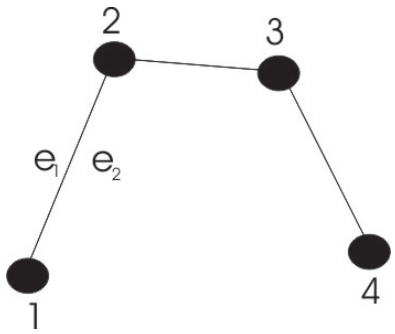
DrawProcessor

Wie oben schon angedeutet wurde, wird zwischen berechnenden Klassen in Form von *EdgeProcessor* und *VehicleProcessor* und der zum Zeichnen benutzten Klasse *VisualizationView* unterschieden. Die berechnenden Klassen stellen der zeichnenden Klasse alle zum Zeichnen benötigten Informationen zur Verfügung. Durch diese Trennung wird eine Modularität erreicht, was zur Folge hat, dass sich einzelne Komponenten relativ einfach durch andere ersetzen lassen. Ein Beispiel dafür ist der Austausch von OpenGL und Java3D. Die Klasse *DrawProcessor* dient als Obertyp der berechnenden Klassen *VehicleProcessor* und *EdgeProcessor*. Dadurch wird sowohl ein enger semantischer Zusammenhang zwischen denen im Folgenden erklärten berechnenden Klassen hergestellt, als auch grundlegende Funktionen für diese bereitgestellt.

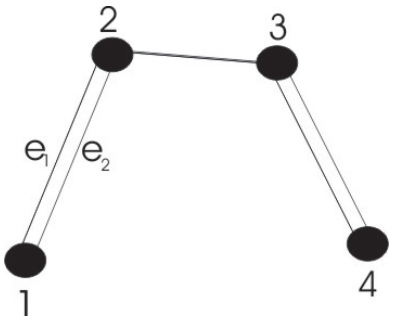
EdgeProcessor

Die Klasse *EdgeProcessor* stellt im wesentlichen alle relevanten Daten zusammen, welche von der Klasse *VisualizationView* zum Zeichnen des Straßennetzes benutzt werden. Aufgrund des vorliegenden Datenformates (vgl. Kap. 3.1), durch das das zu Zeichnende Straßenetz beschrieben wird, müssen die Daten angepasst werden, um eine adäquate graphische Darstellung des Straßennetzes zu ermöglichen. Wie im folgenden gezeigt wird, besteht die Problematik in der Beschreibung der einzelnen Fahrtrichtungen einer Straße, da aus den Datensätzen keine differenzierten Informationen über die Positionen der einzelnen Sraßenseiten einer Straße im karthesischen Koordinatensystem gewonnen werden können. Es ist lediglich bekannt, ob eine Straße eine oder zwei Straßenseiten hat, bzw. ob sie uni- oder bidirektional ist.

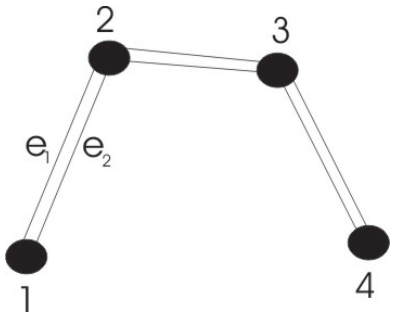
Aus Kapitel 4.12 geht hervor, dass die Position einer Kante im Koordinatensystem durch eine geordnete Liste von Intermediates beschrieben wird, die Kanten selbst stellen die jeweiligen Fahrtrichtungen einer Straße bzw. eines Straßenstücks dar. Dem zufolge wird eine Straße durch eine oder zwei Kanten repräsentiert, abhängig davon, ob die Straße uni- oder bidirektional ist. Das bedeutet, dass jede Fahrtrichtung bzw. Straßenseite einer Straße durch genau eine Kante bestimmt wird. Allerdings hat sich diese Repräsentation der Straßen für die graphische Darstellung der einzelnen Fahrtrichtungen als problematisch erwiesen. Die Problematik besteht darin, dass die Intermediatelisten der Kanten einer bidirektionalen Straße die gleichen Intermediates enthalten. Die Richtung einer Straßenseite im Koordinatensystem wird dabei durch die Reihenfolge der Intermediates in der geordenten Liste festgelegt. Dem zu Folge enthalten die geordneten Listen der Intermediates der beiden Kanten einer bidirektionalen Straße die gleichen Intermediates in entgegengesetzter Reihenfolge. Gerade durch diese Darstellung stehen keine differenzierten Infomationen über die Positionen der einzelnen Straßenseiten einer Straße im Koordinatensytem zur Verfügung. Eine direkte graphische Darstellung der Kanten hätte damit zur Folge, dass eine visuelle Unterscheidung der einzelnen Fahrtrichtungen einer bidirektionalen Straße nicht mehr möglich wäre, da die beiden Kanten, durch die die einzelnen Straßenseiten beschrieben werden, übereinander gezeichnet würden. Um dieses Problem zu umgehen und eine differenzierte Betrachtung der einzelnen Straßenseiten zu ermöglichen, ist eine Verschiebung der Kanten notwendig. Abbildung 4.20(a) zeigt eine Straße, die durch zwei Kanten e_1 und e_2 beschrie-



(a) Ohne Verschiebung



(b) Verschiebung der gesamten Kante



(c) Verschiebung einzelner Kantenabschnitte

Abbildung 4.20: Problematik der Kantenverschiebung

ben wird. Die Intermediates der Kanten entsprechen den schwarzen, nummerierten Kreisen. Daraus folgt, dass die Kanten e_1 und e_2 durch die geordneten Listen von Intermediates $l_{e_1} = \{1, 2, 3, 4\}$ bzw. $l_{e_2} = \{4, 3, 2, 1\}$ beschrieben werden. Die Kante e_1 führt dem zu Folge vom Intermediate 1 zum Intermediate 4 und die Kante e_2 vom Intermediate 4 zum Intermediate 1. An dieser Stelle sei nochmals

darauf hingewiesen, dass sich Intermediates von den oben beschriebenen Knoten unterscheiden. Knoten enthalten keine Informationen über die Positionen der Kanten im Koordinatensystem, sie dienen vielmehr dem logischen Aufbau des Straßennetwerks (vgl. Kap. 3.1). Theoretisch könnte die Richtungsbestimmung einer Kante auch über die jeweiligen Anfangs- und Endknoten der Kante erfolgen, allerdings hat sich zum Zeichnen die Richtungsbestimmung der Kanten, durch die Betrachtung der Intermediatereihenfolgen, als praktischer erwiesen.
Durch Abbildung 4.20 wird die Problematik der ursprünglichen Kantenbeschreibung durch eine geordnete Liste von Intermediates verdeutlicht, da die Kanten e_1 und e_2 übereinander liegen. Abbildung 4.20(b) zeigt die gleiche Straße noch mal, allerdings wurden hier die Kanten im gesamten verschoben. Auch dieses Vorgehen ist nicht ganz unproblematisch, da die Darstellung der einzelnen Straßenseiten relativ ungenau ist. So liegen zum Beispiel die Kanten zwischen den Intermediates 2 und 3 übereinander. Der Grund für dieses Problem liegt darin, dass die Kanten als ganzes um einen festen Wert verschoben wurden und es so, gerade bei stark knickenden Kanten, zu Überschneidungen kommt. An dieser Stelle hat es sich als hilfreich erwiesen, die Kanten in kleinere Teilstücke zu zerlegen, welche dann einzeln, abhängig von der Lage im Koordinatensystem, um einen bestimmten Wert verschoben werden. Wie Abbildung 4.20(c) zeigt, lassen sich durch die Zerlegung der Kanten Überschneidungen vermeiden. Die Kante e_1 wurde zum Beispiel zunächst in die drei, durch jeweils zwei Intermediates festgelegte Teilstücke $\{1, 2\}$,$\{2, 3\}$ und $\{3, 4\}$ zerlegt, welche dann individuell verschoben wurden. Auf die Berechnung der Verschiebung wird im weiteren Verlauf genauer eingegangen. Zunächst soll die Kantenzerlegung genauer beschrieben werden.
Da durch die Zerlegung der Kanten neue Informationen bezüglich der Kantenteilstücke entstehen wird eine neue Datenstruktur benötigt, die diese Informationen aufnehmen kann. Die verwendete Datenstruktur ist beispielhaft in Abbildung 4.21 dargestellt ist. Als Basis der Datenstruktur dient ein Vektor

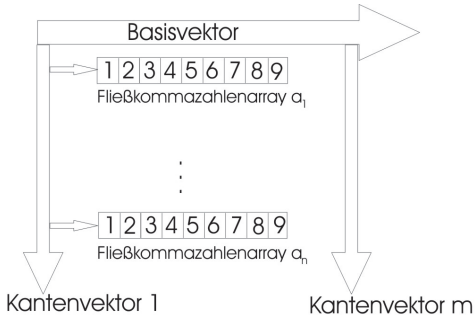


Abbildung 4.21: Datenstruktur der Straßen

(siehe Abbildung 4.21, Basisvektor), welcher eine Art Containerfunktion hat, da dieser für jede Kante einen Vektor aufnimmt (siehe Abbildung 4.21, Kantenvektor). Das bedeutet, dass zunächst für jede Kante ein eigener Vektor erzeugt wird, welcher in den Basisvektor eingefügt wird. Damit entspricht die Größe des Basisvektors der Kantenanzahl. In welcher Reihenfolge die einzelnen Vektoren, welche die Kanten repräsentieren, in den Basisvektor eingefügt werden, spielt an dieser Stelle keine Rolle.
Die Vektoren der einzelnen Kanten bestehen aus unterschiedlich vielen Fließkommazahlenarrays, welche die eigentlichen Informationen, in Form von Koordinaten, zum Zeichnen beinhalten. Wie oben schon angedeutet wurde, werden die einzelnen Kantenstücke über die zur Kante gehörigen Intermediates gebildet. Der Zusammenhang zwischen Kantenstücken und Arrays besteht darin, dass jedes Array ein Kantenstück beschreibt, in dem für jedes aufeinander folgende Paar von Intermediates einer Kante ein Array zu dem entsprechenden Kantenvektor hinzugefügt wird. Neben weiteren Informa-

tionen beinhalten die Arrays die Koordinaten der Intermediatepaare, durch die sie gebildet werden. Dabei dient der erste Intermediate des Paares als Startpunkt und der zweite als Endpunkt des Kantenstücks. Nach der Zerlegung entsprechen die einzelnen Kantenvektoren den geordnete Listen von Intermediates, welche durch die einzelnen Arrays ersetzen werden.
Auf die Bedeutung der einzelnen Arrayelemente wird später eingegangen; zunächst wird die Zerlegung der Kanten in kleinere Stücke formalisiert. Dazu seien $i_1 ... i_n$ die Intermediates einer Kante e und $L_e = \{i_1, i_2, ..., i_{n-1}, i_n\}$ die zugehörige geordnete Liste der Intermediates, des weiteren bezeichne n die Anzahl der Intermediates der Liste L_e. Gemäß der Zerlegung wird für jedes aufeinander folgende Paar von Intermediates $\{i_j, i_{j+1}\}$, mit $1 \leq j$ und $j + 1 \leq n$, der Liste L_e genau ein Array a_j in den Vektor der Kante e eingefügt. Damit enthält der Vektor der Kante e genau $n - 1$ Arrays, welche im Folgenden mit $a_1 ... a_{n-1}$ bezeichnet werden. Um die Reihenfolge der Intermediates zu bewahren, welche durch die geordnete Liste festgelegt ist, beginnt die Zerlegung beim ersten Intermediatepaar $\{i_1, i_2\}$ und der damit verbundenen Erzeugung des Arrays a_1, falls die Liste der Kante mehr als zwei Intermediates enthält, wird danach mit dem Intermediatepaar $\{i_2, i_3\}$ und der Erzeugung des Array a_2 fortgesetzt, usw. Auf diese Weise wird jede Kante über die zugehörigen Intermediates in Teilstücke zerlegt, in dem der erste Intermediate (i_j) eines Paares als Anfangspunkt und der zweite (i_{j+1}) als Endpunkt des Kantenteilstücks dient, welches durch das Array a_j beschrieben wird.
Die einzelnen Arrays bestehen aus neun Fließkommazahlen, deren Bedeutung durch die folgende Liste beschrieben wird:

1. x-Koordinate des Anfangspunktes eines Kantenteilstücks (bzw. des ersten Intermediate des Intermediatepaares)
2. y-Koordinate des Anfangspunktes eines Kantenteilstücks (bzw. des ersten Intermediate des Intermediatepaares)
3. x-Koordinate des Endpunktes eines Kantenteilstücks (bzw. des zweiten Intermediate des Intermediatepaares)
4. y-Koordinate des Endpunktes eines Kantenteilstücks (bzw. des zweiten Intermediate des Intermediatepaares)
5. Verschiebung (xOffset) der x-Koordinaten
6. Verschiebung (yOffset) der y-Koordinaten
7. Eindeutige Identifikationsnummer (ID) des Startknoten der Kante
8. Kantenqualität
9. Anzahl der Spuren der Kante

Die ersten vier Arrayelemente entsprechen den x- bzw. y-Koordinaten der Intermediates, durch die die Teilstücke gebildet werden. Aus der Darstellung geht hervor, dass innerhalb eines Vektors die x- und y-Koordinaten des Endpunktes eines Arrays a_j, den x- und y-Koordinaten des Anfangspunktes des darauf folgenden Arrays a_{j+1} entsprechen, mit $j < n - 1$ für $n > 2$ (andernfalls besteht der Vektor aus lediglich einem Array). Durch diese Eigenschaft lassen sich die ursprünglichen Kanten, welche durch jeweils einen Vektor beschrieben werden, sehr einfach rekonstruieren. Innerhalb eines Vektors müssen lediglich die Anfangs- und Endpunkte des ersten Arrays miteinander verbunden werden, danach muss der Endpunkt des ersten Arrays mit dem Endpunkt des zweiten Arrays verbunden werden, falls es mehr als ein Array gibt, usw. Im allgemeinen gilt also, dass sich eine Kante durch die Verbindung des Anfangs- und Endpunktes des Arrays a_1 und der darauf folgenden Verbindung der

Endpunkte der Arrays a_j und a_{j+1}, mit $1 \leq j < n-1$ für $n > 2$, rekonstruieren lässt.
Die Arrayelemente fünf und sechs beinhalten die Werte für die Verschiebung der x- bzw. y-Koordinaten (offset) der einzelnen Kantenabschnitte. Erst durch die Berechnung der Offsetwerte und die anschließende Verschiebung der Kanten um diesen Wert wird eine Richtungsdarstellung und Interpretation, wie sie in Kapitel 7.2.2 beschrieben wird, möglich. Ohne diese Berechnung und Verschiebung würden die Straßen, wie weiter oben erläutert wurde, einfach übereinander gezeichnet. Im Wesentlichen ergibt sich die Verschiebung durch die Bildung eines zu dem Kantenabschnitt orthogonalen Vektors. Abbildung 4.22 verdeutlicht die Verschiebung. In der Abbildung wird ein Kantenteilstück durch den

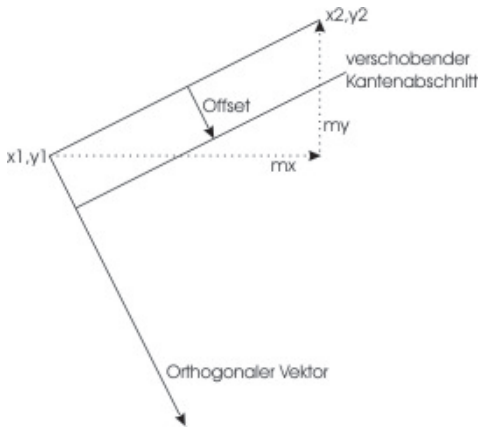


Abbildung 4.22: Berechnung der Verschiebung

Anfangspunkt (x1,y1) und den Endpunkt (x2,y2) beschrieben. Des Weiteren ist ein zu dem Kantenstück orthogonaler Vektor dargestellt, welcher das Kantenstück im Punkt (x1,y1) schneidet. Der orthogonale Vektor ergibt sich aus der Steigungsvektordarstellung $\binom{mx}{my}$ des Kantenabschnitts als Vektor $\binom{my}{-mx}$. Die in der Abbildung durch den Vektor „offset“ dargestellte Verschiebung, entspricht der Normierung des orthogonalen Vektors, welche sich durch das Skalarprodukt $\frac{1}{\sqrt{mx^2+my^2}} * \binom{my}{-mx}$ berechnen lässt. Damit stellt das Offset einen normierten, orthogonalen Vektor bzw. Einheitsvektor dar, was einer Verschiebung um 1 entspricht. Nach diesen Erkenntnissen lässt sich die Verschiebung folgendermaßen formalisieren:

$$mx = x2 - x1 \tag{4.1}$$

$$my = y2 - y1 \tag{4.2}$$

$$xOffset = 0,4 * my * \frac{1}{\sqrt{mx^2 + my^2}} \tag{4.3}$$

$$yOffset = 0,4 * -mx * \frac{1}{\sqrt{mx^2 + my^2}} \tag{4.4}$$

Aus den Formeln 4.2 und 4.2 ergibt sich der Steigungvektor des Kantenstücks, welche für die Bestimmung des orthogonalen Vektors benötigt wird. Die Formeln 4.3 und 4.4 entsprechen der Berechnung des zu dem Kantenstücks orthogonalen, normierten Vektors. Des Weiteren werden die einzelnen Verschiebungsvektorwerte mit 0,4 multipliziert, was einer Verkürzung des Verschiebungsvektors um 0,6 entspricht. Die Verkürzung des Vektors hat lediglich damit zu tun, dass die Straßenseiten nicht zu weit

auseinander gezeichnet werden, prinzipiell sind aber auch andere oder keine Verkürzungen möglich. Die einzelnen Verkürzungswerte müssen lediglich übereinstimmen, da die Berechnung dem Skalarprodukt aus Verkürzungswert und orthogonalen, normierten Vektor entspricht.
Aus Kapitel 4.12 geht hervor, dass jede Kante einen Anfangs- und Endknoten besitzt. Die ID des Anfangsknoten einer Kante wird im siebten Arrayelement eines jeden Arrays gespeichert, obwohl es prinzipiell reichen würde, die ID des Startknotens einmal für die gesamte Kante anzugeben. Allerdings wird so eine gewisse Einfachheit der Datenstruktur gewahrt. Die Klasse *VisualizationView* liest lediglich zum graphischen Darstellen die ID des ersten Arrays aus. Prinzipiell könnten aber alle IDs eines Vektors benutzt werden, da diese Identisch sind. Auf die ID des Endkoten wird komplett verzichtet, da sich diese aus dem Startknoten der nachfolgenden Kante ergibt.
Das achte Arrayelement enthält die Kantenqualität, welche sowohl für die im Kapitel 7.2.2 erläuterte farbliche Qualitätsdarstellung als auch für die Qualitätsdarstellung als Zahl benutzt wird. Die Kantenqualität selbst wird nicht von der Klasse *EdgeProcessor* berechnet. Aufschluss über die Berechnung gibt Kapitel 2.5. Hier würde es theoretisch auch reichen, die Qulität einer Kante einmal für die gesamte Kante anzugeben. Allerdings könnte man so zukünftig Kantenqualitäten noch differenzierter darstellen, in dem die Qualitäten für einzelne Kantenabschnitte berechnet werden. Diese Funktionalität wird jedoch noch nicht unterstützt.
Im neunten Arrayelement wird die Anzahl der Spuren einer Kante (vgl. Kap. 4.12) gespeichert, um auch diese graphisch darstellen zu können und so ein graphisches Testen des Verkehrsmodells zu ermöglichen. Genauere Informationen zur Darstellung der einzelnen Fahrspuren können Kapitel 7.2.2 entnommen werden.

VehicleProcessor

Neben den Informationen zum Zeichnen der Straßen werden weitere Informationen zum Zeichnen der Fahrzeuge benötigt. Die Klasse *VehicleProcessor* stellt alle zum Zeichnen der Fahrzeuge relevanten Daten zusammen. Dafür wird eine zweite Datenstruktur erzeugt, welche in Abbildung 4.23 dargestellt ist. Die Datenstrukur der Fahrzeuge besteht aus einem Basisvektor (vgl. Abb. 4.23), welcher für je-

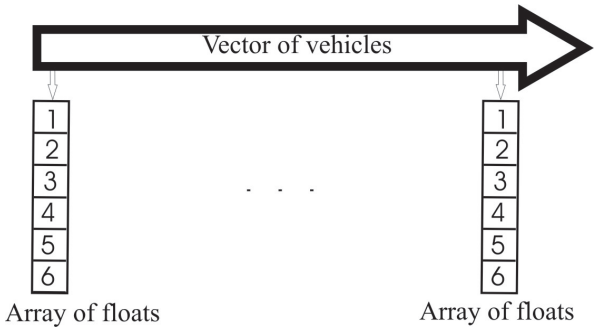


Abbildung 4.23: Datenstruktur der Fahrzeuge

des Fahrzeug ein Fließkommazahlenarray der Länge sechs enthält. Die folgende Liste beschreibt die einzelnen Arraypositionen und die damit verbundenen Informationen:

1. x Koordinate des Fahrzeugs im kartesischen Koordinatensystem
2. y Koordinate des Fahrzeugs im kartesischen Koordinatensystem

3. Eindeutige Identifikationsnummer des Fahrzeugs
4. Offsetwert auf der x-Achse im kartesischen Koordinatensystem
5. Offsetwert auf der y-Achse im kartesischen Koordinatensystem
6. Repräsentative Nummer für den Routingalgorithmus, mit dem das Fahrzeug geroutet wird

In den ersten beiden Arrayelementen wird die Position des Fahrzeugs im kartesischen Koordinatensytem gespeichert. Die Position eines Fahrzeugs im Koordinatensystem ergibt sich aus den Koordinaten der Intermediates des Kantenstücks auf dem sich das Fahrzeug befindet und der Kopfposition des Fahrzeugs, welche einer Zelle der jeweiligen Kante entspricht (vgl. Kap. 4.5). Da die Position eines Fahrzeugs lediglich über die Zelle einer Kante beschrieben wird, muss zunächst das Teilstück der Kante bestimmt werden, auf dem sich das Fahrzeug befindet. Da keine Information über die Verteilung der Zellen einer Kante auf die einzelnen Teilstücke vorliegt, lässt sich das Teilstück lediglich über Längenberechnungen bestimmen. Dafür wird die Eigenschaft ausgenutzt, dass die Länge einer Kante dem Produkt aus der Anzahl der Zellen der Kante und der Zellgröße entspricht. Des Weiteren kann die vom dem Fahrzeug zurückgelegte Strecke auf einer Kante aus dem Produkt der Kopfposition und der Zellgröße bestimmt werden. Ausgehend vom ersten Teilstück der Kante wird die Länge des Teilstücks mit der zurückgelegten Strecke verglichen. Ist die zurückgelegte Strecke des Fahrzeugs kleiner als die Länge des Teilstücks, so ist das Teilstück, auf dem sich das Fahrzeug befindet, gefunden. Falls die zurückgelegte Strecke länger als das Teilstück ist, wird das nächste Teilstück betrachtet, allerdings muss zur Länge des Teilstücks die Länge der vorher betrachteten Teilstücke addiert werden. Die Betrachtung der Teilstücke wird so lange fortgesetzt, bis die zurückgelegte Strecke des Fahrzeugs kleiner ist, als die Länge der bereits betrachteten Teilstücke. Trivialerweise endet die Suche spätestens bei der Betrachtung des letzten Teilstücks. Algorithmus 3 formalisiert die beschriebene Suche; dabei berechnet die Methode *computeLength(Intermediate1,Intermediate2)* die Länge eines Teilstücks, welches durch zwei Intermediates beschriebenen wird.
Nachdem das durch die Intermediates beschriebene Teilstück gefunden wurde, kann mit der eigentli-

