Universität Stuttgart
Fakultät Informatik

Prüfer:	Prof. Rothermel
Betreuer:	Dipl.-Inf. Uwe Kubach
begonnen am	19.06.2000
beendet am	18.12.2000
CR-Klassifikation:	I.6.7, C.2.1

Studienarbeit Nr. 1790

Entwicklung einer Simulationsumgebung für kontextbezogenen Informationszugriff in einer mobilen Umgebung

Mario Hegele

Institut für parallele und verteilte
Höchstleistungsrechnersysteme
Breitwiesenstraße 20-22
D-70565 Stuttgart

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **3**

2.1 Submodelle und Infrastrukturgraph 3

2.2 Ablauf der Simulation 4

2.3 Prototyp 5

3 Benutzermodelle **7**

3.1 Der Infrastrukturgraph 8

3.2 Mobilitätsmodell 9

3.3 Informationsmodell 10

3.4 Zugriffsmodell 10

3.5 Abhängigkeiten 11

4 Der Simulationslauf-Algorithmus **13**

4.1 Der Simulationslauf 13

4.2 Bestimmung der zu besuchenden Knoten 15

4.3 Wegesuche 16

4.3.1 Enumeration (OPT) 17

4.3.2 Nearest Neighbour (NN) 18

4.3.3 Minimum Spanning Tree (MST) 18

4.3.4 Weitere Heuristiken 20

4.3.5 Vergleich der programmierten Algorithmen 20

4.4 Ermittlung der abgefragten Informationseinheiten 23

4.5 Ergebnisse eines Simulationslaufes 24

5 Einsatzbeispiel **27**

5.1 Mobilitätsmodell 27

5.2 Informationsmodell 28

5.3 Zugriffsmodell 28

5.4 Anwendung des Einsatzbeispiels 30

6 Die Simulationsumgebung **33**

6.1 Start der Applikation 33

6.2 Hauptfenster 34

6.3 Menüleiste 34

6.3.1 Menü *File* 36

6.3.2 Menü *Graph* 37

6.3.3 Menü *Results* 38

6.3.4 Menü *Help* 39

6.4 Submodell-Fenster 39

6.5 Infrastrukturgraph-Fenster 40

6.6 Simulationslauf-Fenster 41

6.7 Systeminformation-Fenster 42

6.8 Durchführung der Simulation 43

6.8.1 Vor der Simulation 43

6.8.2 Während der Simulation 44

6.8.3 Nach der Simulation 46

6.9 Optionen-Fenster 46

6.10 Infrastrukturgraph und Besuchswahrscheinlichkeitskarte 49

6.10.1 Einfärbung der Besuchswahrscheinlichkeitskarte 51

6.11 Textuelle Darstellung des Infrastrukturgraphen 52

6.12 Textuelle Darstellung der Simulationsergebnisse 53

6.12.1 Zähler für die Kategorie *Knoten* 53

6.12.2 Zähler für die Kategorie *Kante* 54

6.12.3 Zähler für die Kategorie *Informationseinheit* 54

6.12.4 Zähler für die Kategorie *Besuchsfeld* 55
6.13 Dateiformate 56
6.13.1 Konfigurations-Datei 57
6.13.2 Optionen-Datei 58
6.13.3 Infrastrukturgraph-Datei 59
6.13.4 Log-Datei 62
6.13.5 Resultats-Datei 64

7 Implementierung 67
7.1 Einbindung eigener Submodelle 67
7.1.1 Erstellen eines Submodells 68
7.1.2 Erstellen eines Submodell-Managers 71
7.1.3 Einbinden eines Submodell-Managers 75
7.2 Überblick über die erstellte Software 76
7.2.1 Umfang der Software 76
7.2.2 Verwendete Tools 76

8 Zusammenfassung 79

Literaturverzeichnis 81

A Glossar 83

B Klassenübersicht 87

Abbildungsverzeichnis

2.1 Graph-Repräsentation von Stuttgarts Innenstadt 4

3.1 Graph-Repräsentation von Tadmor 9
3.2 Gleich- und "Zipf"-Verteilung 11

4.1 Pseudocode Simulationslauf 14
4.2 Pseudocode für die Bestimmung der Knoten 16
4.3 Pseudocode *Nearest Neighbour*-Heuristik 19
4.4 Schwäche der *Nearest Neighbour*-Heuristik 20
4.5 Zufälliger vollständiger Graph mit 25 Knoten 21
4.6 Vergleich der Weglängen (für geschlossene Wege) 22
4.7 Vergleich der Weglängen (für nicht geschlossene Wege) 23
4.8 Vergleich der Laufzeiten 24
4.9 Subgraph mit kürzesten Pfaden zwischen den Zielknoten 25
4.10 Nach Abstrahierung über die Durchreiseknoten 25
4.11 Kürzester Weg über die Zielknoten 25
4.12 Nach Rückführung auf Originalgraph 26

5.1 Konfiguration des Mobilitätsmodells 28
5.2 Konfiguration des Informationsmodells 29
5.3 Konfiguration des Zugriffsmodells 29
5.4 Liniennetz 31
5.5 Student 31
5.6 Ausflügler 31
5.7 Gleichverteilung 32

5.8 "Zipf"-Verteilung 32

6.1 Hilfetext bei Kommandozeilen-Parameter "-h(elp)" 34
6.2 Hauptfenster 35
6.3 Die Menüleiste 35
6.4 Menü *File* 36
6.5 Laden eines Infrastrukturgraphen 37
6.6 Warnung: Verlust der Simulationsergebnisse 38
6.7 Menü *Graph* 38
6.8 Menü *Results* 39
6.9 "About"-Fenster 39
6.10 Submodell-Fenster 40
6.11 Infrastrukturgraph-Fenster 41
6.12 Simulationslauf-Fenster 42
6.13 Systeminformation-Fenster 43
6.14 Warnung: Verlust der Simulationsergebnisse 43
6.15 Laufende Simulation 45
6.16 Gestoppte Simulation 45
6.17 Mobilitatsmodell/Zielknoten-Problem 45
6.18 Optionen-Fenster (Infrastrukturgraph) 48
6.19 Optionen-Fenster (Besuchswahrscheinlichkeitskarte) 48
6.20 Darstellung von Infrastrukturgraph und Besuchswahrscheinlichkeitskarte 50
6.21 Von der Kantenbenutzung zum Feldbesuch 51
6.22 Einfärbung von Feldern 52
6.23 Textuelle Darstellung der Infrastrukturgraph-Knoten 52
6.24 Textuelle Darstellung der Infrastrukturgraph-Kanten 53
6.25 Zähler für die Knoten 54
6.26 Zähler für die Kanten 55
6.27 Zähler für die Informationseinheiten 55
6.28 Zähler für die Besuchsfelder 56
6.29 Beispiel für eine Konfigurations-Datei 58

6.30 Beispiel für eine Optionen-Datei 60
6.31 Format einer Infrastrukturgraph-Datei 62
6.32 Beispiel für eine Infrastrukturgraph-Datei 63
6.33 Beispiel für eine Log-Datei 64

B.1 Package-Übersicht 87

Kapitel 1

Einleitung

In den letzten Jahren konnte eine enormer Zuwachs an Forschungsbemühungen und -ergebnissen auf dem Gebiet des "Mobile Computing" verzeichnet werden. Mit der Ausbreitung der drahtlosen Kommunikations-Infrastruktur gingen auch neue Klassen von Applikationen einher, z.B. für mobile Büros oder Navigationshilfen.