Algorithmus 3 Teilstücksuche

1: **Eingabe:** Liste von Intermediates $L_e = \{I_1, ..., I_n\}$ der Kante e, auf der sich das Fahrzeug befindet
2: **Eingabe:** Zellgröße g
3: **Eingabe:** Kopfzelle des Fahrzeugs z
4: **Ausgabe:** Intermediatepaar $\{I_j, I_{j+1}\} \in L_e$
5: **Init:** Vom Fahrzeug zurückgelegte Strecke $s := z * g$
6: **Init:** j:=1
7: **Init:** Temporäre Intermediates $ITMP1 := I_j$, $ITMP2 := I_{j+1}$ durch die das jeweils betrachtete Kantenstück beschrieben wird
8: **Init:** Länge der betrachteten Kantenstücke $l := computeLenght(ITMP1, ITMP2)$
9: **while** $l < s$ **do**
10: j:=j+1
11: $ITMP1 := I_j$
12: $ITMP2 := I_{j+1}$
13: $l := l + computeLenght(ITMP1, ITMP2)$
14: **end while**
15: **Return:** $\{ITMP1, ITMP2\}$

chen Positionsbesrechnung begonnen werden. Für die Bestimmung der Position des Fahrzeugs wird die Einteilung der Kanten in Zellen ausgenutzt (vgl. Kap. 4.5), in dem die Zellen auf die x- und y-Achsen des Koordinatensystem projiziert werden. Abbildung 4.24 verdeutlicht dieses Vorgehen. Die

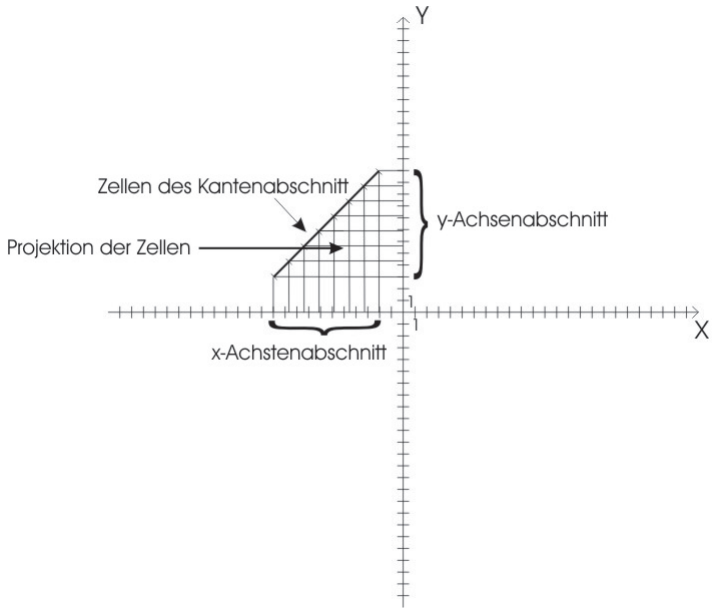


Abbildung 4.24: Projektion der Zellen auf die Achsen des Koordinatensystem

Projektion der Zellen auf die Achsen des Koordniatensystems wird in Abbildung 4.24 durch die Linien zwischen dem Kantenstück und den Achsen angedeutet, durch die sich eine Einteilung des x- bzw. y-Achsenabschnitts des Kantenstücks in kleinere Zellen ergibt. Da die Größe der Zellen der Achsenabschnitte und die der Kante verschieden sind, muss die Zellgröße der jeweiligen Abschnitte aus dem Quotienten der Länge des Achsenabschnitts und der Anzahl der Zellen des Kantenstücks berechnet werden. Ein Fahrzeug, von dem Anfangs nur die Kopfposition bekannt war, lässt sich nun sehr einfach im Koordinatensystem positionieren. Zur Bestimmung der x-Koordinate des Fahrzeugs muss lediglich das Produkt aus der Kopfposition des Fahrzeugs und der Zelllänge des x-Achsenabschnitts mit der x-Koordinate des Anfangsintermediate des Kantenstücks addiert bzw. subtrahiert werden. Ob summiert oder subtrahiert wird, hängt von der Lage des Kantenstücks im Koordinatensystem ab. Wenn die x-Koordinate des Startintermediates kleiner ist als die des Endintermediates wird addiert, da sich das Fahrzeug im Koordinatensystem auf der Kante quasi von „links“ nach „rechts“ bewegt, was einer Vergrößerung der x-Koordinate entspricht. Ist die x-Koordinate des Startintermediates größer, wird subtrahiert, da es sich in diesem Fall von „rechts“ nach „links“ bewegt, was einer Verkleinerung der x-Koordinate entspricht. Sollten beide Koordinaten gleich sein, muss weder addiert noch subtrahiert werden, da sich das Fahrzeug in diesem Fall nicht auf der x-Achse bewegen kann. Die Bestimmung der y-Koordinate erfolgt analog.
Die Arrayelemente drei und vier beschreiben den Verschiebungs- bzw. Offsetwert des Fahrzeugs. Da die Fahrzeuge genau auf die verschobenen Kanten gezeichnet werden, entspricht dieser Wert dem Offsetwert des Kantenstücks. Die Berechnung der Verschiebungswerte gestaltet sich analog zur Berechnung der Verschiebungswerte der Kantenstücke, daher wird auf eine Beschreibung der Berechnung an dieser Stelle verzichtet.
Im sechsten Arrayelement wird eine repräsentative Fließkommazahl für den Algorithmus, mit dem

das Fahrzeug geroutet wird, gespeichert. So kodiert zum Beispiel eine eins den von der Projektgruppe entwickelten BeeJamA-Algorithmus und eine zwei den Dijkstra Algorithmus. Diese Information wird für die Färbung der Fahrzeuge benötigt (vgl. Kap. 7.2.2).

4.10 Implementierung des Verkehrsmodells

Das bereits in Abschnitt 3.2 erwähnte Verkehrsmodell wurde in drei Klassen gekapselt. Da sich das Verkehrsmodell aus zwei seperaten Modellen zusammensetzt, bot es sich an jedes Modell in einer eigenständigen Klasse zu implementieren. Das Einspurmodell liefert die Grundlage für den *MovementComputer* und das Mehrspurmodell für den *LaneChanger*. Um den Ablauf zwischen beiden Klassen zu koordinieren und um die Struktur des Simulators beizubehalten der *TrafficController* als Kontrollerklasse. Dieser regelt alle verkehrsmodellrelevanten Aufrufe und leitet diese an die oben erwähnten Klassen weiter. Weiterhin erweitert er die vom Simulator geforderte Klasse *Operator*, so dass er in die einzelnen Chains eingegliedert werden kann. Um den Code innerhalb des *LaneChanger* und *MovementComputer* lesbarer zu gestalten wurde der *TrafficModelHelper* geschrieben, welcher vom Verkehrsmodell benötigte Hilfsmethoden anbietet. Da das Verkehrsmodell selber keine Regelung von Situationen an Kreuzungen vorsieht, wurde dieser Bereich im *CrossroadChecker* gekapselt. Zum besseren Verständnis und der Übersicht halber zeigt Abbildung 4.25 den Aufbau des Verkehrsmodells in Form eines UML-Klassendiagramms. Im Folgenden werden die einzelnen Klassen detailiert erläutert.

TrafficController Der TrafficController ist wie bereits erwähnt die Kontroll-Instanz des gesamten Verkehrsmodells. Er wird als Operator in die StepLastChain geladen, da er pro Schritt einmal aufgerufen werden muss. Es ist unbedingt notwendig den *TrafficController* in der StepLastChain aufzurufen, da vor der Neuberechnung der Fahrzeugpositionen die Routenberechnung durchgeführt werden muss. Er erhält über das World-Objekt alle aktiven Fahrzeuge und berechnet für jedes die neue Position, indem er im LaneChanger die Methode *computeLaneChange(Vehicle)* und im *MovementComputer* die Methode *computeMovement(Vehicle)* aufruft. Die daraus resultierende neue Position wird zwischengespeichert bis diese für alle Fahrzeuge berechnet ist. Danach werden die gespeicherten Positionen für alle Fahrzeuge in das World-Objekt übertragen. Weiterhin ist der TrafficController für das Aktualisieren der Postlane beim Befahren einer neuen Kante und nach dem Spurwechsel zuständig. Eine grafische Darstellung des Ablaufs zeigt Abbildung 4.26. Die Postlane ist ein Konstrukt, welches es möglich macht über das Vehicle-Objekt eine Referenz auf die Lane auf der nächsten Kante zu erhalten (vergl. Abschnitt 4.10).
Zuletzt sollte noch erwähnt werden, dass sich die Parameter des Verkehrsmodells (Abschnitt 3.2.4) als Attribute im TrafficController befinden. Diese können vor Simulationsbeginn über die GUI verändert und gespeichert werden (siehe Abschnitt 7.2).

LaneChanger Der *LaneChanger* setzt die Bedingungen des Mehrspurmodells aus Abschnitt 3.2.3 um. Bedingung 1 beschreibt die Sicherheitsüberprüfung auf der Zielspur und wurde in zwei interne Methoden aufgeteilt. Einmal die Methode *checkSavetyRegionOnLeftLane(Vehicle)* und zum anderen *checkSavetyRegionOnRightLane(Vehicle)*. Die Methoden haben im Wesentlichen die selbe Funktionalität, betrachten allerdings unterschiedliche Spuren und werden deshalb und zum besseren Verständnis separiert. Innerhalb dieser Methoden wird das Konstrukt der Postlane benötigt, um gegebenenfalls am

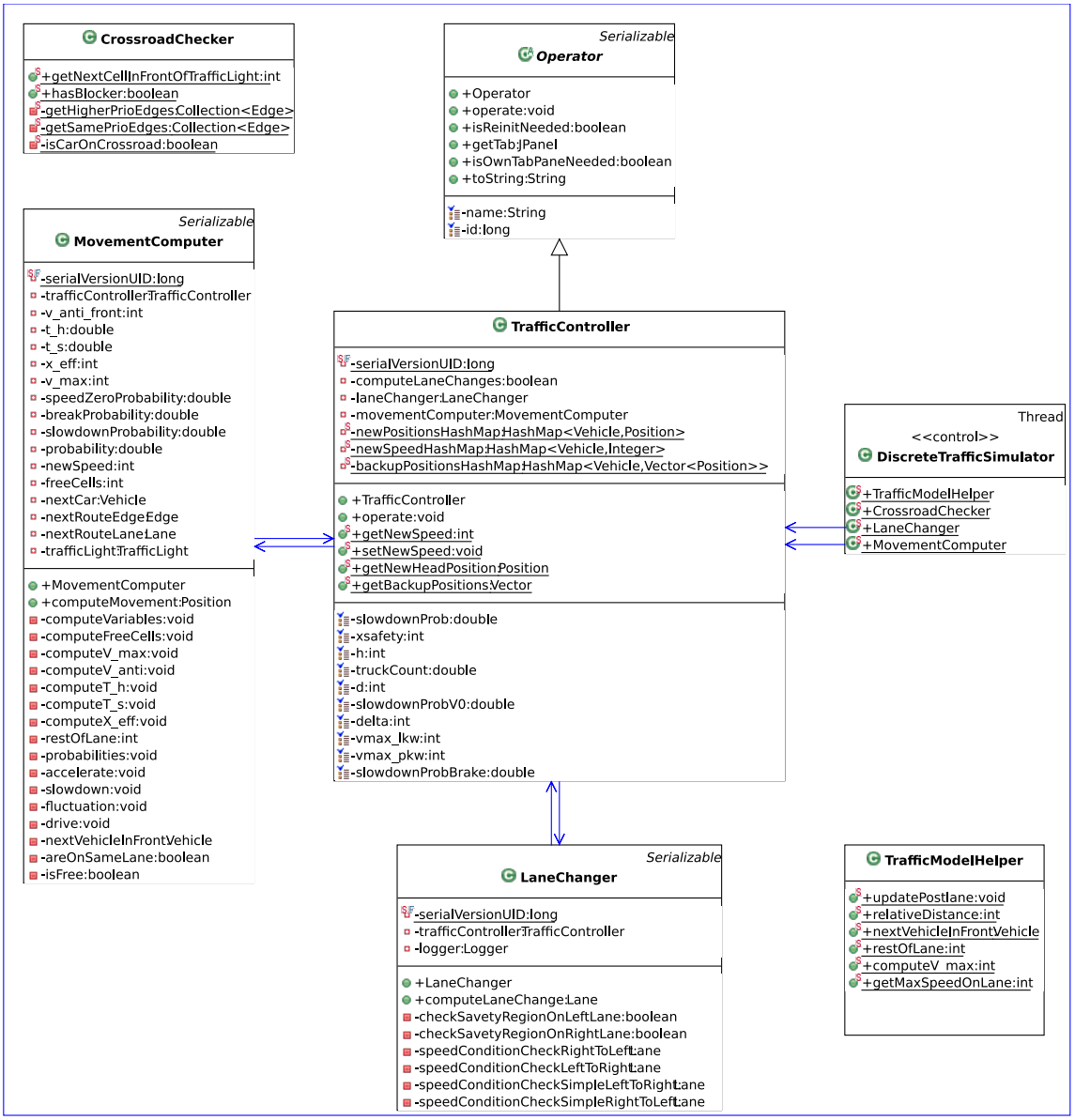


Abbildung 4.25: UML Klassendiagramm des Verkehrsmodells

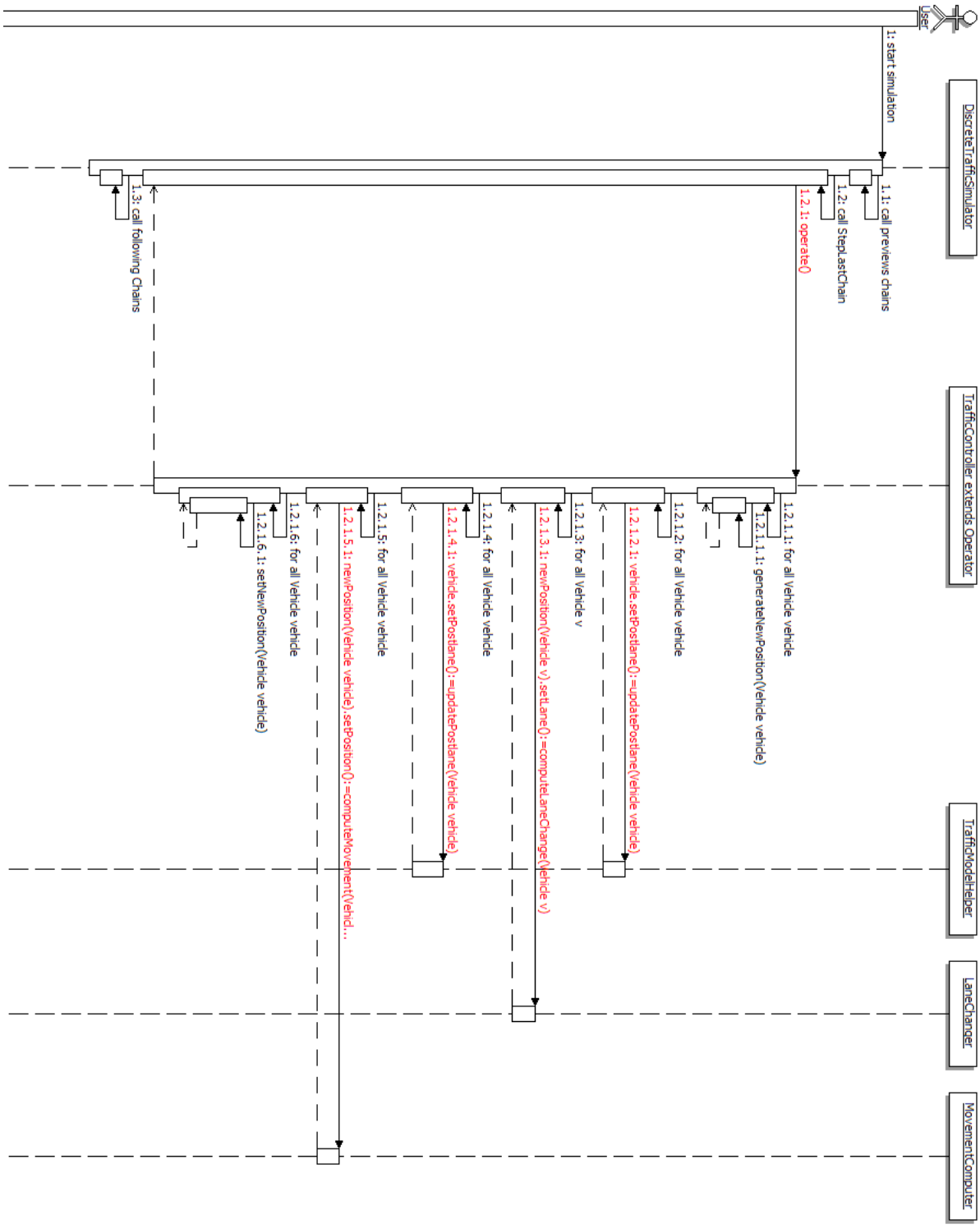


Abbildung 4.26: Ablauf einer Berechnung des Verkehrsmodells für einen Schritt

Ende einer Kante auf der Lane der nächsten Kante den Sicherheitsbereich prüfen zu können. Für Bedingung 2 existieren sowohl symmetrische als auch asymmetrische Wechselbedingungen. Diese werden je nach auftretendem Fall aufgerufen werden. Die entsprechenden internen Methoden haben folgenden Rumpf: *speedConditionCheckRightToLeft(Vehicle)* für den asymmetrischen Fall und *speedConditionCheckSimpleRightToLeft(Vehicle)* für den symmetrischen Fall. Es existieren für beide Fälle jeweils zwei Methoden die Spurwechsel nach links und nach rechts unterscheiden, da dort unterschiedliche Bedingungen gelten (siehe Abschnitt 3.2.3).
Weiterhin gibt es die nach außen sichtbare Methode *computeLaneChange(Vehicle)* welche den kompletten Ablauf eines Spurwechsels enthält und die bereits erwähnten internen Methoden in entsprechender Reihenfolge aufruft. Mit komplettem Ablauf ist das Prüfen eines vorausfahrenden Fahrzeugs und das Aufteilen der Spurwechsel in links nach rechts Wechsel in ungeraden Schritten und rechts nach links Wechsel in geraden Schritten gemeint. Den groben Ablauf eines Spurwechsels kann man Abbildung 4.27 entnehmen. Es wurde bewusst auf Details im Diagramm verzichtet damit die Lesbarkeit gewährleistet bleibt. Detailiertere Angaben zum Ablauf eines Spurwechsels sind der Klasse *LaneChanger* zu entnehmen, welche umfassend kommentiert ist.

MovementComputer Im MovementComputer werden die Regeln des Einspurverkehr umgesetzt. Anwendung finden das Nagel-Schreckenberg-Modell 3.2.2, des VDR Modell 3.2.2sowie die Bremslichterweiterungen des Nagel-Schreckenberg-Modells. Neben den fünf beschriebenen Schritten, müssen außerdem noch einige weitere Situationen beachtet werden. Da das Straßennetz im Simulator nicht aus einer einzigen langen Fahrspur besteht, sondern sich aus Straßenabschnitten zusammensetzt, müssen die Übergänge zwischen diesen Abschnitten gesondert betrachtet werden. Abbildung 4.28 zeigt den Ablauf der Berechnungen in der *MovementComputer* Klasse. Nach dem Aufruf der Methode *computeMovement(Vehicle)* durch den *TrafficController*, werden zuerst die nächste zu befahrende Kante und Spur ermittelt. Diese Informationen wurden bereits im *TrafficController* für jedes Vehicle berechnet und müssen nur noch ausgelesen werden. Diese Information ist wichtig, da sonst nicht festgestellt werden kann, welche Fahrzeuge Einfluss auf das Fahrverhalten des aktuelle berechneten Fahrzeugs haben. Weiterhin wird überprüft, ob sich eine Ampel auf der Fahrbahn befindet. Im nächsten Schritt wird eine neue HeadPosition angelegt. Die anfänglichen Werte werden aus der aktuellen Kopfposition des Fahrzeuges kopiert. Es ist wichtig, nicht die aktuelle Position zu verändern, da dadurch nicht alle Fahrzeuge den gleichen Zustand der Welt (bzw. des Automaten) als Berechnungsgrundlage verwenden würden. Die neu angelegte Position wird dann durch die folgenden Berechnungen modifiziert.
Der Aufruf der Methode *computeVariables()* kapselt die Berechnung der Werte, die für die Berechnung der Einspurregeln laut Abschnitt 3.2.2 benötigt werden. Als erstes wird das vorrausfahrende Fahrzeug mit *nextVehicleInFront(Vehicle)* berechnet. Durch das vorausfahrende Fahrzeug werden die Berechnungen maßgeblich beeinflusst. Im nächsten Schritt wird dann der Abstand zu diesem Fahrzeug, sofern eines vorhanden ist, berechnet. Befindet sich kein vorrausfahrendes Fahrzeug auf der Straße, wird der Abstand auf die Maximalgeschwindigkeit der jeweiligen Kante gesetzt. Die antizipierte Geschwindigkeit berechnet sich aus dem Minimum der Geschwindigkeit der Vorausfahrenden erhöht um eins und dem Abstand des Vorausfahrenden zu dessen Vordermann. Dadurch wird eine einfache Annahme der gechätzen Geschwindigkeit getroffen. Mit *computeT_h()* wird der zeitliche Abstand zum Vorrausfahrenden Fahrzeug ermittelt. Dieser ist einfach der Quotient aus freien Zellen und Geschwindigkeit. *computeT_s(Vehicle)* berechnet den Interaktionsradius welcher eine Funktion der aktuellen Geschwindigkeit und dem in Abschnitt 3.2.4 beschriebenen Parameter h, der die Reichweite des Bremslichtes angibt. Dieser Paramteter wird beim Start der Simulation festgelegt, und kann über die entsprechende *getH()* Methode aus dem *TrafficController* ausgelesen werden. *Compu-*

Abbildung 4.27: Interner Ablauf für die Berechnung einer neuen Spur für ein Fahrzeug

teX_eff(Vehicle) berechnet letztendlich den effektiven Abstand zum vorrausfahrenden Fahrzeug unter Beachtung der antizipierten Geschwindigkeit.
Jetzt folgt die eigentliche Berechnung laut dem in 3.2.2 beschriebenen Schritten. Diese Methoden sind unmittelbar dem theoretischen Modell nachempfunden.

CrossroadChecker Der CrossroadChecker dient zur Realisierung des Verhaltens an Kreuzungen sowie kurz davor. Er besteht aktuell aus den folgenden Methoden

$getNextCellInFrontOfTrafficLight(Vehicle, x\_eff)$
: Fährt das Auto auf eine rote Ampel zu ist das die letzte Zelle die befahren werden darf. Diese Zelle wird durch den Aufruf der Methode *getNextCellInFrontOftrafficLight(Vehicle, int)* zur¨uckgegeben. Der effektive Abstand wird übergeben, um einen eventuell geringeren, vorherberechneten Abstand, zu beachten. In diesem Fall ist die Berechnung der nächsten Ampel nicht relevant.

$hasBlocker(Vehicle)$
: Die Methode *hasBlocker(Vehicle)* hat die Aufgabe zu bestimmen, ob ein Fahrzeug auf eine Kreuzung fahren darf oder nicht. Dabei werden mehrere Bedingungen überprüft. Es wird überprüft was für Straßen auf die Kreuzung zulaufen. Dabei werden die Straßen nach der Priorität (Landstraße, Autobahn, etc.) geordnet. Möchte ein Auto von einer Straße höherer Priorität ebenfalls die Kreuzung passieren, hat dieses Vorrang. Autos niedrigerer Priorität müssen dementsprechend warten. Befinden sich nur Straßen gleichen Rangs an der Kreuzung, und es wollen mehrere Autos die Kreuzung passieren, wird derzeit per Zufall ein Auto ausgewählt. Das entspricht nicht der im Straßenverkehr üblichen rechts vor links Regeln, ist aber ein akzeptables Verfahren um trotzdem den Verkehrsfluss zu simulieren. Weiterhin stellt diese Regelung bei den hier vorwiegend betrachteten Autobahnen keine Hindernis dar, da sich dort keine Kreuzungen sondern nur Auf- und Abfahrten befinden. Autobahnauffahrten werden in Abschnitt 3.2.5 behandelt.