Im Rahmen des Nexus-Projekts wird am Institut für Parallele und Verteilte Höchstleistungsrechner (IPVR) an der Universität Stuttgart eine Plattform für mobile Anwendungen mit Kontextbezug entwickelt. Kontextbezug bedeutet, dass den Anwendungen Wissen über den Kontext des Benutzers, z.B. seine geographische Position, zur Verfügung steht. Der Kontextbezug kann in einer Vielzahl von Anwendungen ausgenutzt werden, um die angebotene Funktionalität an den jeweiligen Kontext des Benutzers anzupassen. Ein Beispiel ist das automatische Bereitstellen von Informationen über die aktuelle Umgebung eines Benutzers in einem elektronischen Touristen- oder Einkaufsführer (z.B. [CDMF00]).

Erste Erfahrungen bei der Entwicklung der Nexus-Plattform haben gezeigt, dass das Benutzerverhalten, insbesondere die Benutzermobilität und das Verhalten beim Zugriff auf Informationen, einen entscheidenden Einfluss auf die Effizienz der entwickelten Algorithmen haben. Da zum gegenwärtigen Zeitpunkt noch keine Applikation der Plattform vorhanden ist, die sich für einen Feldversuch eignet, muss zur Evaluierung dieses Zusammenhangs auf Simulationen zurückgegriffen werden.

Die Aufgabe dieser Studienarbeit ist es, eine Simulationsumgebung für den kontextbezogenen Informationszugriff in einer mobilen Umgebung zu entwickeln. Diese Simulationsumgebung basiert auf einem neuen Modell, welches das Verhalten von Benutzern in einer mobilen Umgebung nachstellt. Im Gegensatz zu den meisten bestehenden Benutzermodellen, die sich mit Aspekten entweder nur der Mobilität oder nur des Zugriffs auf Informationen beschäftigen, behandelt das in dieser Arbeit verwendete Modell beide dieser Gesichtspunkte.

Die Simulationsumgebung soll über eine komfortable, graphische Oberfläche verfügen, die u.a. die Konfiguration der Simulationen und die Präsentation von Ergebnissen erlaubt. Weitere Anforderungen sind: ein flexibler und modularer Aufbau, eine einfache Installierbarkeit, eine umfangreiche Dokumentation des Quellcodes und das Einrichten eines Einsatzbeispiels.

Das nächste Kapitel liefert die wichtigsten Grundlagen für diese Ausarbeitung. Es folgt das dritte Kapitel, in dem detailliert die zugrundeliegenden Modelle und Alternativen dazu beschrieben werden. Daran schließt sich Kapitel vier an, das den Ablauf der Simulation beschreibt. Ein Einsatzbeispiel für die Simulationsumgebung wird im fünften Kapitel vorgestellt. Das sechste und siebte Kapitel stellen die Applikation an sich und das Vorgehen bei deren Implementierung vor. Den Abschluss bildet eine Zusammenfassung.

Im Anhang findet sich das Literaturverzeichnis, das Glossar, welches fachspezifische und fremdsprachliche Begriffe erklärt, und eine Übersicht über sämtliche Klassen und Pakete der Applikation.

Kapitel 2

Grundlagen

Ein Leitfaden für diese Studienarbeit bilden [KR00] und ein dazugehöriger Programm-Prototyp. Die in [KR00] beschriebene Modellierung und die Architektur des Simulations-Frameworks, die in diesem Kapitel kurz erläutert werden, wurden in dieser Studienarbeit teilweise übernommen.

2.1 Submodelle und Infrastrukturgraph

In [KR00] wird ein Benutzermodell vorgestellt, das im Gegensatz zu den meisten bisherigen Modellen sowohl die Mobilität des Benutzers als auch dessen Zugriff auf Informationen wiedergibt. Dieses Benutzermodell, das in dieser Studienarbeit in leicht abgeänderter Form (s. Kapitel 3) übernommen wurde, besteht aus drei sogenannten "Submodellen", die die Bewegungen und die Informationsabrufe des Benutzers regeln.

Für das Simulations-Framework werden einige verschiedene Instanzen pro Submodell angeboten, die flexibel ausgetauscht werden können. Offene Schnittstellen garantieren die Erweiterbarkeit um neue Submodell-Instanzen. Die Bewegungen (also die Mobilität) des Benutzers eines mobilen Informationssystems finden auf einem *Infrastrukturgraphen* statt; das ist ein Graph, wie man ihn z.B. aus der Theoretischen Informatik kennt, und dessen Knoten Orte und dessen Kanten Verbindungen (z.B. Straßen oder Schienen) zwischen diesen Orten symbolisieren. Weitere Informationen zum *Infrastrukturgraphen* können im Abschnitt 3.1 gefunden werden.

2.2 Ablauf der Simulation

Wie geht nun die Simulation bei [KR00] vonstatten? Vor der eigentlichen Simulation, muss der Simulator konfiguriert werden: dazu gehören zumindest die Auswahl einer Instanz für jedes der drei Submodelle, die Entscheidung für einen Infrastrukturgraphen und die Festlegung der Anzahl der zu simulierenden Umläufe (s. Kapitel 4 und 6).

Eine Simulationsreihe besteht aus einer Folge von Simulationsläufen, die sequentiell und unabhängig voneinander durchgeführt werden. Bei einem Simulationslauf wird das Verhalten eines Benutzers eines mobilen Informationssystems simuliert: er besucht Orte, die der Infrastrukturgraph anbietet, und ruft dort Informationseinheiten ab.

Die zu besuchenden Orte bestimmt das *mobility model*, die abgerufenen Informationseinheiten werden vom *request frequency model* und *request distribution model* festgelegt. Zur späteren Analyse protokolliert die Simulation das Verhalten des Benutzers in bezug auf die Besuche und die Informationsabfrage in einer Datei.

Als abschließendes Beispiel wird bei [KR00] die Simulation auf einen Infrastrukturgraphen angewandt, der die Stuttgarter Innenstadt (s. Abbildung 2.1) repräsentiert: zwei Benutzertypen (Tourist, Einkäufer) werden simuliert, die unterschiedliche bevorzugte Orte besuchen. Je nachdem, ob die Benutzer einen Ort bevorzugterweise besuchen oder gezwungenermaßen, weil er "auf dem Weg liegt", werden mehr oder weniger Informationen angeboten und abgefragt.