TrafficControllerPolice() Die TrafficControllerPolice Klasse wurde eingeführt , um Fehler bei der Implementierung des Verkehrsmodells aufzudecken. Zur Zeit sind darin drei Methoden implementiert.

$checkSuperposed()$
: Überprüft, ob sich zwei Fahrzeuge auf exakt der selben Position befinden, also übereinander liegen.

$checkPositionConsistence()$ In dieser Methode wird für alle Fahrzeuge überprüft ob die Positionen konsistent sind. Sollte eine Position eine Lane und eine Kante beinhalten die nicht zusammengehören wird dieser Fehler hier erkannt.

$waitingVehicleAllowedToDrive()$
: Diese Methode überprüft den Fall, ob ein Auto an einer Kreuzung warten muss, da es auf eine Fahrbahn mit weniger Spuren als auf der aktuellen wechseln möchte und die aktuelle Lane keine Postlane besitzt, also keine folgende Spur auf der nächsten Kante vorhanden ist. In diesem Fall muss entsprechend den Mehrspurregeln erst nach rechts gewechselt werden, um auf einer Lane zu sein die auch eine folgende Lane besitzt. Ein solches Fahrzeug darf nicht die Erlaub-

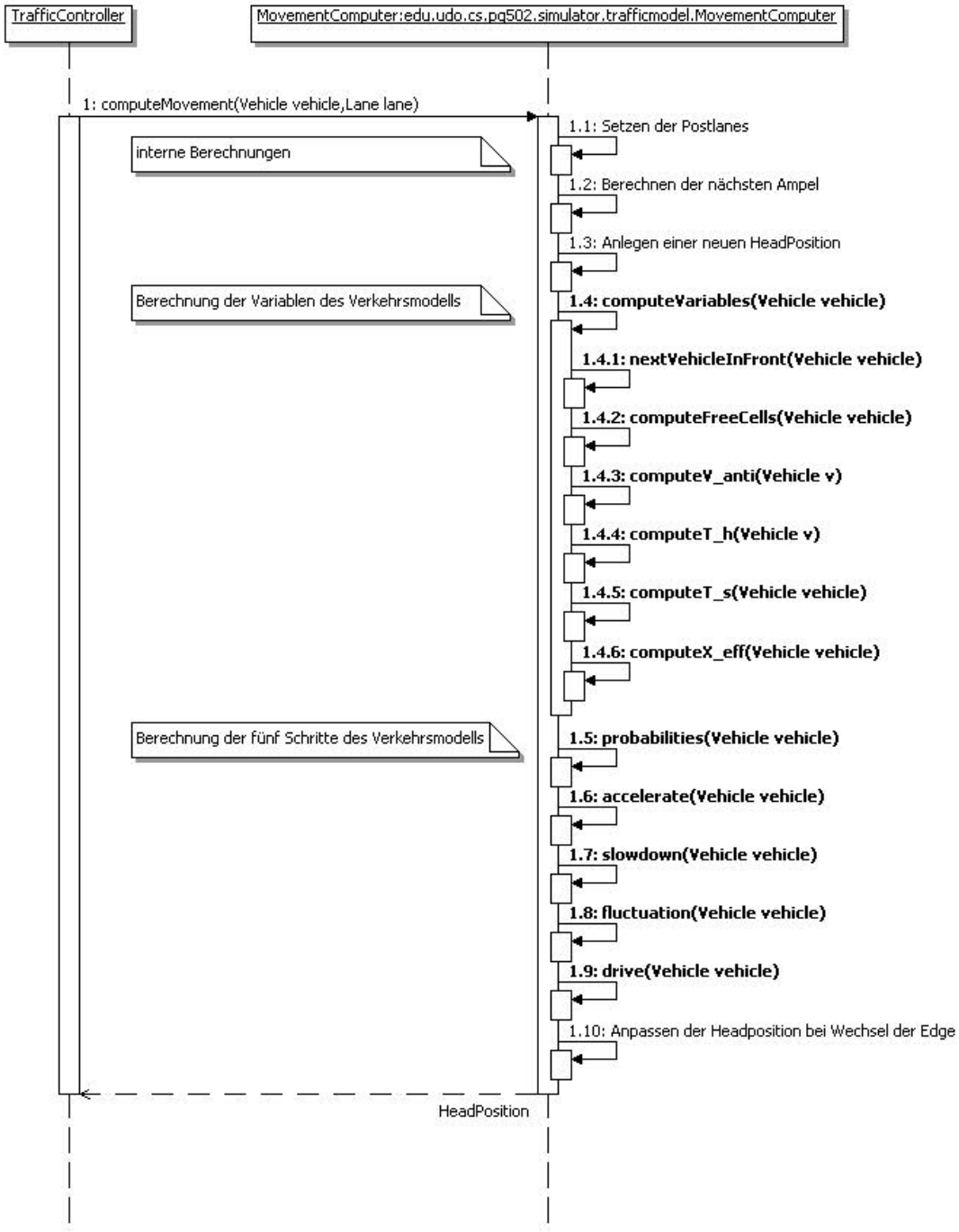


Abbildung 4.28: Sequenzdiagramm des MovementComputer

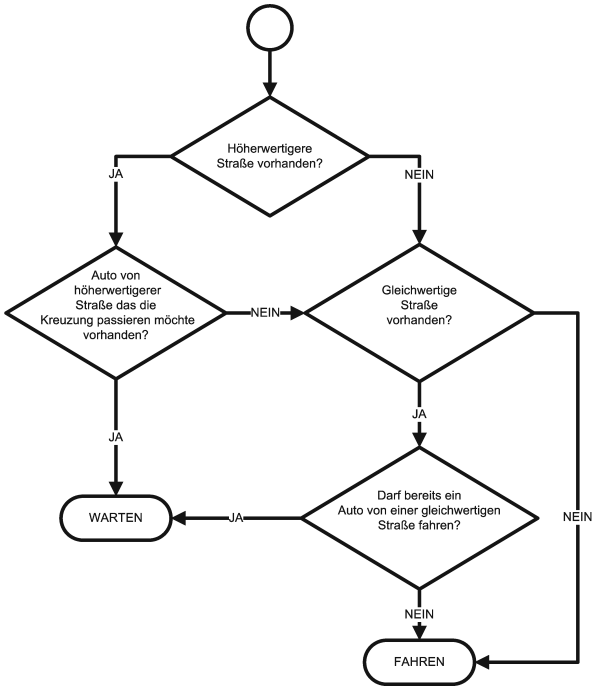


Abbildung 4.29: Verhalten an Kreuzungen

nis erhalten die Kreuzung zu passieren, da es ohnehin nicht möglich ist. Sollte das dennoch passieren wird dieser Fehler hier aufgedeckt.

Entsprechend dem aufgetretenen Fehler wird dann eine entsprechende *TrafficControllerException* erzeugt. Die Ausnahmen sind im Package *trafficmodel* definiert. Diese Klasse kann beliebig erweitert werden. Sie dient dazu, neue Implemetierungsdetails des Simulators während der Simulation überprüfen zu können. Die *TrafficControllerPolice* Klasse muss, wenn sie benutzt wird, als Operator in die StepLastChain eingef/ügt werden. Der Simulator kann allerdings auch ohne diese Klasse ausgeführt werden.

TrafficModelHelper Diese Klasse enthält Hilfsmethoden welche aus den anderen Klassen zur besseren Lesbarkeit des Codes ausgelagert wurden. Die Methoden sind intuitiv nach ihrer Funktionalität benannt und werden im folgenden kurz aufgelistet.

$updatePostlane(Vehicle)$
: Die Berechnung der Postlane, also der Lane welche von dem Fahrzeug als nächstes befahren wird, wird im TrafficController vor der Berechnung der eigentlichen Verkehrsregeln durchgeführt. Diese Vorrausbrechnung ist nötig, da das im Simulator verwendete Straßennetz aus vielen einzelnen Elementen (Kanten) mit mehrenen Spuren besteht, deren Anzahl sich zusätzlich noch entsprechend den Straßentypen (Autobahn, Landstraße, etc.) ändert. Außerdem müssen an Kreuzungen Routinginformationen mitbeachtet werden.

$relativeDistance(Vehicle, Vehicle)$
: Dient der Berechnung von Abständen zwischen Fahrzeugzeugen unter Beachtung eventuell unterschiedlicher Kanten.

$nextVehicleInFront(Vehicle)$
Berechnet das vorrausfahrende Fahrzeug.

$restOfLane(Vehicle)$
Berechnet ausgehend von der vordersten benutzten Zelle des jeweiligen Fahrzeugs die Anzahl Zellen, die sich noch auf dieser Kante befinden.

$computeV\_max(Vehicle)$
Gibt den Wert für die maximal mögliche Geschwindigkeit zurück. Dieser setzt sich aus dem Minimum der maximalen Geschwindigkeit der befahrenen Kante und der maximalen Fahrzeuggeschwindigkeit zusammen.

$getMaxSpeedOnLane(Lane, Vehicle)$
Gibt die Geschwindigkeit des schnellsten Fahrzeugs auf der Fahrspur zurück.

4.11 Implementierung der Routingalgorithmen

In diesem Abschnitt werden die einzelnen implementierten Algorithmen vorgestellt und deren programmiertechnische Feinheiten erläutert.

Das Package *routingAlgorithm*

Im Package *routingAlgorithm* befinden sich zwei wichtige Klassen welche von jedem Routingalgorithmus genutzt werden:

RoutingAlgorithm Das Interface *RoutingAlgorithm* wird vom *Operator BeeHiveRouter* implementiert. Es definiert keine zusätzlichen Attribute oder Methoden sondern dient in erster Linie dazu, um vom *RoutingDispatcher* als Routingalgorithmus erkannt zu werden und sich von anderen *Operatoren* abzugrenzen.

CommonCostCalculation Die abstrakte Klasse *CommonCostCalculation* wird benötigt um die Kosten jedes Straßenabschnittes zu berechnen. In dieser Klasse wird die *operate()*-Methode für weitere abgeleitete Klassen überschrieben und sie aktualisiert in jedem Durchlauf des Simulators jede Kante des Systems mit den aktuellen Werten.

4.11.1 Implementierung des BeeJamA-Algorithmus

Die hier vorgestellte Implementierung des BeeJamA-Algorithmus ist nur eine von vielen möglichen Algorithmen, die für den Simulator umgesetzt werden können und basiert auf dem zuvor erklärten Bienenschwarm-Algorithmus 2.3.

Die Kernkomponenten des BeeJamA-Routings sind im Package *routingAlgorithm.beehive* und in deren Unterpaketen zu finden:

beehive In diesem Hauptpaket befinden sich die beiden Kernklassen der Datenstruktur, welche für BeeJamA benötigt werden: *Node* und *Navigator*.

beehive.tables In diesem Package befinden sich alle genutzten Tabellen und deren Generalisierungen

beehive.tables.tableUpdater Jede Tabelle hat in diesem Package eine Update-Klasse welche die Aktualisierung der jeweiligen Tabelle veranlasst.

beehive.exception Hier sind die in Problemfällen zu werfenden Exceptions zu finden.

Des weiteren wurden alle Klassen welche die Klasse *Operator* erweitern im Package *operators.routingOperators* erstellt:

operators.routingOperators.beeHive Alle BeeJamA spezifischen *Operatoren* befinden sich in diesem Package.

operators.routingOperators.strategies.beeHive Hier kann der Entwickler das Verhalten des BeeJamA-Algorithmus ändern indem er die hier befindlichen Klassen bearbeitet oder neue erstellt.

Klassendiagramme

Im Klassendiagramm 4.30 sind die Beziehungen der vom BeeJamA-Algorithmus genutzten Klassen dargestellt. Zu sehen sind die beiden Entitäten *Node* und *Navigator* und die Hilfsklasse *BeeHiveHelper*. Das Package *tables* wird in 4.11.1 dargestellt.

Das Hauptpaket *beehive*

Das Hauptpaket *routingAlgorithm.beehive* besteht im wesentlichen aus den beiden Entitäten *Node* und *Navigator* welche die Repräsentation der in vorherigen Abschnitten vorgestellten Knoten des BeeJamA-Algorithmus darstellen. Hier werden daher nur die für den Algorithmus relevanten Details vorgestellt.

Node Die Klasse *Node* bildet die Hauptentität im BeeJamA-Algorithmus. Sie beschreibt einen Knotenpunkt im Routinggraphen und benötigt daher alle ein- bzw. ausgehenden Kanten und eine Referenz zu ihrer IFZArea-Tabelle (beschrieben in Abschnitt 4.31). Da innerhalb des Simulators die Entitäten *Vertex* und *Edge* bereit stehen und der Overhead minimal sein soll wurde auf eine weitere Implementierung dieser Funktionalitäten verzichtet und der *Node* erbt daher direkt von *Vertex*.

Navigator Die Entität *Navigator* ist eine Erweiterung der Klasse *Node* und gibt ihm Referenzen zu den Tabellen der Netzebene. Je nach vorgegebener Grösse der Bereiche in der Bereichsebene kann die Anzahl der *Navigatoren* innerhalb der Simulationsläufe variieren. Aufgrund dessen muss nicht jeder der Knotenpunkte ein *Navigator* werden um wiederum den Overhead durch Einsparung der Tabellenreferenzen zu minimieren.

BeeHiveHelper Dies ist eine Hilfsklasse und beinhaltet einige Fuktionen welche in mehreren Klassen Verwendung finden und innerhalb von BeeHiveHelper als statische Methoden gesammelt werden. Beispielhaft soll hier die Funktion *getDistance(from, to)* erwähnt werden, welche die räumliche Differenz zweier Knoten berechnet. Ebenso sind in dieser Klasse auch alle benötigten Lade- und Speicherfunktionen innerhalb des BeeJamA-Algorithmus untergebracht.

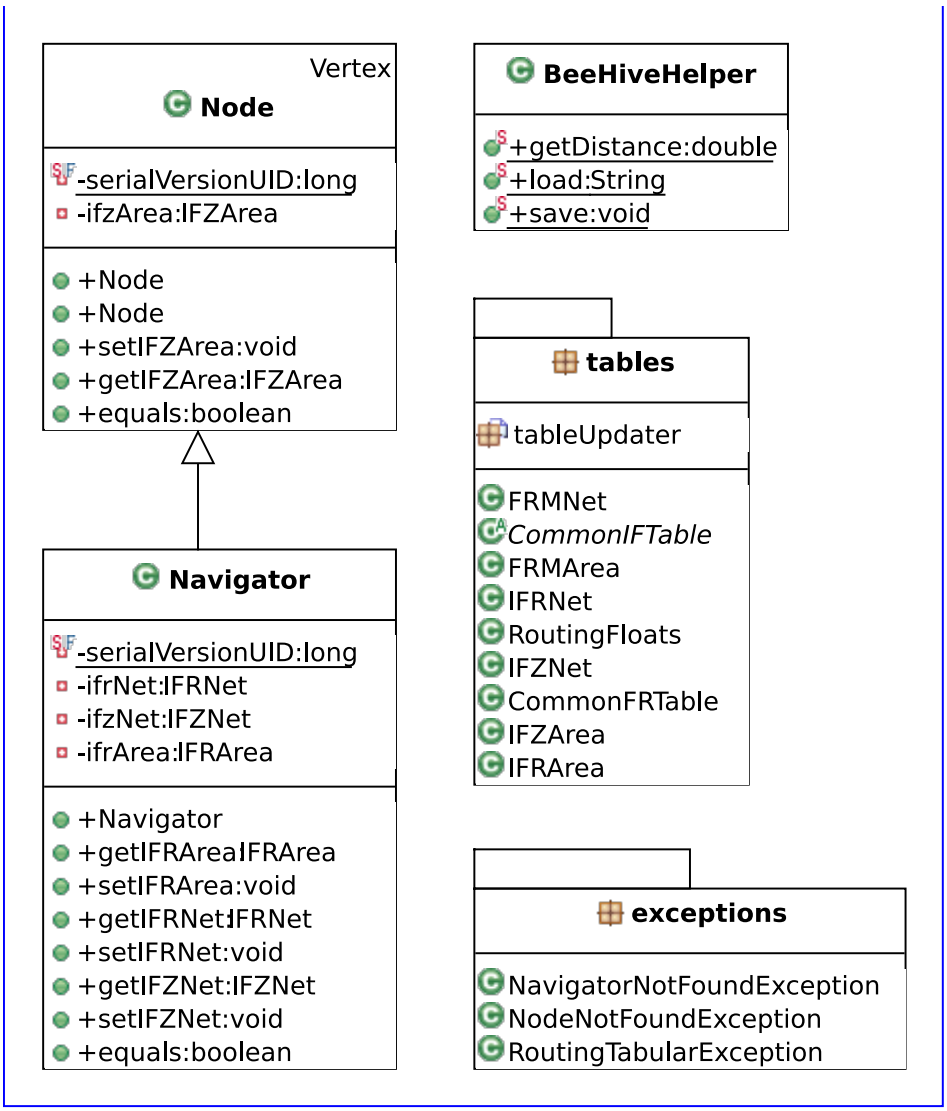


Abbildung 4.30: Klassendiagramm des BeeJamA-Algorithmus

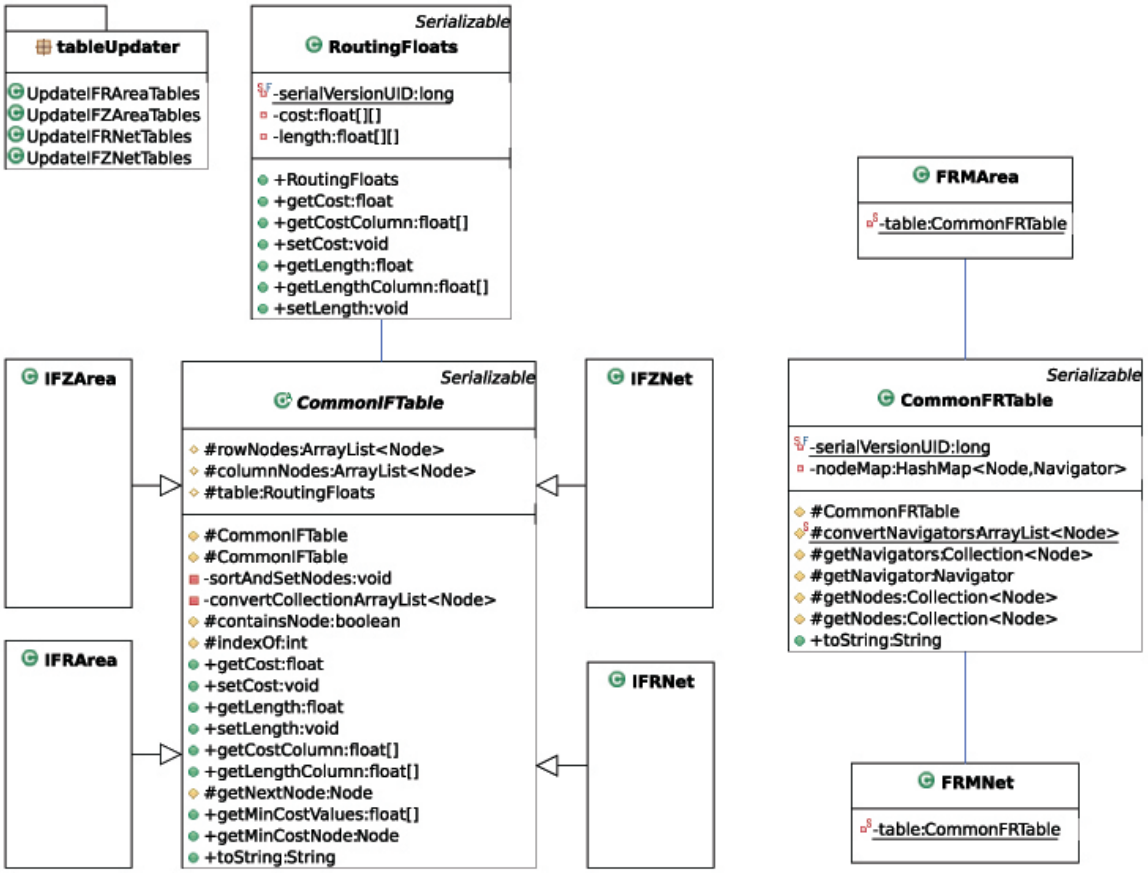


Abbildung 4.31: Klassendiagramm Package tables

Das Package *exceptions*

Alle Exceptions, welche vom BeeJamA-Routing geworfen werden können, befinden sich im gleichnamigen Package. Es gibt drei mögliche Fehlerquellen:

- ein *Node* wurde in einer Tabelle nicht gefunden
- die Referenz zu einem *Navigator* war nicht zu finden
- eine Tabelle wurde falsch oder doppelt erstellt

Für alle diese Probleme gibt es gleichnamige *NodeNotFound-*, *NavigatorNotFound-* und *RoutingTabularException*. Alle Klassen, welche direkt oder indirekt über andere Klassen mit der BeeJamA-Implementierung in Beziehung stehen, müssen gegebenenfalls auf solche Exceptions vorbereitet sein. Die problematischste Klasse in dieser Beziehung ist die Klasse *BeeHiveRouting*, da dort alle Knoten und deren Vererbungsklassen sowie alle möglichen Tabellen in Zusammenhang gebracht werden und daher dort die größte Fehlerquelle entsteht.

Die Tabellen

Das BeeJamA-Routing benutzt sechs verschiedene Tabellen in zwei verschiedenen Ebenen. Namentlich sind die beiden Ebenen die Bereichsebene (area layer) und die Netzebene (net layer).

FRM-Tabellen Jede Ebene besitzt eine FRM-Tabelle welche jedem Ebenenknoten einen repräsentativen Knoten zuordnet. Diese Zuordnung geschieht zur Laufzeit und wird im Preprocessing abgearbeitet. Um einen schnellen Zugriff auf diese Tabelle zu gewährleisten wurde die Zuordnung mittels einer Hashmap implementiert. Da diese Tabellen von jedem Knoten für den Verweis zu dem jeweiligen *Navigator* der nächst höheren Ebene angesprochen werden müssen, wurden die Tabellen als statische Objekte erzeugt.

Tabellen der Knoten Auf die jeweilige Aufgabenstellung zugeschnitten, besitzen die *Nodes* nur Verweise auf die IFZ-Tabelle ihres eigenen Bereiches. Die *Navigatoren* verwalten ihren Bereich der darunterliegenden Ebene und besitzen dadurch auch die IFR-Tabelle des verwalteten Bereiches. Ebenso wie die *Nodes* besitzen die *Navigatoren* auch ihre eigene IFZ-Tabelle ihrer Ebene. In der hier vorgestellten Implementierung gibt es mit der Bereichs- und Netz-Ebene zwei Ebenen. Die oberste Kontrolleinheit bilden dabei die Representative-Navigatoren welche die IFRNet-Tabelle und die Kontrolle über ihren Menge von *Navigatoren* in der Netz-Ebene besitzen.