Abbildung 2.1: Graph-Repräsentation von Stuttgarts Innenstadt

In den folgenden Kapiteln werden ausführlich das Benutzer-, bzw. die Submodelle (Kapitel 3) und der Algorithmus (Kapitel 4), nach dem eine Simulation abläuft, vorgestellt. Es sei nochmals erwähnt, dass sich Submodelle dieser Studienarbeit leicht von denen von [KR00] unterschieden.

2.3 Prototyp

Die in [KR00] vorgestellte Architektur wurde als Prototyp implementiert, auf dem das obige Einsatzbeispiel ausgeführt wurde. Der Prototyp wurde in der Programmiersprache *Java (JDK 1.2.2)* erstellt und umfasst 16 Klassen und ca. 40 kB Quellcode (zum Umfang der in dieser Studienarbeit erstellten Simulationsumgebung, s. Kapitel 7).

In dieser Studienarbeit soll eine Simulationsumgebung entwickelt werden, die u.a. folgende Schwächen des Prototypen verbessert:

- Die Konfiguration der Simulationsumgebung bezüglich des Infrastrukturgraphen oder der Submodelle erfordert aufwendige Änderungen des Quellcodes und eine Neukompilierung.
- Das Dateiformat zur Repräsentation des Infrastrukturgraphen erweist sich als schwierig zu lesen.
- Es fehlen Möglichkeiten, Ergebnisse graphisch wiederzugeben, auf die Darstellung des Infrastrukturgraphen Einfluss zu nehmen oder eine Simulationsreihe abzubrechen, falls ihre Laufzeit durch unzureichende Einstellungen unakzeptabel hoch wird.

Kapitel 3

Benutzermodelle

Bei der Modellierung des Informationszugriffs von mobilen Benutzer, müssen zwei unterschiedliche Gesichtspunkte berücksichtigt werden: die Mobilität der Benutzer und die Art und Weise, wie sie die Informationen abfragen.

Eine verbreitete Methode, die Mobilität zu modellieren, ist es, das Verhalten des Benutzers vom Zufall bestimmen zu lassen. Die Geschwindigkeit und die Richtung, mit der sich hier ein Benutzer fortbewegt, wird von Wahrscheinlichkeitsverteilungen festgelegt (*random walk*). Als Bewegungsgrundlage können z.B. Polygone dienen (Rechtecke, regelmäßige Sechsecke), die lückenlos aneinander grenzen und ein gewisses Gebiet abdecken. Die Bewegung findet von Polygon zu Polygon statt (s. [DGM98]).

Ein etwas differenzierteres Verfahren (s. [LBC98]) ist, die Bewegungen des Benutzers von einer Kombination aus *zufälliger* Bewegung und Bewegungs*mustern* abhängig zu machen. Die Bewegungen finden auf zwei Ebenen statt: auf der höheren Ebene erfolgen sie - von Bewegungsmustern (*user mobility patterns*) gesteuert - zwischen Zellen (*global mobility model*). Die Bewegungsmuster spiegeln die alltäglichen Bewegungen der Benutzer wider (Weg zur Arbeit, zum Einkaufen), die erwartungsgemäß einer gewissen *Regelmäßigkeit* unterworfen sind. Auf einer tieferen Ebene (innerhalb der Zellen, *local mobility model*) wird der Benutzer vom Zufall gesteuert. Hier werden Ereignisse berücksichtigt, die vom Zufall abhängen und die Richtung und Geschwindigkeit des Benutzers beeinflussen können, wie z.B. Ampeln, Staus, Wetterbedingungen, ...

Mobilität kann auch modelliert werden, indem man die Bewegungsbedürfnisse von Gruppen oder Individuen von Benutzern beschreibt. Dies kann zum einen dadurch geschehen, dass Informationen gesammelt werden, *wieviele* Benutzer sich *von* welchem Ort *zu* welchem anderen Ort bewegen. Das erfordert einen größeren Aufwand und erklärt zudem nicht, *warum* sich die Leute in die eine oder andere Richtung bewegen.

Zum anderen können die Bewegungsabsichten von Benutzern dadurch ausgedrückt werden, dass man sie als eine Reihe von individuellen Entscheidungen auffasst ([O95]).

Viele Arbeiten, die sich mit dem Zugriff auf Informationen beschäftigen, entstanden mit Blick auf das WWW. Interessant ist z.B. eine Untersuchung über Zugriffe auf Internet-Seiten ([BCFPS99]), die zeigt, dass die relative Häufigkeit, mit der auf die i.-beliebteste Seite eines (Proxy-)Servers zugegriffen wird, ungefähr $1/i$ beträgt (sogenannte "Zipf"-Verteilung).

Bei [SWH98], einer Arbeit, die sich mit der Kommunikation zwischen mobilen und stationären Computern beschäftigt, wird angenommen, dass die Anzahl der Zugriffe durch mobile Stationen auf Informationseinheiten innerhalb einer bestimmten Zeiteinheit nach einer "Poisson"-Verteilung erfolgt.

Im Folgenden werden der Graph, auf dem sich die Bewegungen des Benutzers eines mobilen Systems abspielen, und das Benutzermodell, das für die entwickelte Simulationsumgebung grundlegend ist, vorgestellt. Das Benutzermodell besteht aus einem *Mobilitätsmodell*, einem *Informationsmodell* und einem *Zugriffsmodell*.

3.1 Der Infrastrukturgraph

Das Benutzermodell basiert auf einer Graph-Repräsentation des Gebiets, auf dem sich der Benutzer bewegt. Dieser Graph wird im Folgenden *Infrastrukturgraph* genannt. Seine *Knoten* verkörpern Orte, die von mobilen Benutzern besucht werden können, wie z.B. Sehenswürdigkeiten, Haltestellen oder Geschäfte. Die Verbindungen zwischen diesen Orten werden durch Straßen oder Schienen hergestellt und im Infrastrukturgraphen als *Kanten* ausgedrückt. Dies erscheint als eine plausible Modellierung eines realen Gebiets.

Die Kanten des Infrastrukturgraphen sind *ungerichtet* und *symmetrisch*, d.h. der Benutzer kann sich auf einer Kante in beide Richtungen fortbewegen und die Länge einer Kante ist für beide Richtungen dieselbe. Außerdem muss der Infrastrukturgraph *zusammenhängend* sein. Diese Forderung ist aus Sicht des Mobilitätsmodell (s.u.) sinnvoll, weil es in dem abgedeckten Gebiet keine "Inseln" geben darf, die für den Benutzer unerreichbar sind. Nicht zusammenhängende Infrastrukturgraphen werden von vorneherein als "ungeeignet" abgelehnt, da Algorithmen *im allgemeinen Fall* keinen Weg finden können, der beliebige Knoten des Graphen verbindet.

Die Position eines Knoten wird durch seine Lage in einem [x,y]-Koordinatensystem bestimmt. Dem Knoten, der in Abbildung 3.1 den *Marktplatz* bezeichnet, ist demzufolge das Koordinatenpaar (1,1) zugeordnet.

Eine Kante definiert sich aus den beiden Knoten, die sie verbindet. Ihre Länge ergibt sich aus der geographischen Distanz zwischen den zwei Knoten (Berechnung mittels des *Satz des Pythagoras*).

Das Koordinatensystem dient lediglich zur Bestimmung der *relativen* Position der Knoten *zueinander*. Es kann daher ohne weiteres verschoben oder skaliert werden, es sei denn, es besteht Interesse an den tatsächlichen Entfernungen.

Abbildung 3.1: Graph-Repräsentation von Tadmor

3.2 Mobilitätsmodell

Das *Mobilitätsmodell* regelt die Bewegungen des Benutzers. Es beschreibt die Bewegungswünsche des Benutzers als eine Reihe von Entscheidungen für oder gegen bestimmte Orte.

Diese Entscheidungen hängen vom Benutzerprofil (z.B. Tourist oder Geschäftsmann) und natürlich auch vom Infrastrukturgraphen (z.B. Paris oder Castrop-Rauxel) ab.

Jedem Ort des Infrastrukturgraphen ordnet das Mobilitätsmodell wird gemäß dem Benutzerprofil eine Besuchswahrscheinlichkeit zu. Aufgrund dieser Besuchswahrscheinlichkeit bestimmt das Mobilitätsmodell bei jedem Simulationslauf neu die sogenannten *Zielknoten*. Dies sind Knoten, die ein Benutzertyp bevorzugt besucht. Hat ein Knoten die Besuchswahrscheinlichkeit 0% zugeteilt bekommen, so kann er nie ein Zielknoten werden. Im Mittel wird ein Knoten mit einer Besuchswahrscheinlichkeit von 25% in jedem *vierten*, mit einer Besuchswahrscheinlichkeit von 100% in *jedem* Simulationslauf zum Zielknoten.

Die andere Kategorie von Knoten sind die *Durchreiseknoten*. Sie werden gezwungenermaßen besucht, weil sie auf der kürzesten oder gar einzigen Verbindung zwischen zwei

Zielknoten liegen. Die Durchreiseknoten ergeben sich erst bei der Suche nach einem Weg, der alle Zielknoten verbindet (s. Abschnitt 4.3).

Knoten, die weder Ziel-, noch Durchreiseknoten sind, werden während eines Simulationslaufes nicht beachtet. Sie können jedoch im nächsten Simulationslauf in eine dieser beiden Kategorien fallen.

3.3 Informationsmodell

In diesem Submodell wird modelliert, welche *Informationseinheiten* grundsätzlich an einem Ort angeboten werden. Die Aufgabe des Submodells ist daher die Zuteilung von Informationseinheiten an Orte. Informationseinheiten können beispielsweise WWW-Seiten, Kartenausschnitte oder sonstige Dateien sein. Die Orte sind analog zum Mobilitätsmodell vom zugrundeliegenden Infrastrukturgraphen abhängig.

Eine grundlegende Annahme beim Zugriff auf Informationseinheiten ist, dass Informationseinheiten nur dann abgefragt werden, wenn der Benutzer eines Informationssystems einen Ort besucht und nicht, während er sich von einem Ort zum anderen bewegt. Soll die Informationsabfrage auf dem Weg zwischen zwei Orten modelliert werden, so kann zwischen diesen beiden Knoten ein weiterer Knoten eingefügt werden.

Die Mengen der Informationseinheiten, die an den verschiedenen Knoten abgefragt werden können, müssen nicht disjunkt sein. In manchen Systemen ist es möglich, dass bestimmte Informationseinheiten an jedem Ort abgefragt werden können (z.B. eine Übersicht über alle Sehenswürdigkeiten bei einem mobilen Touristenführer); in anderen Systemen wiederum kann auf Informationseinheiten nur an *einem* Ort zugegriffen werden (z.B. aktuelle Kartenausschnitte bei Navigationshilfen).

3.4 Zugriffsmodell

Die Aufgabe des *Zugriffsmodell* besteht darin, die *Verteilung* der Zugriffe auf Informationseinheiten an einem Ort zu bestimmen. Die Informationseinheiten an sich liefert das Informationsmodell. Welche und wieviele davon allerdings tatsächlich abgefragt werden, regelt das Zugriffsmodell.

Die Wahrscheinlichkeit, mit der auf eine Informationseinheit zugegriffen wird, legt eine *Wahrscheinlichkeitsverteilung* fest. Je nach Anwendung und Aufwand wird eine Gleich-, "Zipf"- (s. Abbildung 3.2) oder eine andere Verteilung am passendsten sein.

Zwei Instanzen von Zugriffsmodellen könnten sich etwa darin unterscheiden, dass die eine beim Besuch eines Ortes immer dieselben Informationseinheiten anbietet, während die andere beim wiederholten Besuch keine Informationseinheiten offeriert und somit ein *caching* durch das mobile Endgerät simuliert.

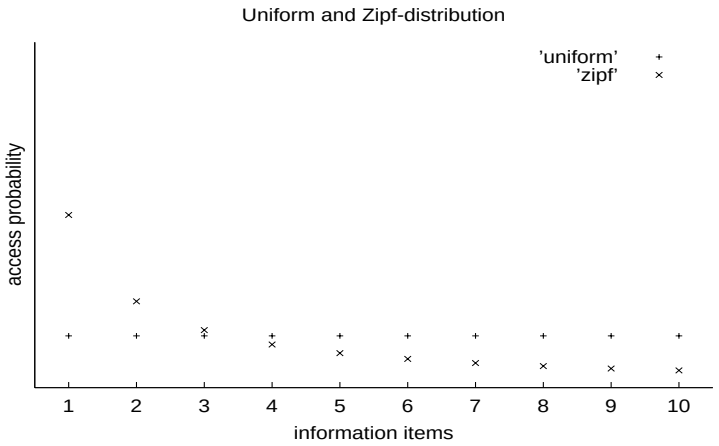

Abbildung 3.2: Gleich- und "Zipf"-Verteilung

3.5 Abhängigkeiten

Noch nicht angesprochen wurde, dass die drei Submodelle voneinander *abhängen* können. Eine offensichtliche Abhängigkeit besteht zwischen dem Zugriffsmodell und dem Informationsmodell: das Informationsmodell bestimmt die Zuordnung von Informationseinheiten zu Knoten, also welche Mengen von Informationseinheiten an einem Knoten abgefragt werden können. Das Zugriffsmodell seinerseits wählt aus diesen Mengen diejenigen Informationseinheiten aus, die dann auch tatsächlich abgefragt werden.