Grundgerüst Die Grundlage für alle vorherig genannten Tabellen bilden die abstrakten Klassen *CommonFRTable* und *CommonIFTable*. Diese Klassen beinhalten schon alle notwendigen Methoden unbhängig von der Ebene der jeweiligen abgeleiteten Implementierung. Um die meistgenutzte Funktion der Suche nach der Spalte des Zielknoten effizient zu implementieren wurde eine sortierte Liste gewählt um die Knoten in der Matrizenzeile zu organisieren. Mittels diesem Preprocessing in der Tabellenerstellung kann die geschilderte Funktion in einer durchschnittlichen Laufzeit von *O(log n)* mittels binärer Suche geschehen. Das Package *tables* ist verkürzt dargestellt in Abschnitt 4.31. Das

Klassendiagramm zeigt nicht die Einzelheilten der jeweiligen Klassenimplementation von den einzelnen Tabellen, da diese hauptsächlich die Methoden der abstrakten Klassen *CommonIFTabel* oder *CommonFRTable* erweitern. Die genutzte Datenstruktur für die gespeicherten Daten der *CommonIFTable* befindet sich in der Klasse *RoutingFloats*. In der aktuellen Implementierung befinden sich zwei Daten je Eintrag:

- Die Kosten der Verbindung welche von der jeweiligen Routingstrategie errechnet wird, deren Berechnung im Abschnitt 2.5 erklärt wird. 4.11.1.
- Die statische Länge der Verbindung.

Die Klasse *CommonRouting*

Das Hauptproblem beim Routing mittels BeeJamA-Algorithmus ist die Berechung der Kosten je Kante des Systems. Die abstrakte Klasse *CommonRouting* erweitert die ebenfalls abstrakte Klasse *CommonCostCalculation* und delegiert die Berechnungsschritte an die einzelnen Routingstrategieen. Die einzige Tabelle welche direkt upgedatet wird ist die IFZArea- und die IFRArea-Tabelle, da dort die direkten Kosten der Kante eingetragen wird. Alle anderen Tabellen bauen auf diesen Werten auf und benötigen daher auch keine äußeren Einflüsse in ihren Berechnungsschritten.
Beim Routing werden ebenfalls Methoden von CommonRouting genutzt, welche wiederum an die angeschlossenen Routingstrategien delegiert werden. Die wichtigste Aufgabe besteht dabei in der Normalisierungsfunktion. Die Normalisierungsfunktion muss in der Lage sein die völlig unterschiedlichen Qualitätswerte auf den Wertebereich [0..1] abzubilden. Dabei ist es wichtig das diese Werte die jeweiligen Wahrscheinlichkeiten repräsentieren, welche die Qualitäten der Einzelrouten berücksichtigen.
Nachfolgend ein Beispiel was das Problem demonstriert:
Der Leser stelle sich zwei Routen zu einem nächsten Knoten vor.

1. Die erste Route hat Kosten von 1.
2. Die zweite Route hat Kosten von 2.

Mit einer Normalisierungsfunktion, welche nicht die Wahrscheinlichkeiten anpasst, ergeben sich die folgenden Werte:

1. Die Route über den ersten Weg hat die Wahrscheinlichkeit von $\frac{2}{3}$ gewählt zu werden.
2. Die Route über den zweiten Weg hat die Wahrscheinlichkeit von $\frac{1}{3}$.

Es ist zu erkennen das die doppelt so teure Route noch eine Wahrscheinlichkeit von $\frac{1}{3}$ hat, wenn es zwei Möglichkeiten gibt. Dieses Verhalten ist inakzeptabel da im System schlechtere Routen mit einer noch zu hohen Wahrscheinlichkeit gewählt werden könnten. Zum Beispiel auch Wege welche zu einem schon besuchten Knoten zurück routen.
Deswegen nutzt der BeeJamA-Algorithmus die Gibbs-Boltzmann-Verteilung um diesem Verhalten entgegen zu wirken:

$$p_x = \frac{e^{X_i}}{\sum_k e^{X_k}}$$

In vielen der erstellten Routingstrategieen ist diese Verteilung eine sehr gute Wahl. Dem interessierten Leser sollte bewusst sein, dass auf ein in manchen Strategien genutzter Schwellwert vor der Norma-

lisierung abgefragt werden muss. Zur weiteren Erklärung zum Aufbau einer Routingstrategie sei auf Kapitel 7.3.2 verwiesen.

Die *Operatoren* des BeeJamA-Algorithmus

Wie bereits in vorherigen Kapiteln beschrieben basiert der Simulator auf verschiedene *Operatoren*. Der BeeJamA-Algorithmus nutzt zum Routen drei verschiedene von *Operator* abgeleitete Klassen:

CommonRouting Die Klasse *CommonRouting* wurde im vorherigen Absatz erläutert.

BeeHiveRouter Der *BeeHiveRouter* übernimmt das eigentliche Routing der Fahrzeuge. Dieser *Operator* wird periodisch vom *RoutingDispatcher* aufgerufen und berechnet bei Bedarf das nächste Teilstück der Route zum Ziel. Es wird im Gegensatz zu manch anderem Routingalgorithmus, wie dem später geschilderten Dijkstra-Algorithmus, nicht die gesammte Route sofort errechnet, sondern bei Anfrage immer die nächste Wegstrecke zur Route hinzugefügt.
Im Bild sieht der Leser das die Funktion *operate()* im wesentlichen drei Untermethoden nutzt:
(Die Erläuterung der Tabellen ist zu finden in)

nextNode(...) Diese Methode sucht nach dem nächsten Knoten welcher sich in derselben IFZArea befindet wie das Fahrzeug.
Der genaue Ablauf ist im Sequenzdiagramm zu sehen.

nextBorderNode(...) Diese Methode wird genutzt, wenn der gesuchte Knoten sich nicht in der IFZArea des momentanen Standpunktes des Fahrzeugs befindet, aber innerhalb der Region des *Navigators*. Dieses bedeutet, dass der gesuchte Knoten sich in einer der Nachbarbereiche des *Navigators* befindet und damit in seiner IFRArea. Die Methode sucht den besten Bordernode und macht dann Gebrauch von der *nextBorderNode()*-Methode wie im Sequenzdiagramm 4.33 zu sehen ist.

nextNavigator(...) Da der BeeJamA-Algorithmus in der aktuellen Konfiguration zwei Ebenen nutzt, muss sich der gesuchte Knoten in einem der Bereiche befinden, welcher sich nicht innerhalb der Region des aktuellen *Navigators* befindet. Daher wird nun in der Netz-Ebene der Weg zum *Navigator* gesucht welche den Bereich des zu suchenden Knoten repräsentiert. Das zu dieser Methode gehörende Sequenzdiagramm ist in Abbildung 4.34 dargestellt. In der hier geschilderten Methode wird der nächste *Navigator* zum Zielbereich gesucht und dann der Methode *nextBorderNode()* übergeben, welche den nächsten Knoten auf der Route errechnet.

Das Klassendiagramm für den *Operator BeeHiveRouter* ist in Abbildung gezeigt und das dazugehörige Sequenzdiagramm in Abbildung .

BeeHiveUpdater Die Klasse *BeeHiveUpdater* hat die Aufgabe alle Tabellen des BeeJamA-Algorithmus periodisch zu aktualisieren.
Wie schon beim Schildern der Klasse *CommonRouting* erwähnt werden nur die IFZArea- und IFRArea-Tabellen mit den aktuellen mittels *Routingstrategie* errechneten Werten gefüllt. Alle anderen Tabellen bauen auf der Grundlage dieser Werte ihre eigenen Einträge auf.
Dieses verhilft dem Algorithmus zu einer kürzeren Updatezeit als wenn man alle Tabellen simultan aktualisiert, hat aber den vermeidlichen Nachteil, dass sich die in den beiden Bereichstabellen geänderten Einträge langsamer als die Tabellen der höheren Ebenen verändern. Die-

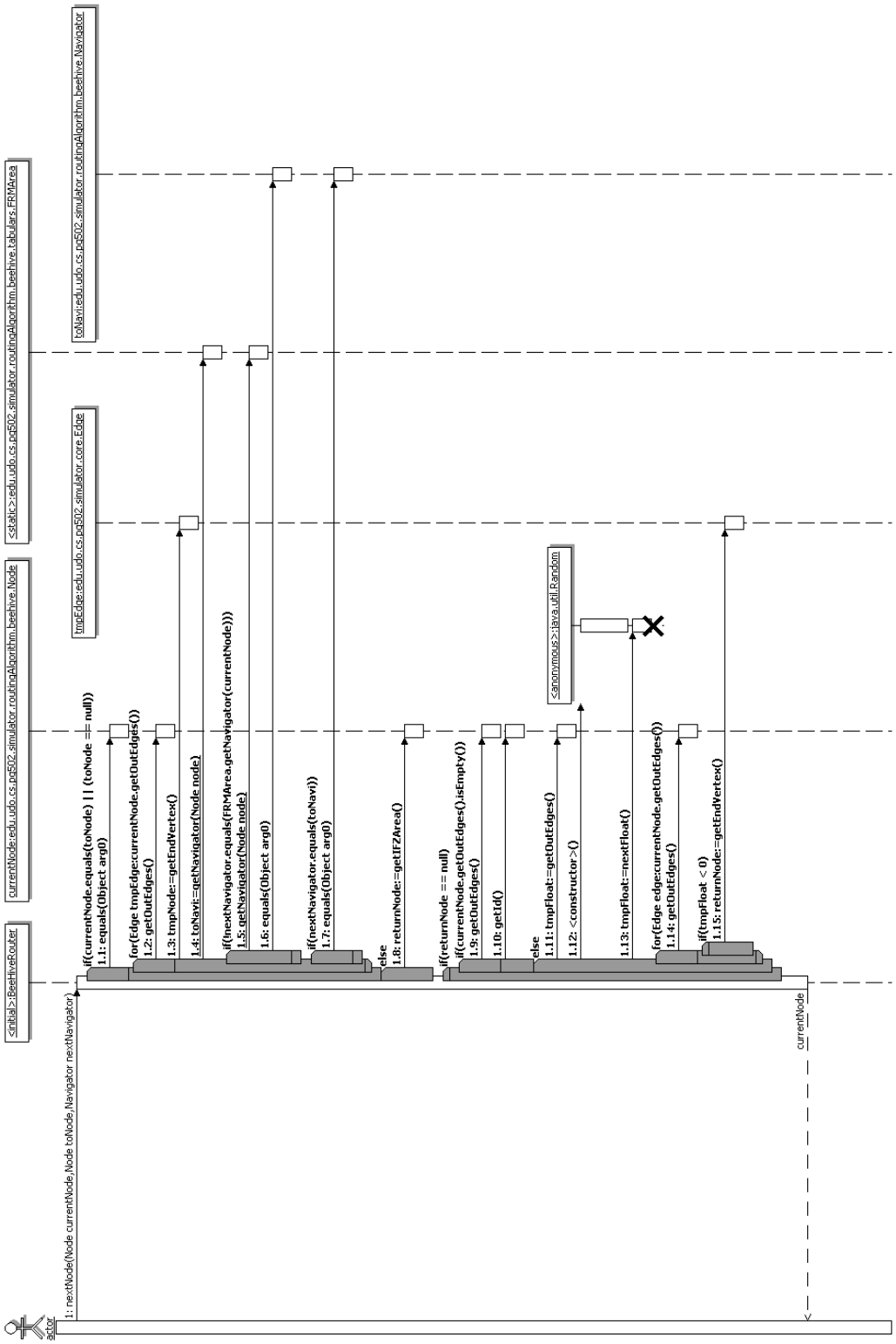


Abbildung 4.32: Sequenzdiagramm BeeHiveRouter.nextNode(...)

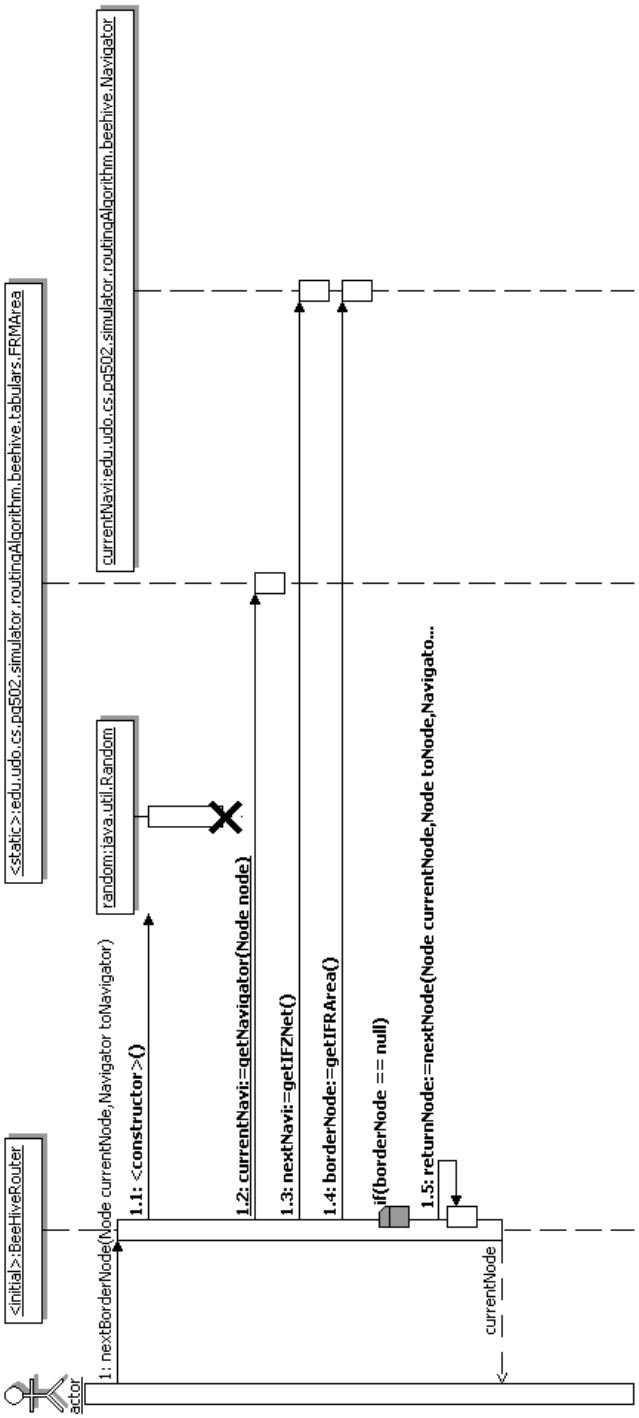


Abbildung 4.33: Sequenzdiagramm BeeHiveRouter.nextBorderNode(...)

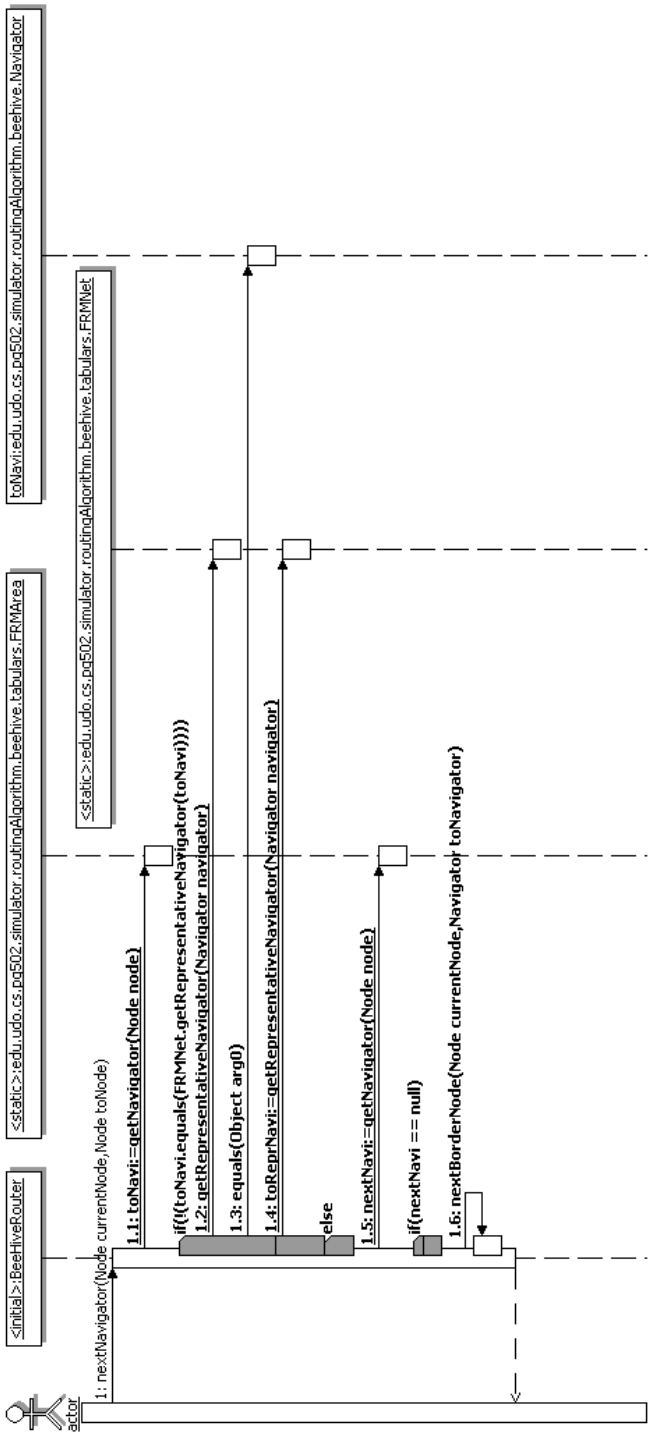


Abbildung 4.34: Sequenzediagramm BeeHiveRouter.nextNavigator(...)

ser Effekt wird von den Autoren toleriert, da durch den langsameren Aktualisierungsprozess wiederum auch nicht jede temporäre Veränderung sofort eine grosse Auswirkung auf das Gesamtsystem hat. So werden nur langfristige Veränderungen in den Tabellen der höheren Ebenen aufgenommen.
Die Klasse BeeHiveUpdater ist sehr übersichtlich aufgebaut da das Update der einzelnen Tabellen in einzelnen Klassen modularisiert wurde. Damit delegiert dieser *Operator* die Aufgaben an die jeweilige UpdateKlasse.
Eine genauere Beschreibung des Aktualisierungsvorgang für die einzelnen Tabellen ist in Abschnitt 2.4.6 zu ersehen.

4.11.2 Implementierung des Dijkstra-Algorithmus

Der Dijkstra-Algorithmus ist innerhalb des Simulators in zwei Klassen implementiert worden:

1. Die eigentliche Implementierung des Algorithmus geschieht in der Klasse *Dijkstra*.

2. Die Verbindung mit dem Simulator regelt der *DijkstraOperator*.

Um den Dijkstra konfigurierbar gegenüber den Qualitäten des BeeJamA-Algorithmus zu gestalten kann dieser mittels einer eigenen Klasse aktualisiert werden welche von der abstrakten Klasse *CommonCostCalculation* erbt.

Die Klasse *Dijkstra*

Die Klasse Dijkstra besteht aus einer Referenz zu der Dijkstra-Implementierung der Bibliothek jGraphT [10]. Diese Referenz wurde mittels Singleton-Pattern erstellt um die rechenintensive Initialisierung nur einmalig auszuführen. In der Methode *getRoute(source, destination)* wird zu einer gegebenen Quelle eine Route zu einem Ziel zurückgegeben.
Um den Algorithmus anpassbar zu gestalten wurde die Methode *updateWeights()* hinzugefügt in welcher der Graph die aktuell errechneten Werte einer beliebigen Routingstrategie gestellt bekommt. Dort wird der in der Kante gespeicherte Wert für den Dijkstra ausgelesen und die jeweilige Kante der Dijkstrareferenz zugeführt.
Eine Erläuterung der genutzten Dijkstra-Implementierung gibt Kapitel 7.5.

Der *DijkstraOperator*

Die von *Operator* abgeleitete Klasse *DijkstraOperator* prüft ob ein Fahrzeug eine Route von der Dijkstra-Instanz benötigt und delegiert die Anfrage an diese. Ebenso aktualisiert der *Operator* die Kosten der Kanten in der Dijkstra-Referenz je nach eingestelltem Updatezyklus.

4.12 Implementierung der Data and Heuristic Component (DHC)

4.12.1 Der Operator *Dataparser*

Der Operator *Dataparser* dient dazu, ein in Knoten-, Kanten- und Intermediatedateien gegebenes Straßennetz einzulesen und daraus ein neues *World*-Objekt inklusive den Instanzen der Typen *Vertex* und *Edge* zu erstellen.

Der *Dataparser* funktioniert nach folgendem Prinzip:

1. Zuerst wird die Knotendatei vollständig eingelesen. Von jedem Datensatz wird für den Graph ein neuer Knoten und ein neuer Intermediate erstellt und diese in zwei *HashMap*s abgelegt, damit später anhand der IDs wieder auf diese zugegriffen werden kann.
2. Anschließend wird die Kantendatei eingelesen und für jeden Datensatz ein (bei Einbahnstraßen) bzw. zwei neue Kanten erstellt und anhand der IDs mit den dazu gehörigen Knoten und Intermediates verknüpft.
3. Abschließend wird die Intermediatedatei eingelesen und die Liste von Intermediates der jeweiligen Kante vervollständigt.

Interne Umrechnung der Koordinaten

Wie in Abschnitt 3.1.1 beschrieben, werden die Koordinaten der Intermediates als geographische Koordinaten in Form von geographischer Länge und Breite gespeichert. Im Gegensatz dazu benötigt die Visualisierungskomponente (s. Abschnitt 4.9) aber zweidimensionale kartesische Koordianten. Es ist also nötig, die geographischen Koordinaten auf eine zweidimensionale Fläche zu projizieren. Da jeweils nur einen sehr kleiner Teil der gesamten Erdoberfläche (z.B. das Ruhrgebiet) betrachtet wird, läßt sich diese Umrechnung leicht durchführen, indem die geographischen Koordinaten in dreidimensionale kartesische Koordinaten umgerechnet und davon jeweils nur die X- und die Y-Koordinate betrachtet werden. Allerdings ist zu beachten, daß die geographischen Koordianten zuvor soweit rotiert werden müssen, daß sich ihre Breiten und Längen im Bereich von ungefähr +/- 5 Grad befinden, damit die durch die Projektion auftretenden Verzerrungen nicht allzu groß werden.

Algorithmus 4 Umrechnung von Kugelkoordinaten in kartesische Koordinaten

1: $Erdradius := 6371009$
2: $Rotation\_Breite := 51.405$
3: $Rotation\_Laenge := 7.005$
4: $x := cos(Breite - Rotation\_Breite) * sin(Laenge - Rotation\_Laenge) * Erdradius$
5: $y := sin(Breite - Rotation\_Breite) * Erdradius$

4.12.2 Der Operator *HeuristicSpeedSetter*

Wie weiter oben angesprochen, muss ein Teil der notwendigen Daten des Straßennetzes heuristisch gesetzt werden. Der Operator *HeuristicSpeedSetter* dient dazu, die Höchstgeschwindigkeit eines Stra-

Tabelle 4.2: Zusammenhang zwischen Straßentyp und Höchstgeschwindigkeit

Autobahn	200 km/h
autobahnähnliche Bundesstraße	130 km/h
Bundesstraße	100 km/h
Regionalstraße	80 km/h
Stadtstraße	50 km/h
andere Straßen	50 km/h

ßensegments zu bestimmen. Der jeweilige Zusammenhang zwischen Straßentyp und der dazugehörigen Höchstgeschwindigkeit läßt sich in Tabelle 4.2 nachlesen.

4.12.3 Der Operator *HeuristicDriveUpCreator*

Ein weiteres Problem stellen Autobahnauffahrten dar. In der ursprünglichen *AND Global Road Data* sind sowohl die Autobahnen, als auch die Autobahnauffahrten, als auch die Autobahnzubringer vom Straßentyp *1* (Autobahn). Für das Verkehrsmodell ist es aber notwendig, dass der Bereich einer Autobahn, an dem sich eine Beschleunigungsspur befindet, den speziellen Typ *7* (Autobahn inkl. Auffahrt) erhält. Dieses resultiert daraus, dass das Auffahren auf eine Autobahn mehr einem Spurwechsel als einem Abbiegen gleicht und gesondert behandelt werden muss.