Eine andere Abhängigkeit kann zwischen Informationsmodell und Mobilitätsmodell bestehen: so kann z.B. das Informationsmodell die Anzahl der Informationseinheiten von der Besuchswahrscheinlichkeit der Knoten abhängig machen, welche vom Mobilitätsmodell festgelegt wird.

Wenn diese zwei Abhängigkeiten zustande kommen und auch noch das Mobilitätsmodell vom Informationsmodell abhängt, z.B. wenn sich die Besuchswahrscheinlichkeit eines Knotens nach den abfragbaren Informationseinheiten richtet, dann besteht eine *zyklische* Abhängigkeit. Diese - egal ob sie direkt oder transitiv zustande gekommen ist - kann nicht aufgelöst werden, d.h. es kann nicht entschieden werden, in welcher Reihenfolge die Konfiguration der Submodelle vor sich gehen muss.

Unter der Voraussetzung, dass die Einbindung der verwendeten Submodelle in die Simulationsumgebung auf die richtige Weise erfolgt ist (die direkten Abhängigkeiten eines Submodells müssen dabei angegeben werden), kann von der Applikation eine zyklische Abhängigkeit erkannt und ein Simulationslauf unterbunden werden.

Kapitel 4

Der Simulationslauf-Algorithmus

Der Simulationslauf-Algorithmus steuert den Ablauf der Simulation. Im vorherigen Kapitel wurde gezeigt, wie der Ablauf von den drei Submodellen (s. Abschnitte 3.2-3.4) und dem Infrastrukturgraphen (s. Abschnitt 3.1) abhängt. Bei der Evaluierung des Prototypen hat es sich als nützlich erwiesen, in der Applikation weitere Konfigurationsmöglichkeiten zu bieten:

- Die Festlegung einer Anzahl minimal/maximal zu besuchender Knoten (s.u.).
- Die Angabe, ob der Weg über die Knoten *geschlossen* sein soll. Geschlossen bedeutet, dass der Anfangsknoten gleich dem Endknoten des Weges ist, man macht also eine "Rundreise".
- Die Auswahl eines *Algorithmus*, mit dem ein Weg über die Knoten, nachdem die zu besuchenden Knoten einmal festgelegt sind (Zielknoten), bestimmt wird (s.u.).

Zusätzlich kann der Simulationsbenutzer noch einstellen, *wieviele* Simulationsläufe voneinander unabhängig und hintereinander, aber mit denselben Einstellungen, ausgeführt werden sollen.

4.1 Der Simulationslauf

Nach der Konfiguration kann die Simulation gestartet werden. Der vereinfachte Ablauf der Simulation wird in Abbildung 4.1 in Pseudocode beschrieben.

Zunächst wird über das Mobilitätsmodell festgestellt, *welche* Knoten/Orte der Benutzer eines mobilen Informationssystems sich zu besuchen vornimmt (Abschnitt 4.2).

```
procedure simulationrun();

var vertices : set of vertices;
    walk : array of vertices;
    v : vertex;
    items: array of items;
    counter: counter for vertices and information items;

begin

  for j := 1 to NumberOfRuns do
    vertices := mobilityModel.determineVertices(infrastructureGraph);
    walk := calculateWalk(vertices);

    for i := walk.firstVertex to walk.lastVertex do
      v := walk[i];
      items := accessModel.getAccessedItems(v);
      counter.note(v, items);
    endfor

  endfor

end
```

Abbildung 4.1: Pseudocode Simulationslauf

Dann erfolgt die Berechnung eines Weges der mindestens diese Knoten miteinander verbindet. Zur Erinnerung: so ein Weg muss existieren, da der Infrastrukturgraph zusammenhängend sein muss. Je nach Konfiguration ist dieser Weg geschlossen oder nicht (Abschnitt 4.3).

Für jeden Knoten des Weges wird dann mittels des gewählten Zugriffsmodells ermittelt, welche Informationseinheiten dort abgefragt werden (Abschnitt 4.4).

Wie schon angedeutet, werden die Simulationsläufe voneinander unabhängig durchgeführt. Allerdings werden die *Ergebnisse* aller Simulationsläufe einer *Simulationsreihe* zusammengefasst und ausgegeben (Abschnitt 4.5). Zu einer Simulationsreihe gehören alle *unmittelbar hintereinander* vom Simulationsprogramm ausgeführten Simulationsläufe.

Der Benutzer der Simulationsumgebung die Möglichkeit hat, den Simulationsvorgang jederzeit vorzeitig zu beenden. Dies könnte bisweilen nötig sein, da es im Voraus nicht immer abzusehen ist, wie rechen- und zeitaufwendig die Simulation bei einer bestimmten Konfiguration ist. Bei einem Abbruch werden dann nur die vollständigen Simulationsläufe gezählt.

4.2 Bestimmung der zu besuchenden Knoten

Die Anzahl der vom Mobilitätsmodell gelieferten Zielknoten muss zwischen den bei der Konfiguration angegebenen Grenzen sein. Die Gründe für die Einführung dieser Grenzen werden im nächsten Abschnitt über die Wegesuche erklärt. Gegebenenfalls wird wiederholt versucht, eine passende Knotenmenge zu suchen. Wenn dies nach einer bestimmten Anzahl von Versuchen nicht glückt, dann bleiben als Alternativen das vorzeitige Stoppen der Simulation durch den Benutzer und ein intensiveres Suchen (d.h. die Anzahl der Versuche wird erhöht).

Die Gründe für das Missglücken sind darin zu suchen, dass das Mobilitätsmodell nichts von den Grenzen für die Zielknoten "weiß" und sich bei deren Ermittlung nur auf die den Knoten zugewiesenen Besuchswahrscheinlichkeiten verlässt (s. Pseudocode in Abbildung 4.2).

Wenn z.B. alle zehn Knoten eines Infrastrukturgraphen die Besuchswahrscheinlichkeit 0,95 besitzen, dann liegt die Wahrscheinlichkeit, dass das Mobilitätsmodell zwischen null und sieben Zielknoten liefert bei

$$\sum_{i=0}^{7} \binom{10}{i} 0,95^i \cdot 0,05^{10-i} = 0,0115\ldots$$

```
function determineVertices(infrastructureGraph)
  returns set of vertices;
var B := {} : set of vertices;
    v : vertex;
    p : array of vertex visit probabilities (between 0.0 - 1.0);
    r : real;
begin
  forall v of infrastructureGraph do
    0.0 ≤ r := randomNumber() < 1.0;
    if (r < p[v]) then B := B ∪ {v};
  endfor
  return B;
end
```

Abbildung 4.2: Pseudocode für die Bestimmung der Knoten

Das bedeutet, dass in beinahe 99 von 100 Fällen acht oder mehr Zielknoten besucht werden müssen. Anders ausgedrückt: wenn die Anzahl der Zielknoten auf einschließlich sieben begrenzt wurde, dann ist unter 100 vom Mobilitätsmodell ermittelten Knotenmengen nur eine geeignete dabei.

4.3 Wegesuche

Ein Weg in einem Graphen ist eine Folge von Knoten $(x_0, x_1, \ldots x_r)$, wobei (x_i, x_{i+1}) eine Kante des Graphen ist. Als *Länge eines Weges* wird hier die Summe der Längen der Kanten bezeichnet (und nicht die Anzahl der Knoten des Weges). Die Länge einer Kante ergibt sich aus der geographischen Distanz der zwei verbundenen Knoten.

Bei der Wegesuche versucht man, einen Weg zu finden, der eine vorgegebene Menge von Knoten (eine Untermenge aller Knoten des Graphen) verbindet. Als Frage bleibt: soll der Weg möglichst kurz sein, soll er möglichst wenig andere Knoten enthalten oder sind solche Eigenschaften nebensächlich?

Bei dieser Simulationsumgebung wird nach dem kürzesten Weg gesucht. Dies ist sinnvoll, weil das Fortbewegungsverhalten von Menschen simuliert wird, die gewöhnlich bestrebt sind, keine unnötig langen Umwege zu machen.

Dazu wird anfangs ein Graph aus dem ursprünglichen Infrastrukturgraphen erstellt, der nur die Zielknoten (s. Abschnitt 3.2) enthält. Alle kürzesten Pfade zwischen diesen

Zielknoten, die im allgemeinen Fall nicht aus nur einer Kante bestehen[1], werden mit dem *Floyd-Warshall*-Algorithmus (z.B. in [S98]) berechnet und zusammen mit ihrer Länge in einer Matrix abgespeichert. Auf diesen Pfaden liegen die Durchreiseknoten, über die somit *abstrahiert* wird (In Abschnitt 4.4 wird in einem Beispiel verdeutlicht, wie die Abstrahierung und die anschließende Rückführung auf den Originalgraphen vor sich geht). Die Problemstellung hat sich folglich reduziert von der Suche nach einem Weg über *bestimmte* Knoten eines Graphen zu der Suche nach einem Weg über *alle* Knoten eines kleineren, vollständigen Graphen.

Dies ist eine Variation des *Traveling Salesman Problem* (TSP), einem NP-vollständigen Problem, d.h. es existiert vermutlich kein Algorithmus, der es in polynomieller Zeit löst. Im Folgenden werden drei Lösungsansätze für dieses Problem beschrieben.

4.3.1 Enumeration (OPT)

Die Enumeration ist der klassische Ansatz zur Lösung von kombinatorischen Optimierungsproblemen, der auch hier verfolgt wird. Es gibt bei n Knoten $n!$ unterschiedliche Wege, die diese Knoten verbinden und die alle überprüft werden müssen. Es ist klar, dass dieser Ansatz somit die Komplexität $O(n!)$ besitzt. Aufgrund dieser Komplexität ist man bestrebt, diesen Ansatz, der mit Sicherheit einen kürzestmöglichen Weg findet, nur mit einer geringen Anzahl von Knoten zu verfolgen. Das erklärt auch die Einführung der *oberen Knotengrenze*[2] (s. Abschnitt 4.2), die die Anzahl der Zielknoten einschränkt.

Als Beispiel: die Berechnung aller 9! (362 880) möglichen Permutationen der Zahlen von 0 bis 8 nimmt 1,7 Sekunden in Anspruch (Intel Pentium 133 MHz). Wenn man zehn Zahlen (3 628 800 Permutationen) nimmt, dann dauert das zehnmal so lange, also 17 Sekunden. Bei elf Zahlen (3 Minuten, 39 916 800 Permutationen) ist es empfehlenswert, die Berechnung als Hintergrund-Prozess ablaufen zu lassen. Vor allem, wenn man bedenkt, dass man oftmals mehr als nur *einen* Simulationslauf durchführen will und ein Simulationslauf eben nicht nur aus der Aufzählung aller Permutationen besteht.

Eine sinnvolle Grenze für die Berechnung des optimalsten - d.h. kürzesten - Weges liegt auf Standard-PCs bei ca. zehn Knoten. Aber was macht man, wenn man mehr als zehn Knoten besuchen will? Dann muss sich *Heuristiken* bedienen, die idealerweise eine polynomielle Laufzeit haben und eine obere Grenze für die Weglänge garantieren. Bei der Verwendung von Heuristiken für das TSP muss man berücksichtigen, dass man dort meist von einem kompletten Graphen ausgeht oder man einen Weg sucht, der alle Knoten nur einmal besucht. Hier wird aber ein Weg gesucht, der nur eine Teilmenge

[1] Es sei denn, der Graph ist komplett, d.h. alle Knoten sind direkt miteinander verbunden

[2] Die *untere Knotengrenze* erklärt sich nicht so sinnvoll wie die obere Knotengrenze. Man vermeidet damit allzu kurze Wege, die nicht so sehr aussagekräftig sind.

von Knoten (die Zielknoten) auf dem ursprünglichen Originalgraphen verbindet; ungeachtet dessen, ob diese oder die restlichen Knoten mehrmals besucht werden. Durch das Erstellen eines vollständigen, kleineren Graphen und der Pfadmatrix wird dieses Problem umgangen.

Bei der in dieser Studienarbeit programmierten Simulationsumgebung werden zwei Heuristiken angeboten: die einfache *Nearest Neighbour (NN)*-Heuristik und die etwas aufwendiger zu programmierende *Minimum Spanning Tree (MST)*-Heuristik.

4.3.2 Nearest Neighbour (NN)

Die *Nearest Neighbour*-Heuristik ist eine sehr einfache Heuristik (s. Pseudocode in Abbildung 4.3) und arbeitet wie folgt: beginne mit irgendeinem Knoten und besuche dann immer den nächstgelegenen - noch nicht besuchten - bis alle Knoten erreicht sind und kehre dann schließlich zum Anfangsknoten zurück (wenn der Weg geschlossen sein soll).

Der Aufwand des Algorithmus, der in dieser Form auch in der Studienarbeit verwendet wurde, beträgt $O(n^2)$. Bei Verwendung eines Heaps für das Finden des nächstgelegenen Knotens ergibt sich für die Komplexität $O(n \lg n)$. Damit ist dieser Algorithmus bedeutend schneller als die Enumeration, es existiert jedoch keine Garantie für die Weglänge. Die erzeugten Wege dienen oft auch als Ausgangsbasis für andere Heuristiken. Seine Schwäche liegt darin, dass unter Umständen Knoten "vergessen" werden und dann hinterher teuer in den Weg aufgenommen werden müssen (s. Abbildung 4.4).