Leider ist es nicht trivial, mithilfe der aus den Basisdaten gegebenen Informationen die Autobahnauffahrten zu bestimmen. Folgender Ansatz führte aber in der Mehrheit der getesteten Fällen zu einem zufriedenstellenden Ergebnis:

Der *HeuristicDriveUpCreator* sucht nach allen Knoten, auf den zwei als Autobahn typisierte Kanten hinzuführen. Anschließend wird anhand der *Intermediates* überprüft, welche der beiden Kanten stärker gekrümmt ist. Diese Kante erhält nun den Typ *4* (Regionalstraße), wohingegen die andere Kante weiterhin vom Typ *1* (Autobahn) ist. Abschließend erhält die von besagtem Knoten abgehende Kante den Typ *7* (Autobahn inkl. Auffahrt).

Allerdings enthält diese Methode zwei Vereinfachungen. Zum einen wird davon ausgegangen, dass die Kante, die mehr *Intermediate*s enthält, auch stärker gekrümmt ist. Eine Überprüfung der tatsächlichen Krümmung der Kanten wird nicht vorgenommen. Zum anderen werden nur einfache Autobahnauffahrten mit zwei eingehenden und einer ausgehenden Kante bearbeitet. Komplexere Situationen, wie Autobahnkreuze bleiben unberührt, da diese nochmals deutlich schwerer zu handhaben sind.

Andere versuchsweise getestete Methoden führten leider zu deutlich schlechteren Ergebnissen. So ist es aufgrund großer Ungenauigkeiten der *AND Global Road Data* z.B. nicht möglich zu sagen, dass sich Autobahnauffahrten immer rechts der Autobahn befinden. Häufig kreuzt die Auffahrt die Autobahn zuerst, um dann von der linken Seite her auf die Autobahn zu führen.

4.12.4 Der Operator *HeuristicLaneCreator*

Ebenfalls enthalten die Basisdaten keine Angaben über die Anzahl der Fahrspuren. Der *HeuristicLaneCreator* erzeugt die benötigte Anzahl an *Lane*s mithilfe des Straßentyps. Wieviele Fahrspuren jeweils erzeugt werden, wird aus Tabelle 4.3 ersichtlich.

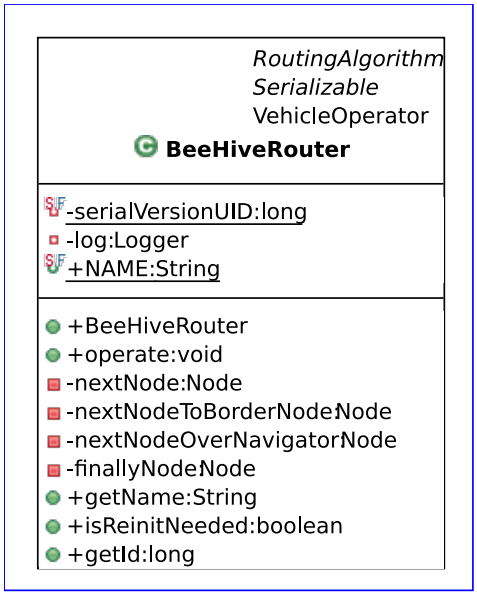


Abbildung 4.35: Die Klasse BeeHiveRouter

Tabelle 4.3: Zusammenhang zwischen Straßentyp und Anzahl Fahrspuren

Autobahn	3 Spuren
autobahnähnliche Bundesstraße	2 Spuren
Bundesstraße	2 Spuren
Regionalstraße	1 Spur
Stadtstraße	1 Spur
andere Straßen	1 Spur

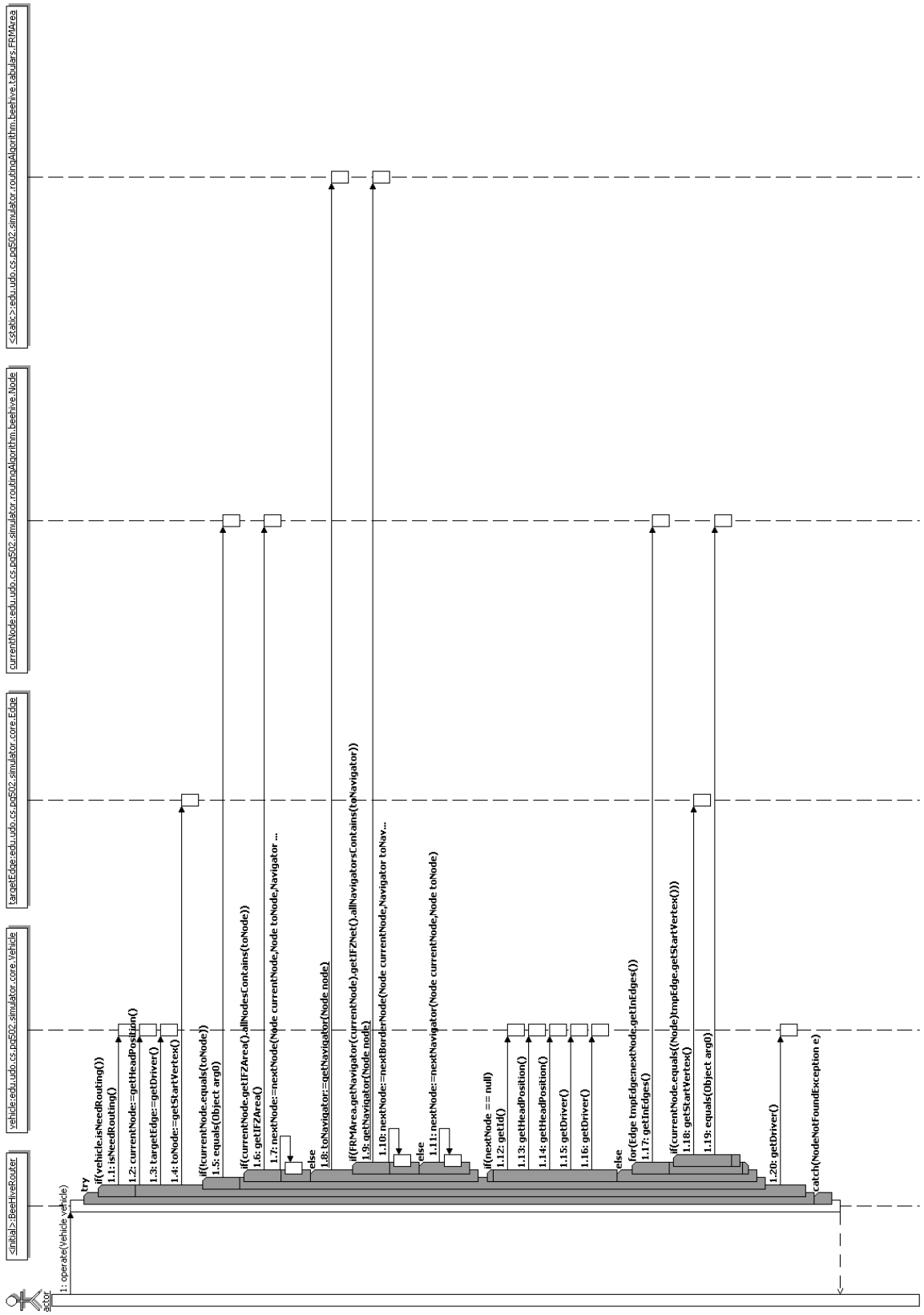


Abbildung 4.36: Sequenzdiagramm BeeHiveRouter.operate()

4.12.5 Der Operator *OppositeEdgeFinder*

Ziel des *OppositeEdgeFinder*s ist es, zu jeder Kante deren Gegenkante (Gegenfahrspur) zu finden. Verwendung kann dieses z.B. beim Simulieren von sogenannten Gafferstaus bei Unfällen finden.

Beim Bestimmen der Gegenkanten wird zwischen zwei Fällen unterschieden: dem trivialen Fall, dass bei Kante und Gegenkante bloß Start- und Endknoten vertauscht sind oder aber dem nicht-trivialen Fall, wie er oft bei Autobahnen auftritt, da dort beide Fahrtrichtungen häufig eigene, disjunkte Knotenmengen besitzen. Zwei Kanten sind nur dann Gegenkanten, wenn der Abstand ihrer Start- bzw. End-Knoten sowie ihre Richtungsdifferenz innerhalb einer Schwelle von 50 Metern bzw. 20 Grad liegt.

Listing 4.1: Der triviale Fall

```
for (Edge edge1 : DiscreteTrafficSimulator.getWorld().getEdges()){
        if (edge1.getOppositeEdge()==null){
                for (Edge edge2 : edge1.getEndVertex().getOutEdges()){
                        if (edge1.getStartVertex().equals(edge2.getEndVertex())){
                                edge1.setOppositeEdge(edge2);
                                edge2.setOppositeEdge(edge1);
                        }
                }
        }
}
```

Listing 4.2: Der nicht-triviale Fall

```
for (Edge edge1 : DiscreteTrafficSimulator.getWorld().getEdges()){
        int dist=50;
        int angle=20;
        if (edge1.getOppositeEdge()==null){

                for (Edge edge2 : DiscreteTrafficSimulator.getWorld().getEdges()){
                        if (edge2.getOppositeEdge()==null){
                                if ((BeeHiveHelper.getDistance(edge1.getStartVertex(), ed
                                        &&(BeeHiveHelper.getDistance(edge1.getEndVertex()
                                        &&(Math.abs(Math.round(((Edge2d)edge1).getHeading
                                        edge1.setOppositeEdge(edge2);
                                        edge2.setOppositeEdge(edge1);
                                }
                        }
                }
        }
}
```

Kapitel 5

Experimente

5.1 Simulationsaufbau

Innerhalb der Projektgruppe wurde der entwickelte Simulator dazu verwendet, den *BeeJamA* Algorithmus in verschiedenen Szenarien auszuwerten und mit den bekannten Routingalgorithmen zu vergleichen. Sowohl einfache Modelle, wie zum Beispiel das Wabenmodell, als auch realistischere Straßenmodelle basierend auf topologische Verkehrsdaten vom deutschen Ruhrgebiet wurden zusammengestellt. BeeJamA wurde mit dem Dijkstra basierten *Shortest Path* Algorithmus verglichen, der heutzutage in den meisten Navigationssystemen, kombiniert mit regelmäßig aktualisierten Verkehrsinformationen, vorhanden ist. In den meisten europäischen Staaten, wird *traffic message channel broadcasting* (TMC) dazu verwendet, Navigationssysteme mit aktuellen Verkehrsinformationen zu versorgen. TMC Aktualisierungen finden meistens jede 5 bis 20 Minuten statt. Aus komerziellen Gründen werden diese Aktualisierungen bis 20 Minuten weiter verschoben. In den Simulationen wurde angenommen, dass Aktualisierungen für das Dijkstra basierte Routing etwa jede 10 Minuten zur Verfügung stehen. In den Auswertungen wurden sowohl individuelle Fahrtzeiten, als auch die Vermeidung von Stausituationen im globalen System berücksichtigt. Das Wabenmodell stellt ein stark vereinfachtes Netzwerkmodell dar. Es wurde am Anfang in der Projektgruppe zu Testzwecken verwendet. Das Modell besteht aus sechseckfoermigen Einheiten und ist beliebig erweiterbar. Ein Ausschnitt einer Einheit ist in Abbildung 5.1 dargestellt.

Ein solches Wabenelement wird als ein Bereich betrachtet. Einzelne Bereiche sind durch je zwei Kanten verbunden, jede Kante gehört zu einem der Bereiche. In Abbildung 5.1 gehören alle roten Kanten zum abgebildeten Bereich. Die grünen bzw. blauen Kanten gehören zu den Nachbarbereichen und sind auf diese gerichtet. Im Rahmen der Experimente wurde auf einem Teilgebiet des Ruhrgebiets auf vier Knoten Verkehr erzeugt. Alle Fahrzeuge sollten ein gemeinsames Ziel erreichen. Die Experimente wurden für *BeeJamA* und *Dijkstra* je zehn mal durchgeführt. Die Durchschnittsgeschwindigkeit von jedem Verkehrsweg wurde während den Versuchen verfolgt.

Die Eingabedaten der Experimente sowie die Parametereinstellungen der Bewertungsfunktion der Verkehrswege sind in den Tabellen 5.1 und 5.2 dargestellt.

Das Verkehrsszenario besteht aus zwei Autobahnen. Die Kanten a und b entsprechen der deutschen Autobahn A40 von Dortmund nach Bochum und die Kante c stellt die Autobahn A45 dar, die den

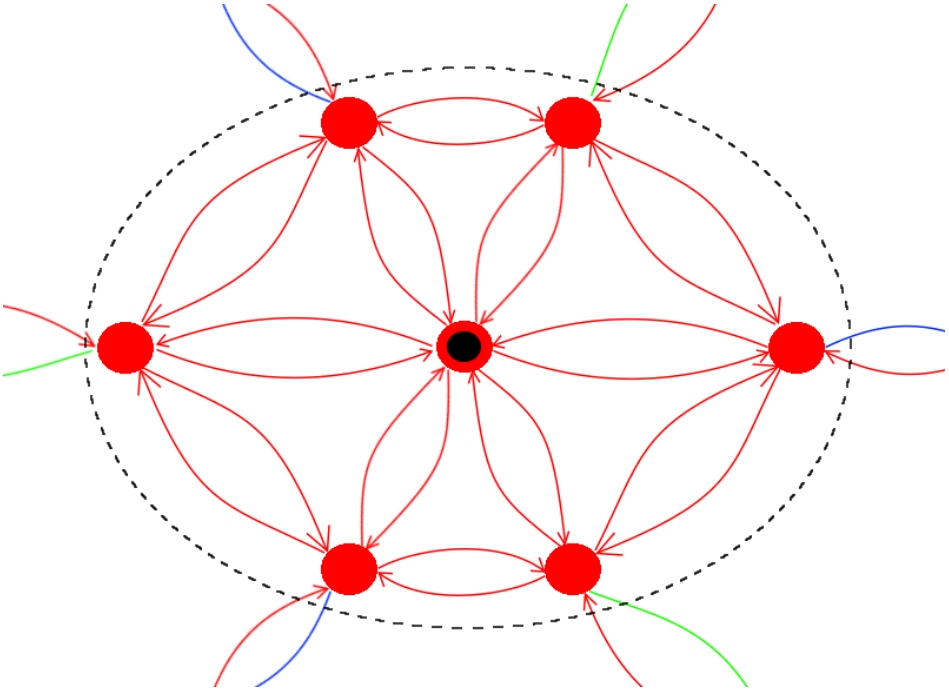


Abbildung 5.1: Wabenmodelleinheit

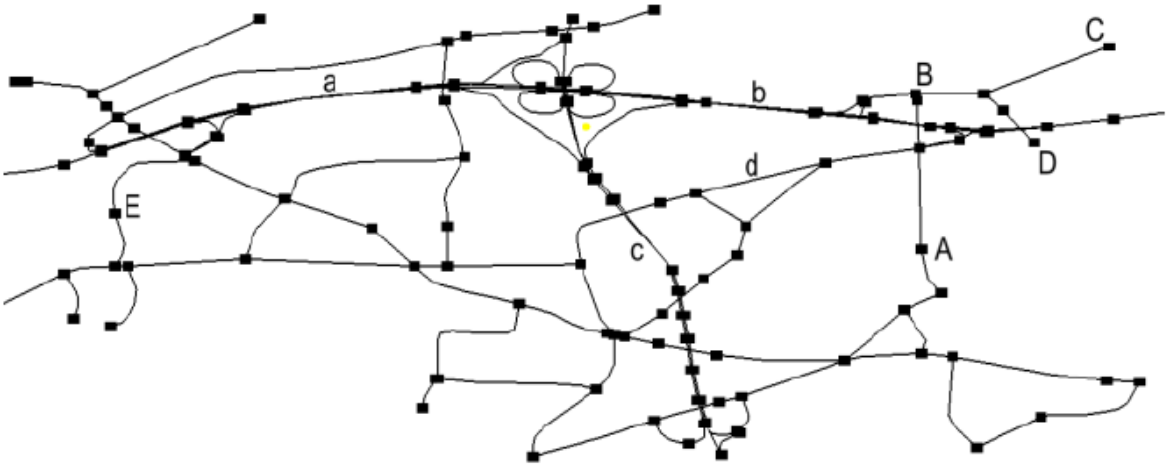


Abbildung 5.2: Einfaches Simulationsszenario eines Teils des Ruhrgebiets

Tabelle 5.1: Konfiguration des Simulators

Ausgangsknoten	A, B, C, D
Zielknoten	E
Neü Fahrzeuge pro Knoten	3 (1 pro Knoten)
Simulationsdaür	3600 Sekunden
Dijkstra Aktualisierungsintervall	600 Sekunden
Geschwindigkeitsbegrenzung	135 km/h (Autobahn)
	80 km/h (Landstraße)
Höchstgeschwindigkeit der Fahrzeuge	135 km/h

Tabelle 5.2: Einstellungen der Fahrzeugdichten

Landstraße (Typ 2 Straße)	$\alpha = 35$, $\beta = 40$ [Fahrzeuge/km]
	A=50, B=10 [km/h]
6-streifige Autobahn (Typ 1 Straße)	$\alpha = 40$, $\beta = 55$ [Fahrzeuge/km]
	A=70, B=30 [km/h]
4-streifige Autobahn (Typ 1 Straße)	$\alpha = 40$, $\beta = 55$ [Fahrzeuge/km]
	A=70, B=30 [km/h]

nördlichen mit dem südlichen Teil von Dortmund verbindet. Die restlichen Straßen werden als Landstraßen bewertet.

5.2 Beobachtungen

Im dem Versuch mit dem Dijkstra basierten Routingalgorithmus, der an den Knoten A, B, C und D erzeugte Verkehr wird am Anfang auf beiden Kanten a und b gesteuert. Alle Fahrzeuge sollen das Endziel E erreichen. Nach 300 Sekunden erscheinen lokale Stausituationen mit etwa 10-15 beteiligten Fahrzeugen, während der Rest des Verkehrs weiterhin fließend bleibt. Nach etwa 500 Sekunden sammeln sich Fahrzeuge auf beiden Kanten a und b an und Stau wird erzeugt. Deshalb werden jetzt Fahrzeuge von den Quellknoten A, C und D auf die Kante d geroutet, auf der aktuell weniger Verkehr ist und deshalb mit einer höheren Qualität, bzw. nierdrigere Kosten, bewertet wurde. Fahrzeuge vom Knoten B werden weiterhin auf der Kante b geroutet, da diese nicht von den Staus auf die Autobahnen, die mit b benachbart sind, betroffen werden. Nach etwa 800 Sekunden: Fahrzeuge mit niedrigen Fahrtzeiten von etwa 220 Sekunden, die auf der leeren Kante d geroutet wurden kommen an ihrem Ziel an. In derselben Zeit kommen verspätete Fahrzeuge, die über der Kante a geroutet wurden mit Fahrtzeiten über 500 Sekunden an ihrem Ziel an. Verkehr sammelt sich schnell auch auf dieser Kante d an und Fahrtzeiten werden höher. In dieser Zeit lösen sich Staus auf den Autobahnen auf und die Fahrtzeiten der Fahrzeuge die vom Knoten B starten verbessern sich messbar. Nach 1200 Sekunden werden Fahrzeuge erneut über die Autobahnen geroutet. Dieser oszillierende Effekt wird in den Graphen 5.3 und 5.5 abgebildet. Wenn BeeJamA verwendet wird, dann werden Fahrzeuge dynamisch sowohl auf Autobahnen als auch auf Landstraßen geroutet. Am Anfang der Simulation werden Fahrzeuge mit Quellknoten A,B,C und D in Richtung von Autobahn a und der diagonalen Landstraße d geroutet. Sobald sich Fahrzeuge auf den Autobahnen a und b ansammeln und deren Qualität sinkt, werden öfter Landstraßen bevorzugt. Nach etwa 600 Sekunden gibt es einen relativ stabilen, fliessenden Ver-

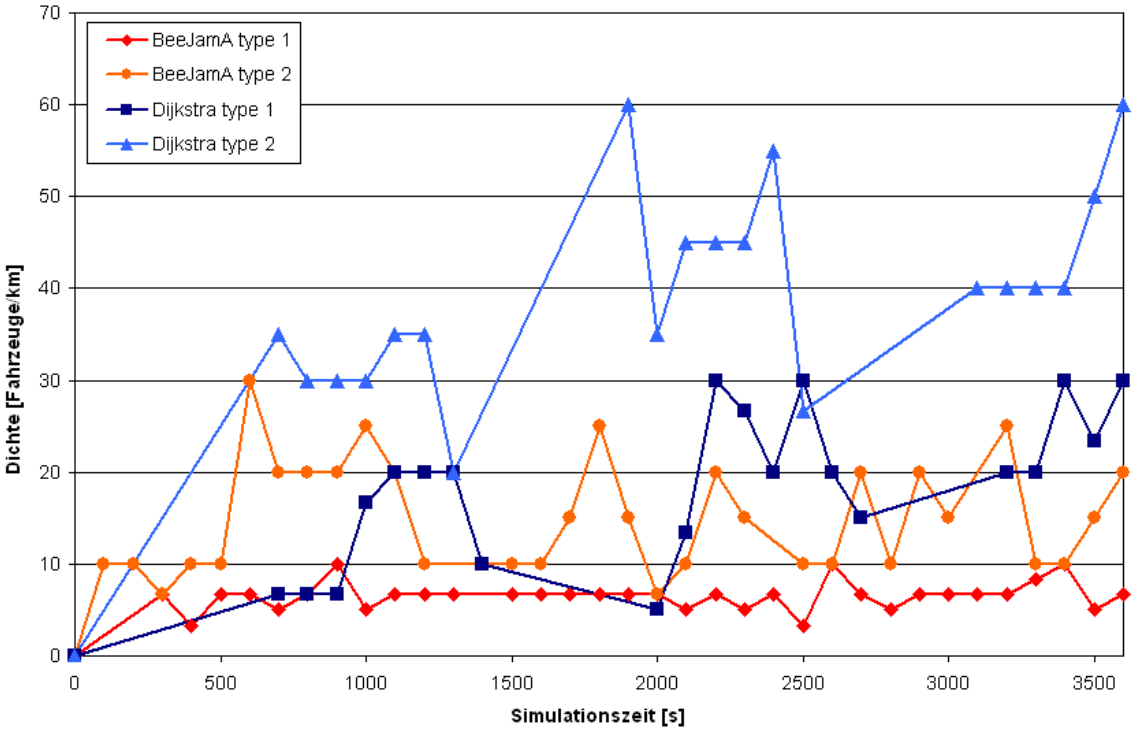


Abbildung 5.3: Fahrzeugdichten

kehr. Diese Situation ändert sich nicht während der restlichen Simulation. Abbildung 5.4 beschreibt die Veränderungen der durchschnittlichen Fahrtzeiten bei der Verwendung von BeeJamA basierten im Vergleich zu Dijkstra basierten Routing. Man bemerkt, dass beim Dijkstra basierten Routing die durchschnittliche Fahrtzeit jedes mal deutlich sinkt, wenn die Route neu berechnet wird. Verwendet man BeeJamA, sind die Fahrtzeiten gleichmässiger und man bemerkt keine grosse Unterschiede.

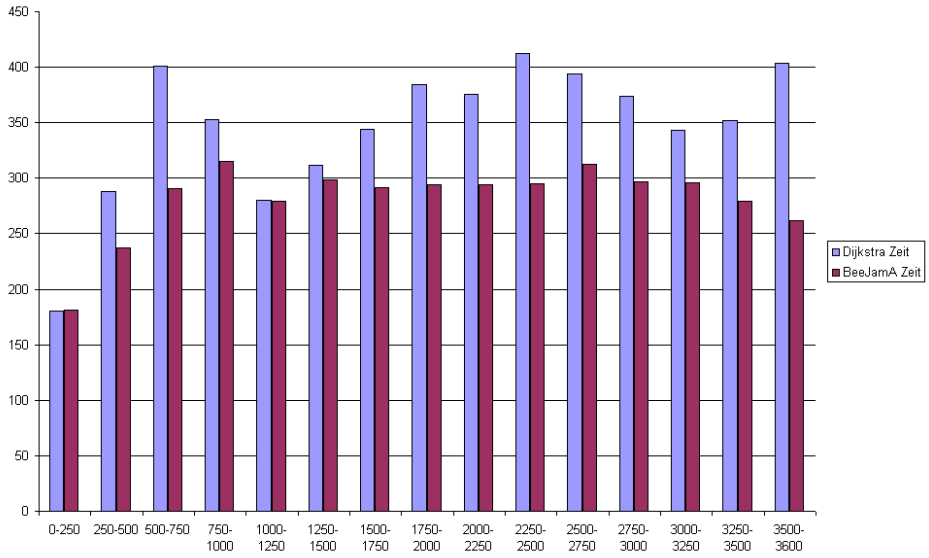


Abbildung 5.4: Durchschnittliche Fahrtzeiten

Nach dem Anfangsintervall von etwa 200 Sekunden sind Verkehrsdichten bei BeeJamA kleiner auf alle beobachteten Straßen, im Vergleich zu den entsprechenden Straßentypen im Dijkstra-Routing. Im Dijkstra basierten Routing erscheinen schwere Stausituationen nach etwa 1800 Sekunden. In diesem Fall liegt eine Verkehrsdichte von 55 Fahrzeuge oder mehr pro Kilometer vor. Wird BeeJamA verwendet, bleibt das System staufrei und die Durchschnittsfahrtzeiten sind kleiner oder höchstens gleich den Durchschnittsfahrtzeiten im Dijkstra basierten Routing.

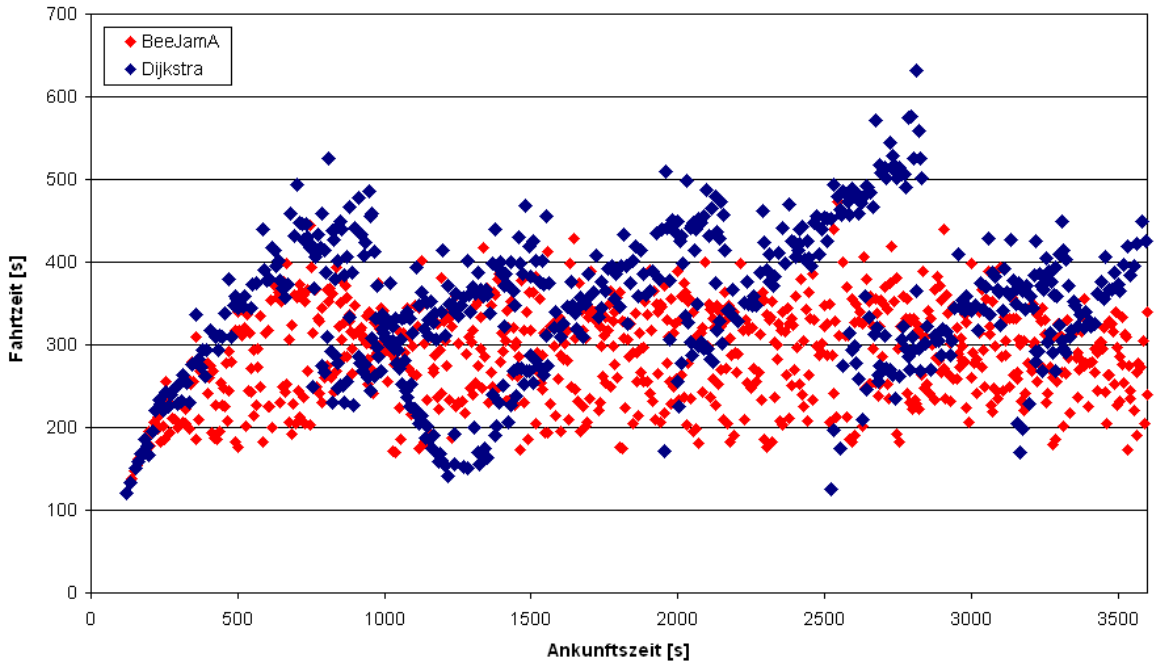


Abbildung 5.5: Individuelle Fahrtzeiten

Kapitel 6

Zusammenfassung und Schlussbemerkungen

6.1 Resümee

Ziel des Projektes war es, verteilte Algorithmen zu entwickeln, welche nicht nur dynamisch und in Koordination mit allen Beteiligten auf vorhandene Staus reagieren, sondern auch im Vornherein ihre Entstehung verhindern. Dabei orientierte sich das Projekt an den von der Natur inspirierten BeeHive Algorithmen und entwickelte ein Multiagentensystem namens BeeJamA.
Das Ziel- und Einsatzgebiet des Projekts ist das Ruhrgebiet, eines der größten und dichtest besiedelten Industriegebiete in Europa. Es besteht aus einer Ansammlung von mehr als zwölf Städten entlang der Ruhr. Während diese Struktur eine sehr hohe, praktische Herausforderung für die Verkehrssteuerung darstellt, ist das sehr dichte und komplexe intra- und interstädtische Straßennetz zugleich ein gutes Einsatzgebiet für schwarmintelligenzbasierende Algorithmen wie BeeJamA. Um ein hoch dynamisches Problem wie das des Verkehrsrouting zu lösen, haben wir innerhalb der Projektgruppe hoch anpassungsfähige Multi-Layer Routing-Algorithmen entwickelt, welche Fahrzeuge next-Hop-basiert in Richtung ihres Zieles leiten. Da die Informationen für den einzelnen Fahrer (rechts, links, geradeaus) schon vor Erreichen der nächsten Teilstrecke verfügbar sein müssen und sich während der Fahrt mehrfach ändern können, müssen die verteilten BeeJamA Operationen in Echtzeit durchführbar sein und Deadlines berücksichtig werden. Der größte Durchbruch wurde erzielt durch die ausführliche Auswertung empirischer Untersuchungen über die Beziehung von Verkehrsdichte, mittlerer Geschwindigkeiten und Verkehrsqualitäten in Bezug auf Staus, wodurch es möglich war mathematische Qualitätsfunktionen für die lokale Verkehrssituation zu definieren. Die entwickelten Algorithmen sind robust, sowohl in Bezug auf unvorhergesehene Ereignisse wie Unfälle oder Straßensperrungen, als auch für den Fall, dass Fahrer sich nicht an die Anweisungen des Systems halten, sei dies beabsichtigt oder nicht. Solche Ereignisse beeinträchtigen das System nur minimal und diejenigen Fahrer, welche sich an die Anweisungen halten, profitieren von den Vorteilen des BeeJamA-Systems. Dies ist einer der wichtigsten Kriterien bei der Einführung von BeeJamA in die Praxis und eine direkte Folge der verteilten Kontrolle.
Die jetzige experimentelle Arbeit hat einen Punkt erreicht an dem ein wesentlicher Bereich des Ruhrgebiets abgedeckt werden kann. Das mehrschichtige Konzept macht es einfach, die Untersuchungen auf ein größeres Gebiet auszudehnen, da BeeJamA hochgradig skalierbar ist. Es ist nicht verwunder-

lich, dass BeeJamA einen großen Vorteil gegenüber traditionellen Routing-Algorithmen in Stausituationen bietet, was die Reslutate aus den Simulationsexperimenten belegen.
Es gibt jedoch noch eine Reihe von Weiterentwicklungen und Untersuchungen die das Projekt noch verbessern oder erweitern können:

Untersuchung der Integrationsdichte, also dem Anteil an Fahrzeugen die unter der Führung von BeeJamA fahren, so dass das System korrekt bzw. zuverlässig genug arbeiten und Staus vermieden werden können.

Anpassungsmöglichkeiten des Verkehrsmodells bzw dessen Regeln, damit die Möglichkeit entsteht, den Simulator um andere Akteure als Fahrzeuge, z.B. Fußgänger, erweitern zu können. Dadurch ließe sich das Einsatzgebiet des Projektes erheblich verweitern.

Sicherheitstechnische Aspekte des Projektes, wie die Angreifbarkeit des Systems von Innen und von Außen heraus können analysiert und geschlossen oder minimiert werden.

Ausbau des verwendeten Datenformats für die Unterstützung von mehr Verkehrselementen wie Schildern, Ampeln, Kreisverkehren o.ä.

Verbesserung der grafischen Oberfläche des Simulators

Hiermit bedankt sich die PG 502 beim Leser dieses Endberichts.

Kapitel 7

Anhang

7.1 Installation

Im folgen werden die einzelnen Schritte angegeben und näher erleutert, die notwendig sind um den Simulator erfolgreich zu installieren.

1. **Download des Simulators**
   Das Projekt STOP ist auf der Homepage con SourceForge unter [4] zu finden und runterzuladen.

2. **Installation von Java**
   Die aktuelle Version ist zu finden unter [9].

3. **Installation einer Entwicklungsumgebung**
   Zur die Erstellung des Simulators wurde die IDE Eclipse verwendet, weshalb sich alle folgenden Schritte, die die IDE betreffen, auf Eclipse beziehen werden. Die aktuelle Version ist zu finden unter [5].

4. **Installation von JoGL**
   JoGL ist die freie OpenGL-Programmbibliothek für Java, welche in dem Simulator in der Version 1.1.0 zum Einsatz kommt um das Stra ßennetz darzustellen. Die aktuelle Version ist unter [11] zu finden. Um JoGL für den Simulator zu installieren muss lediglich die jogl.dll und die jogl_ awt.dll in ein Classpath-Verzeichnis (z.B. unter Windows „C:/ Windows/system32/“) kopiert werden. Die nötige jogl.jar ist in dem Projekt enthalten, daher ist das Kopieren der jogl.jar in das lib/ext/ Verzeichnis der Java-Installation nicht nötig. Die Schritte zur Installation von JoGL aus der Datei Userguide.html brauchen nicht beachtet werden.

5. **Einrichtung der IDE**
   Im Folgenden werden die Schritte aufgelistet die innerhalb der IDE unternommen werden müssen.

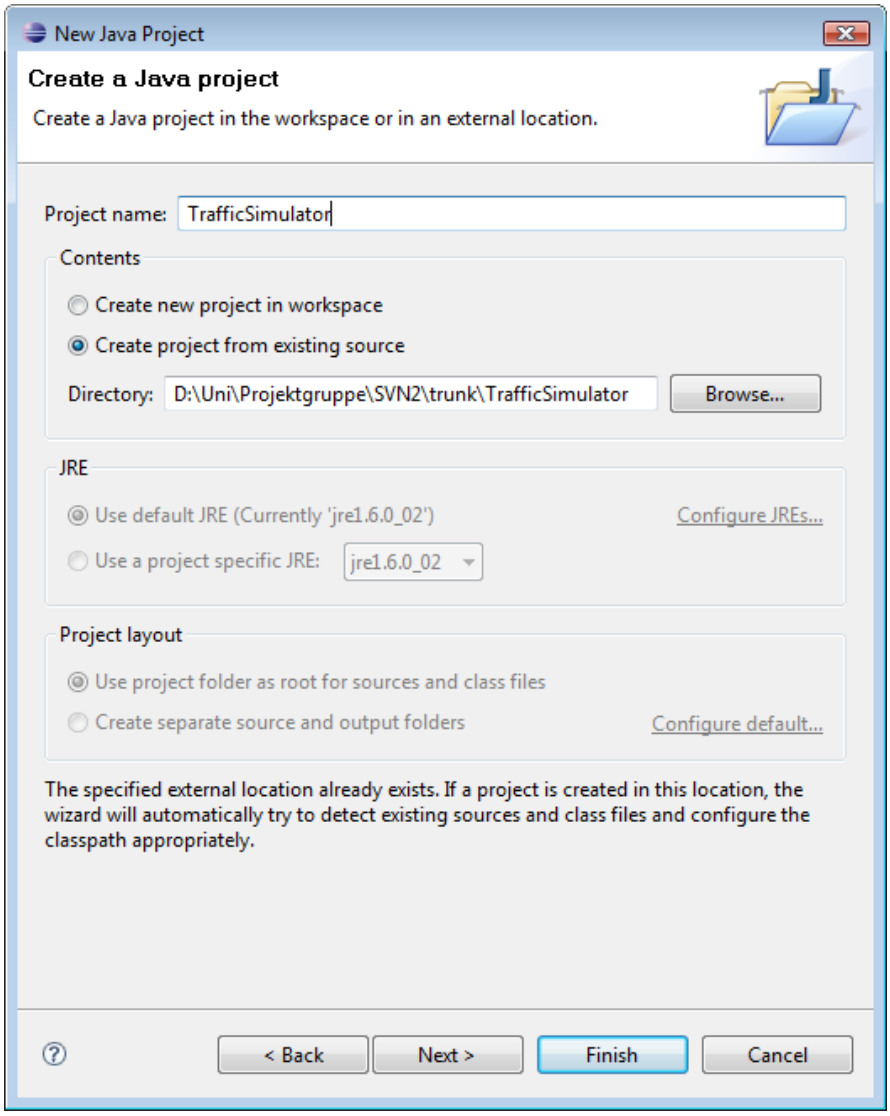


Abbildung 7.1: Installation - Projekteigenschaften

Create new project in worksapce
Ein neues Projekt kann über das Menü *File - New - Project - Java - Java Project* erstellt werden.
Project name
Beliebigen Namen für das Projekt eingeben (z.B. TrafficSimulator).
Create Project from existing source
Unter „Create project from existing source“ muss der Pfad zu den Dateien des Simulators angeben werden(z.B. C:/TrafficSimulator/).
Configure JREs
Für den Simulator ist es erforderlich das „Compiler compliance level“ auf 5.0 zu stellen. Eclipse richtet diese Einstellung normalerweise selbst ein, so dass es hier keiner Handlung des Benutzers bedarf. Um zu kontrollieren ob von Eclipse das richtige „Compiler compliance level“ ausgewählt wurde, kann man dies über das Menü *Project - Properties - Java Compiler* tun.

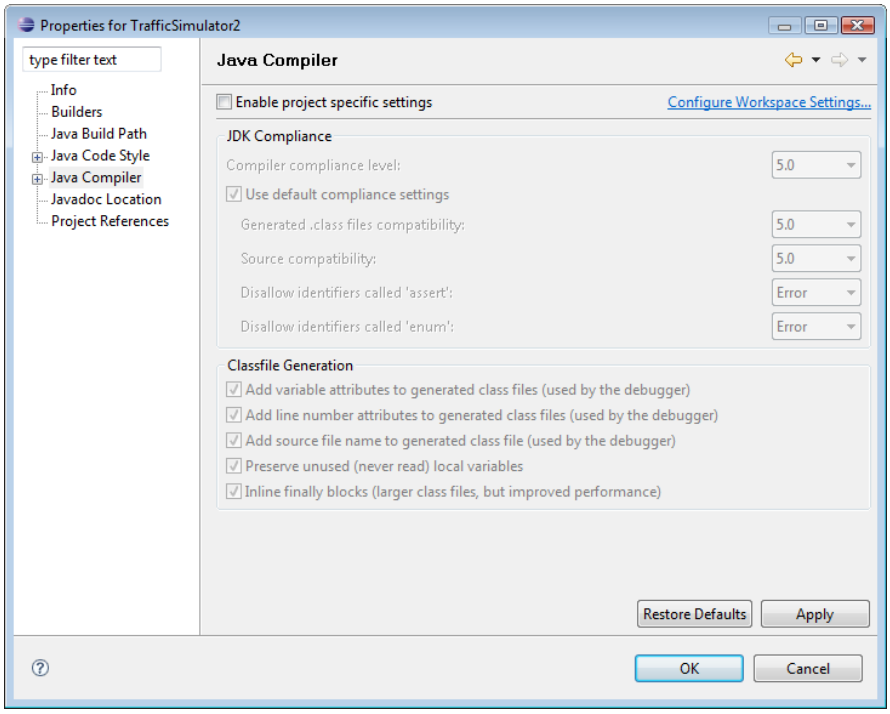


Abbildung 7.2: Installation - JRE konfigurieren

6. **Starten des Simulators**
   Nachdem der Simulator unter Java eingerichtet worden ist, kann er über die Klasse STOPGUI aus dem Paket *gui* gestartet werden.

7.2 Bedienung des Simulators

7.2.1 Simulations-GUI

Mit der Simulations-GUI ist es möglich viele verschiedene Einstellungen für den Simulator vorzunehmen, diese zu speichern und sie zu einem späteren Zeitpunkt wieder zu laden. Zu diesen Einstellungen gehört das Zusammenstellen der Operator-Ketten und der Simulationsparameter wie Anzahl maximaler Schritte oder die Größe der Zellen

Reiter : Configuration

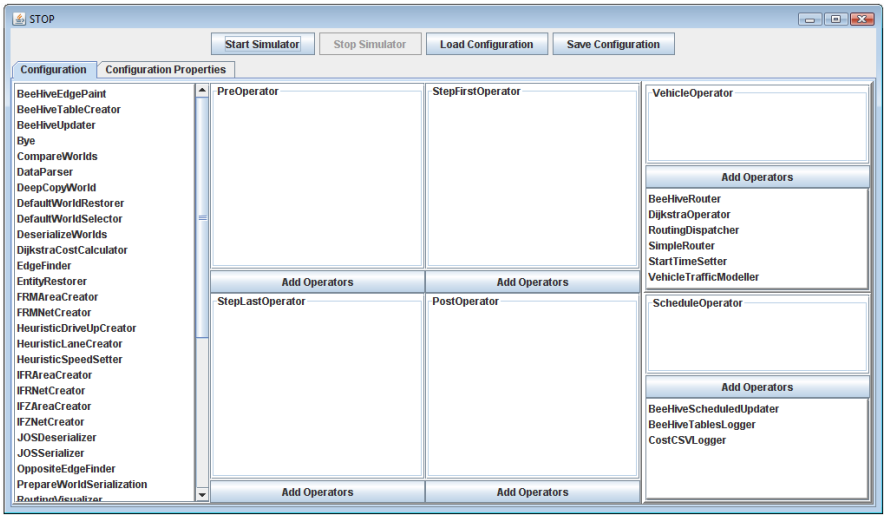


Abbildung 7.3: Configuration

Der erste Reiter („Configuration“) der GUI dient dazu, die Operator-Ketten zusammenzusetzen, welche Teil der Konfiguration des Simulator sind. Dafür gibt es sechs verschiedene Felder in denen die Operatoren der einzelnen Ketten aufgelistet werden. Am rechtem Rand befindet sich eine Liste von den Operatoren, die in eine Pre-, Step- oder Post-Ketten vorkommen können. Für die VehicleOperator-Chain und Schedule-Kette befinden sich die Listen der auswählbaren Operatoren direkt unter den Listen der Ketten. Um ein Operator einer List hinzuzufügen, wird zuerst der gewünschte Operator aus der Liste ausgewählt und danach über den „Add Operator“ Schalter der entsprechenden Kette eingefügt. Das Entfernen von Operatoren aus einer Liste geschieht mit Hilfe der - Entfernen - Taste oder über das Operatormenü, welches sich durch einen Rechtsklick auf den entsprechenden Operator öffnen lässt. Mit Hilfe des Menüs lässt sich auch die Reihenfolge der Operatoren beeinflussen und der „Property Editor“ aufrufen, welcher im folgenden noch weiter beschrieben werden wird.

Reiter : Configuration Properties

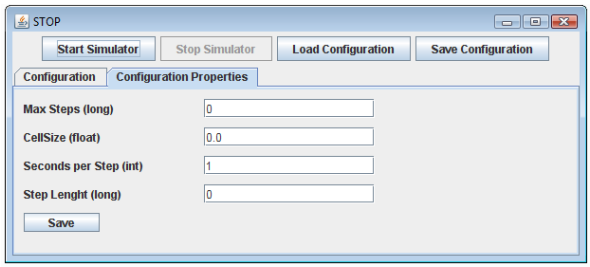


Abbildung 7.4: Configuration Properties

Der zweite Reiter („Configuration Properties“) beinhaltet Einstellungsmöglichkeiten für vier Parameter der Konfiguration des Simulators welche mit dem - Save - Schalter übernommen werden können. Allerdings muss die gesamte Konfiguration noch mit „Save Configuration“ Schalter der obersten Reihe gespeichert werden, damit sie später wieder geladen werden kann.

Property Editor

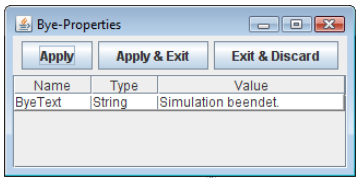


Abbildung 7.5: Property Editor

Der Property Editor ermöglicht es dem Benutzer die Parameter von Operatoren zu modifizieren. Um ihn aufzurufen muss zuerst der entsprechende Operator in eine Operator-Kette eingefügt werden und anschließend über das Operatormenü aufgerufen werden. Der Editor listet alle einstellbaren Parameter auf, welche die Datentypen

String

Integer

Float

Double

Boolean

benutzen. Andere Datentypen wären theoretisch auch editierbar, jedoch würde es sich als schwierig erweisen komplexere Datentypen wie z.B. Fahrzeuge in das Eingabefeld einzugeben. Um einen Parameter zu editieren muss der Benutzer lediglich in das Feld „Value“ des entsprechenden Parameter klicken, den neuen Wert eintragen und auf „Apply“ oder „Apply & Exit“ klicken um die änderungen zu übernehmen.

7.2.2 Bedienung der Visualisierungskomponente

Die Visualisierungskomponente des Simulators dient der Illustration des zugrunde liegenden Graphen und der Fahrzeuge, die auf diesem geroutet werden, um in einer einfachen und anschaulichen Weise das Routingverhalten verschiedener Algorithmen darzustellen. Durch die graphische Darstellung werden Unterschiede zwischen verschiedenen Algorithmen, aber auch Parameteränderungen innerhalb eines Algorithmus einfach erkennbar. Mit der Hilfe dieses Werkzeugs können sicherlich keine repräsentativen und allgemeingültigen Aussagen über die Qualität eines Algorithmus gemacht werden, diesen Zweck hat es aber auch nicht. Es soll viel mehr dazu dienen, dem Entwickler algorithmische Veränderungen in Bezug auf das Routingverhalten in einer direkten Art und Weise zu verdeutlichen. Daher stellt die Visualisierungskomponente in erster Linie ein Mittel zur Unterstützung des Algorithmenentwurfs dar.

Die Visualisierung kann in die Simulation integriert werden, in dem der Operator *RoutingVisualiser* in der *StepFirstOperatorCain* oder der *StepLastOperatorChain* platziert wird.

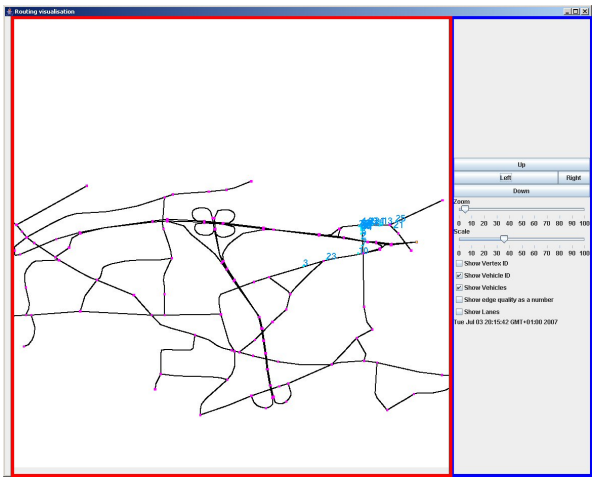


Abbildung 7.6: Die Visualisierungskomponente

Abbildung 7.6 zeigt die Visualisierungskomponente des Simulators, welche aus zwei Teilen besteht. Der rot umrandete Teil von Abbildung 7.6 stellt das Visualisierungssfenster dar und der blau umrandete Teil dient als Benutzerschnittstelle zur Visualisierungskomponente. Das Visualisierungsfenster dient der eigentlichen Darstellung des Graphen bzw. Straßennetzes und der Fahrzeuge. Durch die Benutzerschnittstelle lässt sich das in dem Visualisierungsfenster dargestellte auf die Bedürfnisse des Benutzers anpassen. Im folgenden wird zunächst das in dem Visualisierungsfensters dargestellte erläutert, um danach auf die Bedienung der Benutzerschnittstelle zur Visualisierungskomponente einzugehen.