4.3.3 Minimum Spanning Tree (MST)

Ein wegen seiner Weglänge-Garantie beliebter Algorithmus, dessen Ergebnisse meist 15% bis 20%, bewiesenermaßen höchstens aber 100% über dem Optimum liegen, ist die sogenannte *MST*-Heuristik. Dabei wird folgendermaßen vorgegangen:

1. wähle einen Knoten r als den Wurzelknoten
2. erstelle von r ausgehend einen MST (beispielsweise nach dem Algorithmus von Prim [P57])
3. durchlaufe diesen Baum in Präorder-Reihenfolge und notiere in dieser Reihenfolge jeden erstmals besuchten Knoten
4. die Kanten zwischen diesen Knoten bilden den gefundenen Weg

Die Komplexität dieses Algorithmus wird von der Erstellung des MST bestimmt; bei der Verwendung eines *Heaps* beläuft sie sich ebenfalls auf $O(n \lg n)$.

```
function nearestNeighbour(V: set of vertices)
  returns (vertex,array of vertices)

var edge[i,j] : edge between vertex i and vertex j
    initial, v, w, p : vertex;
    next : array of vertices;[1ex] begin
  initial ∈ V;
  V := V \ initial;
  v := initial;

  while V≠{} do

    l := ∞;

    forall w ∈ V do
      if edge[v,w].length < l then
        l := edge[v,w].length;
        p := w;
      endif

    endfor

    next[v] := p;
    v := p;
    V := V \ v;
  endwhile

  if walkIsClosed then next[v] := initial
    else next[v] := null;

  return (initial,next);

end
```

Abbildung 4.3: Pseudocode *Nearest Neighbour*-Heuristik

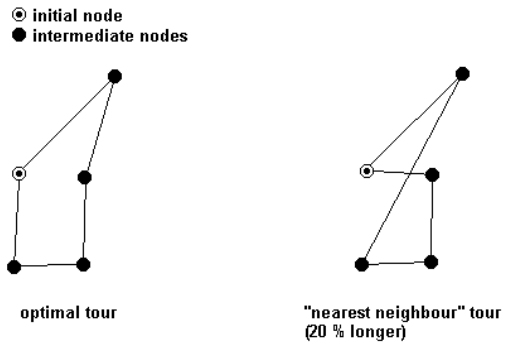

Abbildung 4.4: Schwäche der *Nearest Neighbour*-Heuristik

4.3.4 Weitere Heuristiken

Die obigen zwei Heuristiken wurden in der Studienarbeit wegen ihrer Einfachheit und weiten Verbreitung implementiert. An dieser Stelle seien noch weitere Heuristiken vorgestellt, die zwar meist bessere Lösungen für das *TSP* finden, aber komplizierter sind: das Verfahren nach *Christofides* ([C76]) baut auch auf einem MST auf, garantiert aber eine Tourlänge, die maximal das 1,5-fache der kürzesten Tour beträgt.

Eine andere Klasse von Heuristiken sind die *k-optimalen Touren* (auch als *Lin-Kernighan*-Ansatz ([LK73]) bekannt). Dabei werden - von einer Anfangstour ausgehend - so lange k Kanten gelöscht und durch kürzere ersetzt (die Toureigenschaft muss erhalten bleiben), bis ein (lokales) Optimum gefunden wird. Erfahrungsgemäß liegt die Länge solcher Touren nur ein paar Prozent über der optimalen Länge. Als k wird im allgemeinen 2 oder 3 gewählt.

Bei der *Insertion Method* (Implementierung z.B. in [SDK83]) geht man von einer Teiltour (enthält nur eine Teilmenge aller Knoten) aus und fügt nach und nach die restlichen Knoten ein. Als Fragen bleiben: Mit welcher Teiltour startet man, welchen Knoten fügt man als nächsten ein und wo fügt man diesen ein? Als Ausgangstour dient oft eine auf denjenigen drei Knoten basierende Tour, die das größte Dreieck bilden. Eingefügt werden die neuen Knoten gewöhnlich dort, wo die Tourverlängerung am geringsten ist. Als nächster einzufügender Knoten kann z.B. der Knoten gewählt werden, der die kleinste Verlängerung der Tour verursacht (*Cheapest Insertion*).

4.3.5 Vergleich der programmierten Algorithmen

Nach dem Programmieren wurden die drei Algorithmen (OPT, MST, NN) für die Wegesuche (auf einem Intel Pentium 133 MHz) einem Vergleich in bezug auf Laufzeit

und Weglänge unterzogen. Zusätzlich wurde bei der Gegenüberstellung der Weglängen eine zufällige Knotenreihenfolge (RD) miteinbezogen.

Als zugrundeliegende Infrastrukturgraphen wurden der mit Kanten eher dünn besetzte Stuttgart/Innenstadt-Graph und ein zufällig erstellter vollständiger Graph mit 25 Knoten verwendet (s. Abbildungen 2.1 und 4.5). Hier sei schon vorweggenommen, dass sich die Ergebnisse bei beiden Graphen nicht bedeutend unterschieden haben und nur die Ergebnisse für den Stuttgart/Innenstadt-Graphen dargestellt wurden. Für jeweils

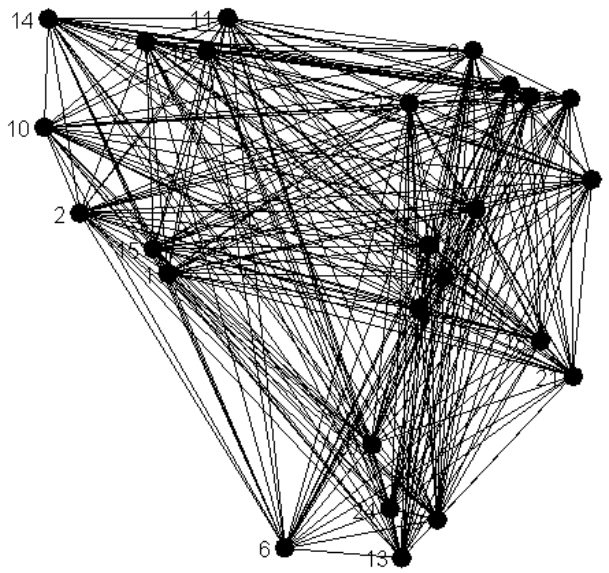

Abbildung 4.5: Zufälliger vollständiger Graph mit 25 Knoten

zwei bis 20 zufällig ausgewählte (Ziel-)Knoten[3] wurden in den Graphen möglichst kurze Wege gesucht. Wie oft dies geschah, kann der Tabelle 4.1 entnommen werden. Für die

	2-6	**7-8**	**9-10**	**11-20**
OPT	100,000	10,000	100	0
MST	100,000	10,000	100	10,000
NN	100,000	10,000	100	10,000
RD	100,000	10,000	100	10,000

Tabelle 4.1: Anzahl der Messungen

[3]Bei geschlossenen Wegen wird der Endknoten nicht mitgezählt.

unterschiedlichen Algorithmen handelte es sich um dieselben Knoten. Aus Laufzeit-Gründen wurde die Messung ab elf Knoten nicht mehr mit dem Enumerationsalgorithmus (OPT) durchgeführt. Eine Zeitmessung bei der zufälligen Knotenreihenfolge (RD) wurde nicht vorgenommen, da keine Wegberechnung stattfindet.