Das Visualisierungsfenster

Die Hauptaufgabe der Visualisierungskomponente ist die graphische Darstellung des Verhaltens und die damit einhergehende Qualität von Routingalgorithmen. Die Qualitätsansprüche an den Routingalgorithmus ergeben sich direkt aus dem Projektgruppenziel; zum einen sollen Verkehrsstaus vermieden werden und zum anderen sollen alle Fahrzeuge in möglichst kurzer Zeit und mit möglichst wenigen Umwegen ihr gewünschtes Ziel erreichen. Dieses Kapitel zeigt, wie die Visualisierung zur Unterstützung des Entwicklungsprozess genutzt werden kann, um einen schnellen Einblick in die Eigenschaften eines Routingalgorithmus in Bezug auf Stauvermeidung und Routingverhalten zu bekommen. Um dies zu bewerkstelligen ist es notwendig, dass die Visualisierungskomponente das Straßennetz auf dem geroutet wird und die Bewegungen der zu routenden Fahrzeuge graphisch darstellen kann. Abbildung 7.7 zeigt einen Ausschnitt aus einer Straßenkarte (bzw. aus einem Straßennetz). Die blauen Quadrate repräsentieren die Fahrzeuge und die schwarzen Linien stellen das Straßennetz dar. Des weiteren hat jede Straße, wie in Kapitel 4.5 erläutert, zwei Knoten, diese werden durch die pinken Quadrate gekennzeichnet. Damit stellen die pinken Quadrate entweder Anfangs- bzw. Endpunkte von Straßen oder Verbindugnspunkte, wie zum Beispiel Kreuzungen, zu anderen Straßen dar. Da sich Straßen nur durch die entsprechenden Identifikationsnummer (ID) der Start- und Endknoten identifizieren lassen, ist es möglich die IDs der Knoten einzublenden. Abbildung 7.8 zeigt eine Straße, die durch

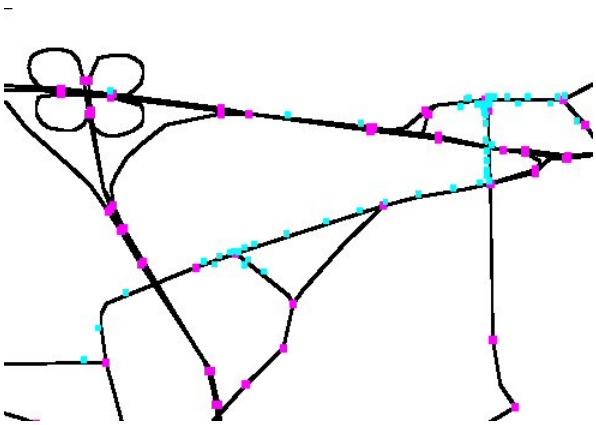


Abbildung 7.7: Ausschnitt aus einer Straßenkarte

Abbildung 7.8: Identififizierung von Straßen und Fahrzeugen

die Knoten mit den IDs 20 und 100 beschrieben wird. Neben den Knoten haben auch die Fahrzeuge eine eindeutige Identifizierungsnummer, diese wird durch die blaue Nummer an den blauen Quadraten dargestellt. So befinden sich in dem Beispiel aus Abbildung 7.8 drei Fahrzeuge, mit den IDs 15, 7 und 6 auf der vorher beschriebenen Straße.

Farben der Fahrzeuge Wie oben beschrieben wurde, werden Fahrzeuge im Allgemeinen durch blaue Quadrate dargestellt. Die Farbe der Fahrzeuge kann sich jedoch ändern, da diese vom verwendeten Routingalgorithmus abhängig ist. Das bedeutet, dass sich durch die Farbgebung der Quadrate der verwendete Routingalgorithmus identifizieren lässt. Ein blaues „Fahrzeug“ deutet dabei auf die Verwendung des BeeJamA-Algorithmus hin. Des Weiteren ist der im Kapitel 7.5 erläuterte Algorithmus von Dijkstra ein zweiter, zum vergleichen implentierter Routingalgorithmus. Die farbliche Darstellung für Fahrzeuge, welche mit diesem Algorithmus geroutet werden, ist gelb.
Die in den Abbildungen 7.7 und 7.8 vorgestellten Ausschnitte aus Straßenkarten entsprechen einer makroskopischen Ansicht. Um ein gewisses Maß an Übersichtlichkeit zu bewahren, werden in dieser Ansicht Details wie die „Straßenqualität“, die Richtung und die Anzahl der Spuren einer Straße ausgeblendet. Diese Details werden erst in einer mikroskopischen Ansicht erkennbar, in die automatisch Umgeschaltet wird, wenn der Zoom einen festgelegten Grenzwert überschreitet (vgl. Kapitel 7.2.2). Im Folgenden wird die in Abbildung 7.9 dargestellte mikroskopische Kartenansicht näher erläutert.

Richtung einer Straße In der mikroskopischen Ansicht werden Straßen durch eine oder zwei farbige Linien, abhängig davon, ob die Straße uni- oder bidirektional ist und einer schwarzen Linie, welche die Bedeutung einer Hilfslinie hat, dargestellt. Die farbliche Bedeutung ist an dieser Stelle nicht von

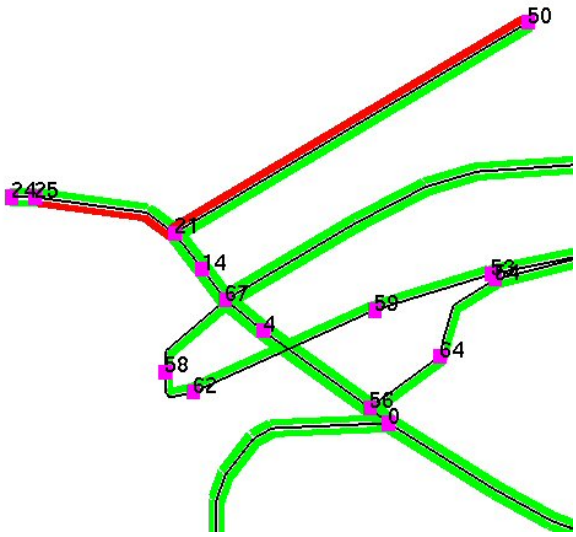
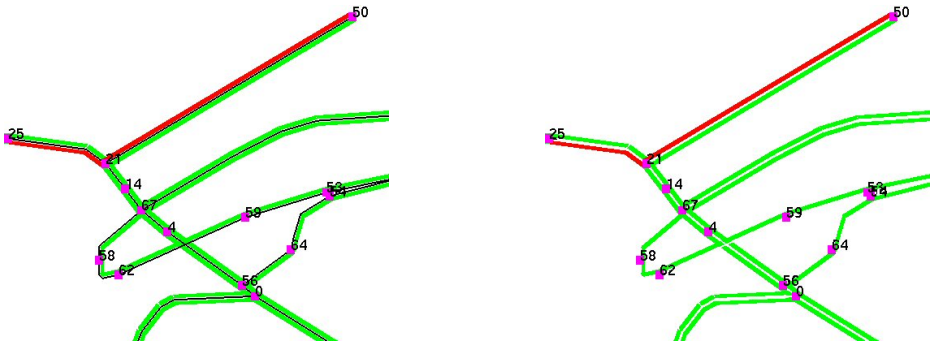


Abbildung 7.9: Mikroskopische Ansicht

Belang und kann daher zunächst vernachlässigt werden. Abbildung 7.9 zeigt eine bidirektionale Straße, die durch die Knoten 50 und 21 beschrieben wird. Diese Straße führt sowohl vom Knoten 50 zum Knoten 21 als auch vom Knoten 21 zum Knoten 50. Die durch die Knoten 58 und 67 beschriebene Straße ist dagegen unidirektional, da es lediglich eine farbige Linie zwischen den beiden Knoten gibt. Das Problem das sich nun ergibt, ist die Interpretation der Richtung einer unidirektionalen Straße. So ist nicht direkt klar, ob die Straße vom Knoten 58 zum Knoten 67 führt oder umgekehrt. Hier bekommt die Hilfslinie eine besondere Bedeutung, erst durch sie wird eine Richtungsinterpratetion möglich. Da die Straße ausgehend von Knoten 58 rechtsseitig von der Hilfslinie liegt, kann die Richtung als vom Knoten 58 (Startknoten) zum Knoten 67 (Endknoten) interpretiert werden. Das bedeutet im Allgemeinen für die Richtungsbestimmung einer unidirektionalen Straße, dass derjenige Knoten als Startknoten zu interpretieren ist, wenn ausgehend von diesem Knoten die farbige Linie rechts von der Hilfslinie liegt. Der entsprechende zweite Knoten wird als Endknoten bezeichnet. Selbiges Verfahren kann auch zur Richtungsbestimmung der verschiedenen Straßenseiten von bidirektionalen Straßen angewandt werden. Abbildung 7.10 verdeutlicht nochmals den Sinn der Hilfslinie. Während



(a) Mit Hilfslinie (b) Ohne Hilfslinie

Abbildung 7.10: Interpretation der Richtung einer Kante

in Abbildung 7.10(a) die Richtung der Kante, die durch die Knoten 58 und 67 gegeben ist, noch als von 58 nach 67 gedeutet werden kann, ist eine Richtungsbestimmung der selben Kante in Abbildung 7.10(b) nicht mehr möglich, da die Hilfslinie nicht eingezeichnet wurde.

Kantenfarben Die in Abbildung 7.9 dargestellten Kantenfarben entsprechen der vom BeeJamA Algorithmus errechneten Kantenqualität (vgl. Kap. 2.5). Dabei ist ein Farbverlauf von Grün nach Rot möglich. Ein hoher Grün Anteil spricht für eine hohe Kantenqualität, ein hoher Rot Anteil dagegen für eine niedrige.

Nummerische Qualitätsdarstellung Neben der farblichen Qualitätsdarstellung ist es möglich, die Kantenqualitäten nummerisch darzustellen. Dazu werden Zahlen im Intervall von null bis eins an die Kanten gezeichnet. Null entspricht dabei der schlechtesten Qualität, welche in der farblichen Darstellung durch die Farbe Rot ausgedrückt wird; eins entspricht in der farblichen Darstellung der Farbe Grün, also der höchsten Kantenqualität. Weitere Informationen zum umschalten zwischen der farblichen und der nummerischen Qualitätsdarstellung gibt Kapitel 7.2.2.

Fahrspuren In der mikroskopischen Ansicht besteht die Möglichkeit, die einzelnen Fahrspuren der Straßen, bzw. Straßenseiten, einzublenden. Abbildung 7.11 zeigt einen Ausschnitt aus einer Straßen-

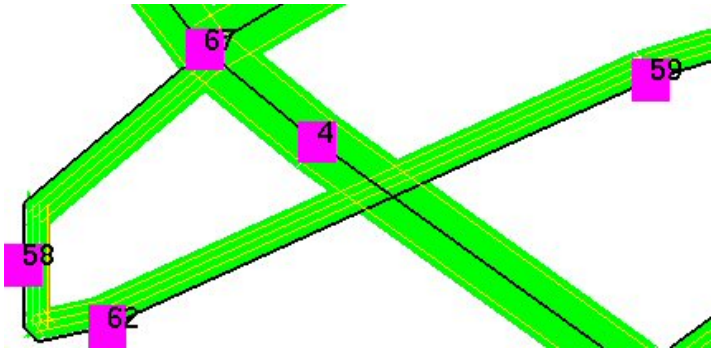


Abbildung 7.11: Straßenspuren

karte in dem die einzelnen Fahrspuren eingeblendet sind. Diese werden durch gelbe Linien auf den jeweiligen Kanten gekennzeichnet. So stellt die Straße zwischen den Knoten 59 und 62 zum Beispiel eine dreispurige Straße mit nur einer Fahrtrichtung, vom Knoten 59 zum Knoten 62, dar. Diese differenzierte „Spurendarstellung“ dient in erster Linie dem Testen des im Abschnitt 3.2 beschriebenen Verkehrsmodells, um das Spurwechsel- und Abbiegeverhalten der einzelnen Fahrzeuge überprüfen zu können.

Die Benutzerschnittstelle der Visualisierung

Die in Abbildung 7.12 dargestellte Benutzerschnittstelle der Visualisierungskomponente dient sowohl dem navigieren in der Straßenkarte, als auch dem ein- und ausblenden von darstellbaren Details bzw. Informationen. Die Navigationsfunktionen stellen im Wesentlichen eine Kamerasteuerung dar, die

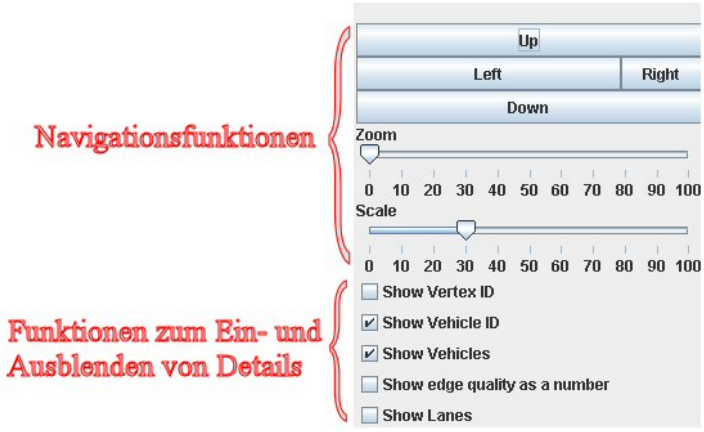


Abbildung 7.12: Benutzerschnittstelle der Visualisierung

es ermöglicht, den sichtbaren Bereich des Straßennetzes in dem Darstellungsfenster (Abb. 7.6, blauer Kasten) durch Kamerabewegungen zu verändern. So kann durch klicken auf „left“ bzw. „right“ die Kamera auf der X-Achse und durch klicken auf „up“ bzw. „down“ die Kamera auf der Y-Achse des kartesischen Koordinatensystem, in dem das Straßennetz gezeichnet wird, bewegt werden. Eine Veränderung des „Zoom“ Reglers bewirkt eine Veränderung der „Höhe“ der Kamera, um so den dargestellten Ausschnitt zu vergrößern, bzw. zu verkleinern. Die Zoomfunktion stellt also im wesentlichen eine Kamerabewegung auf der Y-Achse des kartesischen Koordinatensystems dar. Des Weiteren steuert die Zoomfunktion die in Kapitel 7.2.2 beschriebene makro- bzw. mikroskopische Straßenkartenansicht. Wird ein Zoomwert von 60 % überschritten/unterschritten, wechselt die Visualisierung automatisch in die mikroskopische/makroskopische Ansicht. Eine weitere möglichkeit der Veränderung des Zoomwertes bietet das Mausrad, wobei eine Vorwärtsbewegung des Mausrads der vergrößerung des Zoomwertes entspricht und eine Rückwärtsbewegung der Verkleinerung. Allerdings muss sich die Maus in dem Darstellungsfenster befinden, um diese Funktion nutzen zu können. Durch den „Scale“ Regler kann das gezeichnete Straßennetz skaliert werden, was einer Stauchung bzw. Streckung der Straßenkarte entspricht. Diese Funktion kann hilfreich sein, um zum Beispiel sehr große Straßennetze komplett in dem Visualisierungsfenster darzustellen.
Durch die Funktionen zum Ein- und Ausblenden von Details (vgl. Abbildung 7.12) lassen sich mehr oder weniger Informationen darstellen, um die Visualisierung des Straßennetzes an eigene Bedürfnisse anpassen zu können. Das Ein- bzw. Ausblenden der einzelnen Details wird über verschiedene Felder gesteuert, die entweder markiert oder nicht markiert sein können. Der Name des jeweiligen Details steht dabei rechts neben dem Feld. Wenn ein entsprechendes Feld markiert ist, hat das zur Folge, dass das entsprechende Detail eingeblendet wird. Entsprechend führt das entfernen der Markierung zum Ausblenden des jeweiligen Details. In Abbildung 7.12 ist zum Beispiel das Feld zum Anzeigen der Fahrzeug IDs markiert, was zur Folge hat, dass in dem Visualisierungsfenster die IDs der Fahrzeuge eingeblendet werden. Die folgende Liste stellt die einzelnen Funktionen zusammen:

- **Show Vertex ID:** Ein-/Ausblenden der Knotennummern.
- **Show Vehicle ID:** Ein-/Ausblenden der Fahrzeug IDs
- **Show Vehicles:** Ein-/Ausblenden der Fahrzeuge (die Fahrzeug IDs werden automatisch mit ausgeblendet).

Listing 7.1: Beispiel eines einfachen Operators

```
1 public class HelloWorldOperator extends Operator {
2
3         public void operate() {
4                 System.out.println("Hello World");
5         }
6 }
```

- **Show Edge quality as a number:** Durch das aktivieren des Feldes werden die Kantenqualitäten nicht mehr farbig dargestellt, sondern als Zahl. Diese Funktion wird erst ab einem Zoomwert von 60% oder grösser wirksam.
- **Show lanes:** Blendet die einzelnen Straßenspuren ein bzw. aus. Diese Funktion wird erst ab einem Zoomwert von 60% oder grösser wirksam.

7.3 Programmieren und Erweitern des Simulators

7.3.1 Beispiele zum Gebrauch von Operatoren

Im folgenden Abschnitt soll anhand von einigen einfachen Beispielen gezeigt werden, wie Operatoren selbst implementiert werden können und somit, wie aktiv in den Simulationsprozeß eingegriffen werden kann. Die Vorgehensweise ist dabei stets die selbe: Ableiten einer Klasse aus der Operatorhierarchie und Überschreiben der *operate()*-Methode.

Implementierung von Operatoren Zunächst sei die Implementierung eines einfachen Operators vorgestellt. Abbildung 7.1 zeigt den benötigten Quelltext um einen Hello-World-Operator zu entwickeln. Dazu wird einfach von der Basisklasse aller Operatoren abgeleitet und die *operate()*-Methode entsprechend überschrieben. Soll ein Operator für eine ObjectChain, z.B. der *StepVehicleChain*, geschrieben werden, muß von *ObjectOperator* bzw. in diesem Beispiel von *VehicleOperator* abgeleitet werden (s. Abb. 7.2). Die überschriebene und überladene *operate()*-Methode wird dann während der Simulation stattdessen aufgerufen. Will ein Operator nicht in jedem Schritt etwas tun, kann er sich entweder jeden Schritt aufrufen lassen, prüfen ob er zu diesem Zeitpunkt etwas tun möchte, oder aber, falls der nächste Zeitpunkt a priori bekannt ist, nur in dem entsprechenden Simulationsschritt aufrufen lassen, indem er von *ScheduledOperator* erbt. Abbildung 7.3 zeigt einen Operator, der im Konstruktor festlegt erst nach einer gewissen Zeitspanne in der Simulation aufgerufen zu werden und danach in einem festen Intervall erneut aufgerufen zu werden.

Zugriff auf die Welt Typischerweise müssen Operatoren während ihrer Ausführung auf Entitäten zugreifen oder sogar modifizieren. Da die Entitäten in der *World*-Klasse strukturiert gekapselt sind, ist der Zugriff leicht möglich. Das Beispiel in Abb. 7.4 zeigt einen Operator der alle Fahrzeuge auf allen Spuren des Straßennetzes ausgibt.

Listing 7.2: Beispiel eines VehicleOperators

```
class SimpleVehicleOperator extends VehicleOperator {

public void operate(Vehicle vehicle) {
        System.out.println(vehicle.getId());
}
}
```

Listing 7.3: Beispiel eines ScheduledOperators

```
class HelloWorldScheduledOperator extends ScheduledOperator {

public HelloWorldScheduledOperator() {
        // Let this operator be called the first time
        // after five seconds of simulated time
        schedule(Calendar.SECOND, 5);
}

public void operate() {
        System.out.println("Hello World");

        // Re-register the operator
        schedule(Calendar.SECOND, 2);
}
}
```

Listing 7.4: Beispiel eines weiteren Operators

```
class WorldOperator extends Operator {

        public void operate() {
                World world = DiscreteTrafficSimulator.getWorld();

                for(Edge edge : world.getEdges()) {

                        for(Lane lane : edge.getLanes() {

                                for(Vehicles vehicle : lane.getVehicles()) {
                                        System.out.println(vehicle.getId());
                                        System.out.println(vehicle.getHeadPosition());
                                }
                        }
                }
        }

}
```

Listing 7.5: Beispiel einer ChainSequenceFactory

```
1  class MyChainSequenceChainFactory {
2
3          public ChainSequence createChainSequence() {
4                  Operator myOperator = new WorldOperator();
5
6                  ChainSequence chainSequence = new ChainSequence();
7                  chainSequence.getStepFirstChain().add(myOperator);
8
9                  return chainSequence;
10         }
11
12 }
```

Einfügen von Operatoren in Chains Damit Operatoren während des Simulationsprozeßes ausgeführt werden, müssen sie in die jeweiligen Chains eingefügt werden. Da die Klasse ChainSequence die einzelnen Chains kapselt, kann über sie auf die Chains zugegriffen werden und somit die Operatoren hinzugefügt werden. Praktisch umsetzen läßt sich dies am besten, indem man die Möglichkeit nutzt, dem Konstruktor der *Configuration*-Klasse, welche die Referenz auf die *ChainSequence* hält, eine Instanz auf eine *ChainSequenceFactory* zu übergeben. In der dort überschriebenen *createChainSequence()*-Methode, wird dann eine ChainSequence aufgebaut, d.h. die entsprechenden Operatoren zu den Chains hinzugefügt, und diese dann zurückgegeben. Abb. 7.5 zeigt ein Beispiel.

Initialisierung von Entitäten Hier sei noch einmal das Beispiel aus Abbildung 4.5 aufgegriffen und schematisch gezeigt wie dieses Straßennetz manuell modelliert werden kann. Die Abbildung 7.6 zeigt den resultierenden Quelltext. Schon bei diesem kleinen Netz nimmt dessen Länge aber schnell überhand, weswegen bei größeren Netzen die Entitäten in der Welt praktisch nur von einer DHC automatisch erzeugt werden können.

Um Fahrzeuge für den Simulator zu erstellen benötigt man folgende Entitäten:

- Eine der Klasse Vehicle abgeleitete Entität
- Einen Driver
- Einen RoutingAlgorithmus, welcher das Interface RoutingAlgorithm implementiert
- Einen Startpunkt und ein Ziel der Route, welche jeweils von der Entität Edge sein muss.

Ein Democode welcher ein Fahrzeug in den Simulator einfügt ist im folgenden aufgeführt (7.7).

Ein typisches Chain-Sequence-Szenario Die Abbildung 7.13 gibt einen Überblick über den Aufbau einer ChainSequence die durch die Factory *PresentationChainSequenceFactory* erzeugt wurde. Es stellt ein typisches Szenario dar, welches von der PG für vielfältige Zwecke, wie der Name vermuten läßt, ursprünglich eine Präsentation, genutzt wurde. Alle wichtigen Operatoren die von der PG entwickelt wurden, sind inkludiert. Es folgt eine kurze Auflistung dieser Operatoren und eine Erläuterung der zugedachten Funktion. Zunächst die *PreChain*, welche die Operatoren zum Initialisieren der Simulation enthält:

Listing 7.6: Initialisieren der Entitäten für das Straßennetz aus Abbildung 4.5

```
1  public void initEdges(World world) {
2  // *** Vertices ***
3
4  Vertex v1 = new Vertex();
5  Vertex v2 = new Vertex();
6  Vertex v3 = new Vertex();
7  Vertex v4 = new Vertex();
8
9  // *** Edges ***
10 // Constructor parameters: (startVertex, endVertex, lengthInMeter, type)
11
12 Edge e1 = new Edge(v1, v2, 100, 0);
13 Edge e2 = new Edge(v2, v1, 100, 0);
14 Edge e3 = new Edge(v2, v3, 100, 0);
15 Edge e4 = new Edge(v3, v2, 100, 0);
16 Edge e5 = new Edge(v2, v4, 100, 0);
17
18 // *** Lanes ***
19
20 Lane l1 = new Lane(e1);
21 Lane l2 = new Lane(e1);
22 Lane l3 = new Lane(e2);
23 Lane l4 = new Lane(e3);
24 Lane l5 = new Lane(e3);
25 Lane l6 = new Lane(e4);
26 Lane l7 = new Lane(e5);
27
28 l1.setLeftNeighbor(l2);
29 l2.setRightNeighbor(l1);
30 l4.setLeftNeighbor(l5);
31 l5.setRightNeighbor(l4);
32
33 l1.addPostLane(l4);
34 l1.addPostLane(l5);
35 l1.addPostLane(l7);
36 l2.addPostLane(l4);
37 l2.addPostLane(l5);
38 l6.addPostLane(l3);
39
40 l7.addPreLane(l1);
41 l4.addPreLane(l1);
42 l5.addPreLane(l1);
43 l3.addPreLane(l6);
44
45 // Add edges to the world
46 world.addEdge(e1);
47 //...
48 world.addEdge(e5);
49 }
```

Listing 7.7: Beispiel einer Methode welche ein Fahrzeug in den Simulator einfügt

```
1  /**
2   * @param source
3   * @param destination
4   */
5  private void createCar(Edge source, Edge destination) {
6          Vehicle vehicle = new Car(3, 80);
7          vehicle.setNeedRouting(true);
8          Driver driver = new Driver(source, destination,
9                  new DijkstraOperator(100));
10         vehicle.setDriver(driver);
11         driver.getSource().addVehicle(vehicle);
12 }
```

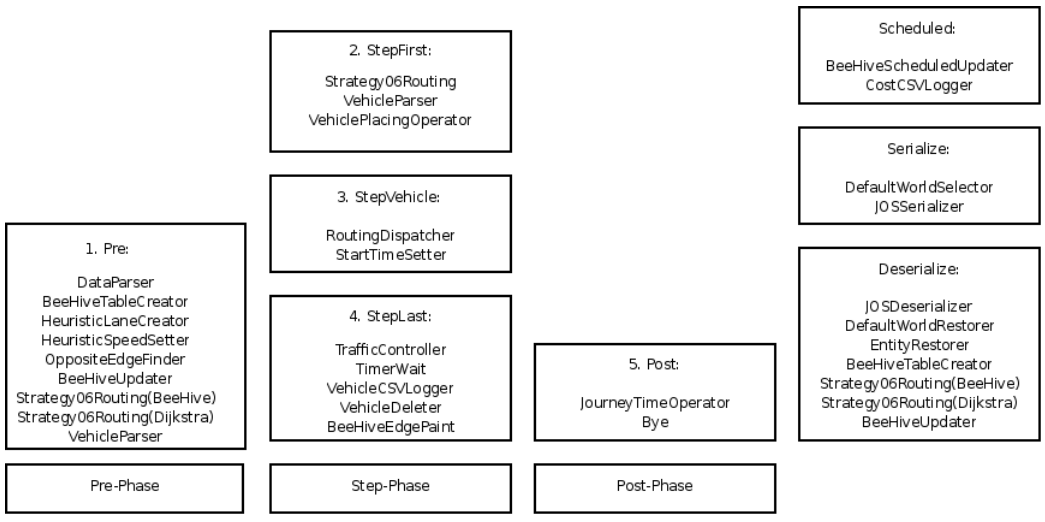


Abbildung 7.13: Operatorenanordnung durch die *PresentationChainSequenceFactory*.

DataParser: Parst Dateien im AND-Datenformat ein und generiert den Graphen

BeeHiveTableCreator: Erstellt alle notwendigen Tabellen für den BeeJamA Algorithmus

HeuristicLaneCreator: Legt heuristisch eine Anzahl von Spuren pro Kante fest

HeuristicSpeedSetter: Legt heuristisch die maximale Geschwindigkeit einer Kante fest

OppositeEdgeFinder: Bestimmt gegenüberliegende Kanten

BeeHiveUpdater: Erneuert alle BeeJamA-Tabellen

Strategy06Routing(BeeHiveRouter): Initialisiert eine bestimmte Routingstrategie für den BeeJamA-Algorithmus

Strategy06Routing(DijkstraOperator): Initialisiert eine bestimmte Routingstrategie für den Dijkstra-Algorithmus

VehicleParser: Parst aus einer Datei die gewünschte Konfiguration für die im Simulator auftretenden Fahrzeuge

Darauf folgen die drei Chains der *StepPhase* in der die eigentliche Simulation stattfindet. Die erste Chain ist die *StepFirstChain*:

Strategy06Routing: Eine beispielhaft implementierte BeeJamA-Routingstrategie

VehicleParser: Parst aus einer Datei die gewünschte Konfiguration für die im Simulator auftretenden Fahrzeuge

VehiclePlacingOperator: Setzt Fahrzeuge nach einem bestimmten Regelsatz auf die Straßen

Die zweite der drei Chains ist die *StepVehicleChain*:

RoutingDispatcher: Führt den Routing-Operator des *Driver*-Objekts des aktuell betrachteten *Vehicles* aus

StartTimeSetter: Neuen zur Simulation hinzugefügten Fahrzeugen wird hier die Abfahrtszeit gesetzt

Die *StepPhase* wird abgeschlossen durch die *StepLastChain*:

TrafficController: Implementation der TMC

TimerWait: Wird benötigt um den Simulator für den Benutzer in annehmbarer Geschwindigkeit ablaufen zu lassen

VehicleCSVLogger: Speichert alle fahrzeugspezifischen Daten in eine tabellenartig aufgebaute Logdatei

VehicleDeleter: Löscht am Ziel angekommene Fahrzeuge aus dem Straßennetz

BeeHiveEdgePaint(): Zeichnet die Güten mittels BeeJamA-spezifizierten Werten ein

Die Simulation insgesamt wird durch die *PostChain* beendet:

JourneyTimeOperator: Gibt die durchschnittliche benötigte Reisezeit aus

Bye: Gibt eine Meldung über die Beendigung der Simulation aus

Listing 7.8: Starten einer Simulation

```
1  public void startSimulation
2          (ChainSequenceFactory chainSequenceFactory,
3          long maxSteps, int secondsPerStep, float cellSize) {
4
5          Configuration conf = new Configuration(chainSequenceFactory);
6          conf.setMaxSteps(maxSteps);
7          conf.setSecondsPerStep(secondsPerStep);
8          conf.setCellSize(cellSize);
9
10         World world  = new World();
11         initEdges(world);
12         initVehicles(world);
13
14         DiscreteTrafficSimulator dts = new DiscreteTrafficSimulator(conf, world);
15         dts.start();
16 }
```

Die *ScheduledChain*:

BeeHiveScheduledUpdater: Ein Updater für den BeeJamA-Algorithmus welcher in beliebigen Zeitintervallen ausgeführt werden kann

CostCSVLogger: Speichert alle Kosten der Kanten in einer tabellenartig aufgebauten Logdatei

DefaultWorldSelector: Selektiert die zu serialisierenden Daten, hier: die aktuelle *World* Instanz

JOSSerializer: Schreibt die zu serialisierenden Daten mittels JOS in eine Datei

JOSDeserializer: Liest den gespeicherten Simulationsstand aus einer Datei aus

DefaultWorldRestorer: Interpretiert die deserialisierten Daten als *World* Instanz

EntityRestorer: Restauriert die Objektreferenzen

BeeHiveTableCreator: Am Ende der Deserialize-Chain müssen dann wieder einige Operatoren aus der *PreChain* eingefügt werden, da diese selber nicht ausgeführt wird (s.o.), um die Simulationsumgebung zu initialisieren.

Strategy06Routing(BeeHiveRouter): s.o.

Strategy06Routing(DijkstraOperator): s.o.

BeeHiveUpdater: s.o.

Starten der Simulation Wurde eine *Configuration*- und eine *World*-Instanz erzeugt, kann der Simulator gestartet werden. Abb. 7.10 zeigt dies in einem zusammenhängenden Kontext.

7.3.2 Implementation einer Routingstrategie

Um eine eigends entwickelte Routingstrategie für den Simulator zu erstellen muss nur eine eigene Klasse die abstrakte Klasse *CommonCostCalculation* erweitert welches zwei zu überschreibende Methoden definiert:

calculateEdgeCost(Edge) Zu einer gegebenen Kante berechnet diese Methode die aktuellen Kosten. Diese können durch eine beliebige Funktion errechnet und als Rückgabewert definiert werden. Der Entwickler muss dabei allerdings beachten, dass diese Methode in jedem Durchlauf des Simulators für jede Kante des Systems aufgerufen wird. Daher sollte diese Methode sehr effizient gestaltet werden.

getAlgorithm() Diese Methode gibt eine Zeichenkette zurück welche das für die obere Funktion genutzte Routing kennzeichnet. Zurückgegeben wird der Name des Routingalgorithmus für welcher die Strategie erstellt wurde.
Falls eine Strategie für mehrere Algorithmen genutzt werden soll müssen mehrere Objekte dieser Strategie erzeugt werden.

Die Strategieen befinden sich alle im Package *operators.routingOperators.strategies* oder deren Unterpakete.

Spezialfall der BeeJamA-Routingstrategien

Die Strategieen für den BeeJamA-Algorithmus unterscheiden sich leicht von anderen Routingstrategieen. Die abstrakte Klasse *CommonRouting* erweitert die Klasse *CommonCostCalculation* und fügt zwei abstrakte Methoden hinzu:

computeCost(float[, float[])] Diese Methode gibt dem Entwickler die Möglichkeit die für das Routing benötigten Kosten von der länge des Weges abhängig zu berechnen. Die Kostenwerte in den Tabellen sollten unabhängig von der Länge der Kante sein und werden erst im letzten Schritt des Routings hier zusammengeführt.

cumulateCosts(float, float) Je nach gewählter Funktion dürfen unter Umständen die errechneten Kosten nicht einfach miteinander addiert werden. Deswegen kann ein Entwickler in dieser Methode definieren, wie die beiden Kosten beim Tabellenupdate verrechnet werden.

normalize(float[ )] In der normalize()-Methode kann die genutzte Normalisierungsfunktion überschrieben werden.
Hier ist es auch wichtig anzumerken, dass die in den Tabellen stehenden Kosten in Qualitäten umgerechnet werden müssen da ein kleinerer Wert in den Kosten eine höhere Wahrscheinlichkeit haben muss.

Wie in der Vorlage 7.3.2 zu sehen benötigt die Klasse *CommonRouting* eine Instanz der neu erstellten Klasse um an diese die Anfragen zu delegieren. Am Einfachsten ist dieses im Construktor der neu erstellten Klasse zu implementieren. Die statische Methode *getRouting()* gibt dann diese Instanz zurück.

Vorlagen

In diesem Abschnitt werden zwei Vorlagen für die im Simulator genutzten Routingalgorithmen vorgestellt. Hier sind nur die Methoden implementiert welche absolut notwendig sind zum Betrieb des Simulators und müssen je nach Vorgabe angepasst werden.

Listing 7.9: Vorlage für eine Strategie des Dijkstra-Algorithmus

```
1  /* STOP - A jam avoidance framework
2   *
3   * Copyright (C) 2006-2007 PG STOP, pgstop@cs.uni-dortmund.de
4   *
5   * This program is free software; you can redistribute it and/or modify
6   * it under the terms of the GNU General Public License as published by
7   * the Free Software Foundation; either version 2 of the License, or
8   * (at your option) any later version.
9   *
10  * This program is distributed in the hope that it will be useful,
11  * but WITHOUT ANY WARRANTY; without even the implied warranty of
12  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
13  * GNU General Public License for more details.
14  *
15  * You should have received a copy of the GNU General Public License
16  * along with this program; if not, write to the Free Software
17  * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
18  */
19
20 package edu.udo.cs.pg502.simulator.operators.routingOperators.strategies;
21
22 import edu.udo.cs.pg502.simulator.core.Edge;
23 import edu.udo.cs.pg502.simulator.routingAlgorithm.CommonCostCalculation;
24
25 /**
26  *
27  */
28 public class CostCalculator extends CommonCostCalculation {
29
30         /* (non-Javadoc)
31          * @see edu.udo.cs.pg502.simulator.routingAlgorithm.CommonCostCalculation
32          *      #calculateEdgeCost(edu.udo.cs.pg502.simulator.core.Edge)
33          */
34         public float calculateEdgeCost(Edge edge) {
35 //              TODO
36                 return 0;
37         }
38
39         /* (non-Javadoc)
40          * @see edu.udo.cs.pg502.simulator.routingAlgorithm.CommonCostCalculation
41          *      #getAlgorithm()
42          */
43         public String getAlgorithm() {
44 //              TODO
45                 return null;
```

```
46          }
47  }
```

Listing 7.10: -Vorlage für eine Strategie des BeeJamA-Algorithmus

```
1  /* STOP - A jam avoidance framework
2   *
3   * Copyright (C) 2006-2007 PG STOP, pgstop@cs.uni-dortmund.de
4   *
5   * This program is free software; you can redistribute it and/or modify
6   * it under the terms of the GNU General Public License as published by
7   * the Free Software Foundation; either version 2 of the License, or
8   * (at your option) any later version.
9   *
10  * This program is distributed in the hope that it will be useful,
11  * but WITHOUT ANY WARRANTY; without even the implied warranty of
12  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
13  * GNU General Public License for more details.
14  *
15  * You should have received a copy of the GNU General Public License
16  * along with this program; if not, write to the Free Software
17  * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
18  */
19
20 package edu.udo.cs.pg502.simulator.operators.routingOperators.strategies.beeHive;
21
22 import edu.udo.cs.pg502.simulator.core.Edge;
23 import edu.udo.cs.pg502.simulator.operators.routingOperators.beehive.CommonRouting;
24
25 /**
26  *
27  */
28 public class RoutingStrategy extends CommonRouting {
29
30          /**
31           * Constructor
32           */
33          public RoutingStrategy() {
34                  CommonRouting.routing = this;
35          }
36
37          /* (Kein Javadoc)
38           * @see edu.udo.cs.pg502.simulator.routingAlgorithm.beehive.CommonRouting
39           *      #cumputeQuality(float, float)
40           */
41          @Override
42          public float[] computeCost(float[] quality, float[] length) {
43  //              TODO
44                  return null;
45          }
46
47          /* (non-Javadoc)
48           * @see edu.udo.cs.pg502.simulator.routingAlgorithm.CommonCostCalculation
```

```
49          *       #getCost(edu.udo.cs.pg502.simulator.core.Edge)
50          */
51         public float calculateEdgeCost(Edge edge) {
52 //              TODO
53                 return 0;
54         }
55
56         /* (non-Javadoc)
57          * @see edu.udo.cs.pg502.simulator.routingAlgorithm.beehive.CommonRouting
58          *       #normalize(float[])
59          */
60         @Override
61         public float[] normalize(float[] data) {
62 //              TODO if necessary
63                 return super.normalize(data);
64         }
65
66         /* (non-Javadoc)
67          * @see edu.udo.cs.pg502.simulator.routingAlgorithm.beehive.CommonRouting
68          *       #cumulateCosts(float, float)
69          */
70         @Override
71         public float cumulateCosts(float quality1, float quality2) {
72 //              TODO
73                 return 0;
74         }
75
76         /* (non-Javadoc)
77          * @see edu.udo.cs.pg502.simulator.routingAlgorithm.CommonCostCalculation
78          *       #getAlgorithm()
79          */
80         public String getAlgorithm() {
81 //              TODO
82                 return null;
83         }
84 }
```

7.4 Lokale Parameterbeeinflussung

Zur lokalen Beinflussung des Verkehrsflusses ist es möglich, lokal begrenzt einige Straßeneigenschaften zu verändern. Zu diesem Zweck wurde eine einfache und leicht zu erweiternde GUI geschrieben, die es ermöglicht bestimmte Edges des aktuellen Simulationslaufes auszuwählen und deren Eigenschaften zu verändern. Diese Klasse nennt sich *EdgeFinder* und ist im Paket *Operators* zu finden. Die Entsprechende grafische Oberfläche namens *EdgeFinderVisu* ist im Paket *trafficmodel* abgelegt. Um den *EdgeFinder* zu verwenden, muss dieser in der PreOperatorChain eingefügt werden. Es ist darauf zu achten, dass bereits alle Daten geparst wurden, da sonst keine Objekte zur Verfügung stehen, deren Parameter beeinflusst werden können. Um dieses Problem zu umgehen, muss der *EdgeFinder* als letztes Element in die *PreOperatorChain* eingefügt werden.

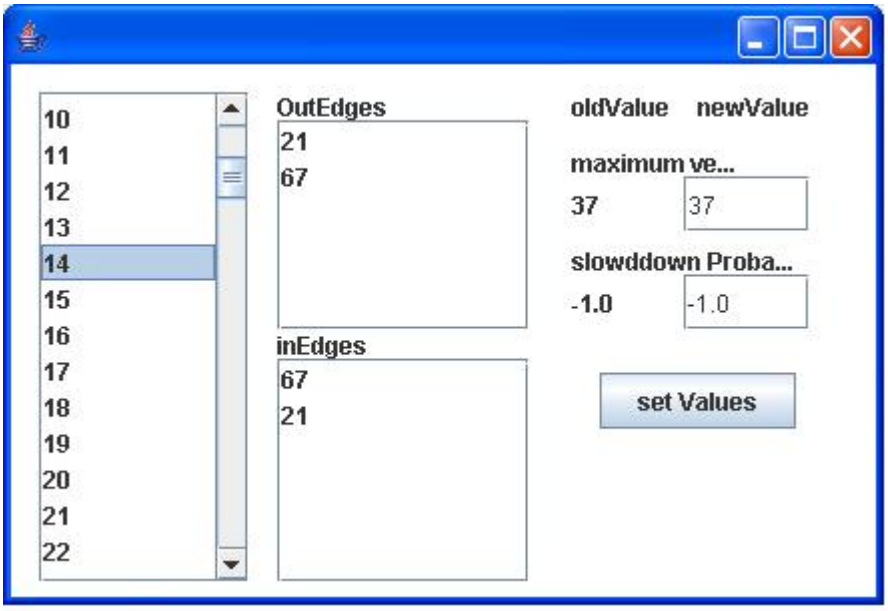


Abbildung 7.14: EdgeFinder

In Abbildung 7.14 ist die Visualisierung zu sehen. Im linken Bereich befindet sich eine aufsteigend sortierte Liste aller im Simulator gefundenen Knoten. Wird einer dieser Knoten per Mausklick ausgewählt, erscheinen in den Boxen rechts daneben ebenfalls Knotennummern. In der Box OutEdges befinden sich die Endknotennummern der Knoten, die wie in Abbildung 7.14 zu sehen, die Edges von Knoten 14 zu Knoten 21 und 67 begrenzen. In der Box InEdges sind dementsprechend StartKnotennummern für die Edges von Knoten 67 und 21 zu Knoten 14 zu sehen. Wird eine Knotennummer in diesen Boxen ausgewählt, erscheinen die Werte der Attribute Maximalgeschwindigkeit und Trödelwahrscheinlichkeit in den Textboxen. Diese können durch Eingabe neuer Werte in den Textboxen verändert und durch Klicken des *SetValue* Buttons in den aktuellen Simulationslauf übertragen werden. Ein Wert -1 in den Textfenstern bedeutet, dass die die Defaultwerte des Simulators verwendet werden. Sollte Bedarf bestehen, weitere Kantenattribute zu verändern, kann dies leicht durch Erweiterung dieser Visualisierung geschehen. Die Attribute der Edges können leicht über die bereits vorhanden Get und Set Methoden ausgelesen und modifiziert werden.

7.5 Dijkstra

Um den BeeJamA Algorithmus mit anderen Routingalgorithmen vergleichen zu können ist es möglich, weitere Algorithmen in den Simulator zu integrieren. Für erste Vergleiche wurde der Dijkstra Algorithmus [3] implementiert, welcher im Folgenden kurz erläutert wird. Der Dijkstra-Algorithmus dient der Berechnung eines besten Pfades zwischen zwei Knoten auf einem kantengewichteten Graphen. Die Kantengewichte werden entweder im einfachen Fall aus den Entfernungen ermittelt oder auf Basis einer anderen Kantenbewertungsfunktion. Als Grundlage für die Kostenermittlung der jeweiligen Kanten, und damit auch Wege, wurde in der vorliegenden Implementierung die in Abschnitt 2 beschriebene Bewertungsfunktion benutzt. Das verwendete Qualitätskriterium ist also nicht der kürzeste Weg, sondern die Strecke, die zum Zeitpunkt der Routenberechnung einen Weg darstellt,

der schenllstmöglich zum Ziel führt. Der Dijkstra-Algorithmus kann in der vorliegeneden Implementierung in einem Preprocessing entweder einmal die benötigten Routen berechnen oder diese nach einem festen Zeitintervall aktualisieren.

Kapitel 8

Literaturverzeichnis

[1] Lars Bensmann, Thomas Büning, Mike Duhm, René Jeruschkat, Gero Kathagen, Johannes Meth, Kai Moritz, Christian Müller, Thorsten Pannenbäcker, Björn Vogel, and Rene Zeglin. *Final Report of Project-Group 439: BeeHive - The Energy Efficient Scheduling and Routing Framework*. September 2004.

[2] Statistisches Bundesamt. *Datenreport 2006 - Zahlen und Fakten Bundesrepublik Deutschland*. 2007.

[3] Clifford Stein Charles E. Leiserson, Ronald L. Rivest. *Introduction to Algorithms*, volume 2. B and T, 2001.

[4] `http://beejama.sourceforge.net`.

[5] `http://www.eclipse.org`.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, January 1995.

[7] `http://www.gnu.de`.

[8] `http://java.sun.com/docs/codeconv`.

[9] `http://www.java.com/de`.

[10] `http://jgrapht.sourceforge.net/`.

[11] `https://jogl.dev.java.net`.

[12] Jörg Kienzle. *Analyse von Einzelfahrzeugdaten*. 2001.

[13] John T. Moy. *OSPF Anatomy of an Internet Routing Protocol*. Addison-Wesley, 1998.

[14] John T. Moy. *OSPF complete implementation*. Addison-Wesley, 2000.

[15] Wu Ning. *Verkehr auf Schnellstraßen im Fundamentaldiagramm - Ein neues Modell und seine Anwendungen*. 2000.

[16] T.D. Seeley. *The Wisdom of Hive*. 1995.

[17] `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4152790`.

[18] Peter Tondl. *Dezentrale Verteilung von Verkehrsinformationen durch Fahrzeug-Fahrzeug-Kommunikation*. Shaker Verlag Aachen, 2006.

[19] Horst F. Wedde and Mudassar Farooq. *BeeHive: An Efficient, Scalable, Adaptive, Fault-tolerant and Dynamic Routing Algorithm Inspired from the Wisdom of the Hive*. Number 801. October 2005.

[20] Horst F. Wedde and Muddassar Farooq. *A comprehensive review of nature inspired routing algorithms for fixed telecommunication networks*, volume 52. Elsevier North-Holland, Inc., New York, NY, USA, 2006.