Die Abbildungen 4.6 und 4.7 zeigen die durchschnittliche Weglänge in Metern für geschlossene und nicht geschlossene Wege. Dabei kommt es jeweils weniger auf die Absolutwerte an als auf die relativen Unterschiede zwischen den Algorithmen.

In Abbildung 4.8 wird die durchschnittliche Zeit für die Berechnung von 100 Wegen in μs (10^{-6} Sekunden) dargestellt. Die y-Achse ist aufgrund der schnellwachsenden Kurve für den Enumerationsalgorithmus (OPT) im logarithmischen Maßstab (Basis 10) skaliert.

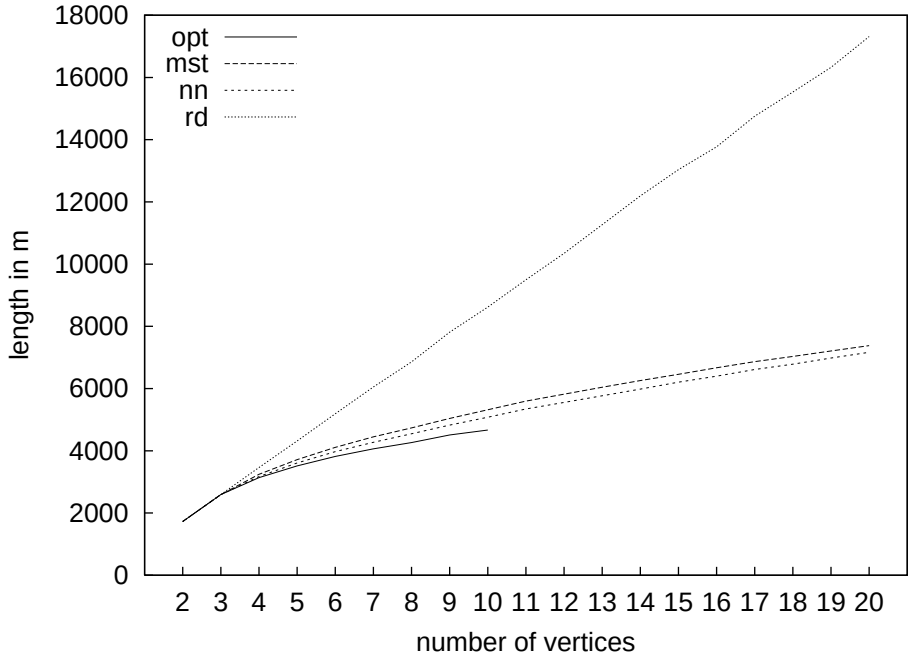

Abbildung 4.6: Vergleich der Weglängen (für geschlossene Wege)

Die Ergebnisse offenbaren teilweise Erstaunliches: die Länge für geschlossene Wege des *Nearest Neighbour*-Algorithmus liegt im Mittel doch überraschend nahe an der kürzestmöglichen (nur ein bis zehn % länger), während die Weglänge beim *Minimum Spanning Tree*-Algorithmus zwischen drei und 15% länger als die optimale ist. Auch bei nicht geschlossenen Wegen[4] schneidet der *NN*-Algorithmus durchgehend besser

[4]Der *MST*-Algorithmus taucht eigentlich nur in Verbindung mit geschlossenen Wegen auf

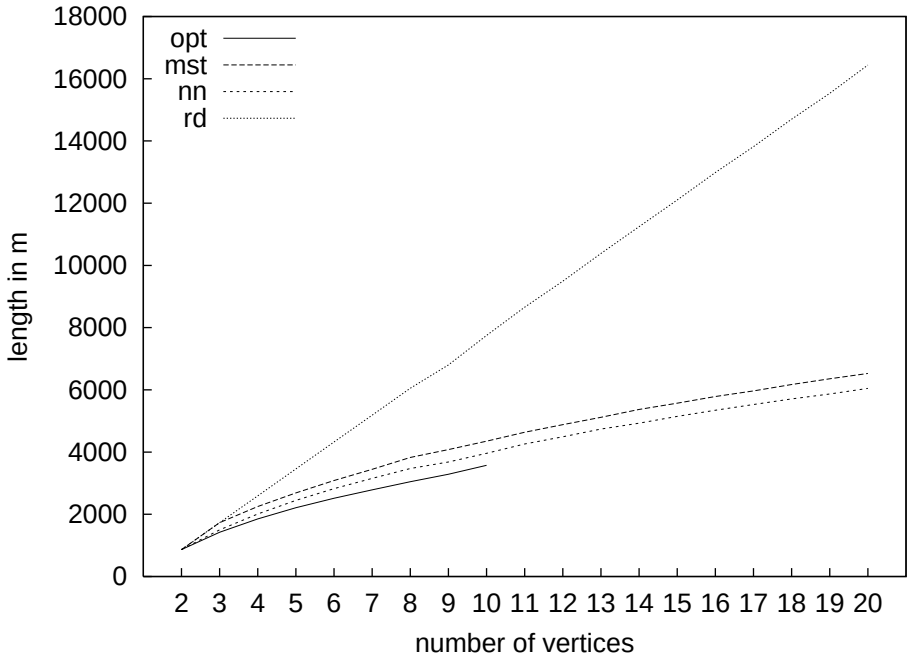

Abbildung 4.7: Vergleich der Weglängen (für nicht geschlossene Wege)

ab als der *MST*-Algorithmus (acht bis 16% länger gegenüber 21-27% länger als das Optimum). Selbst die erwarteten Ausreißer (nicht dargestellt) beim *NN*-Algorithmus waren kürzer als die beim *MST*-Algorithmus. Nicht selten wurde bei *NN* für eine geringe Knotenanzahl (unter acht) der kürzeste Weg gefunden.

Die Laufzeiten hingegen erwiesen sich so wie erwartet: ab zehn Knoten ist der OPT-Algorithmus kaum mehr tragbar. Der *NN*-Algorithmus war ca. fünf bis sechs mal schneller als der *MST*-Algorithmus.

4.4 Ermittlung der abgefragten Informationseinheiten

Vor der Ermittlung der Informationseinheiten, die an den Orten bzw. Knoten abgefragt werden, steht die *Expandierung* des Weges, der alle Zielknoten enthält, zu einem Weg, in dem auch die Durchreiseknoten enthalten sind, über die zuvor abstrahiert wurde (s. Abschnitt 4.3).

Als Beispiel betrachte man die Zielknoten, die das Mobilitätsmodell für die Einstellung [min./max. Knoten=4] für den Infrastrukturgraphen aus Abbildung 2.1 liefern könn-

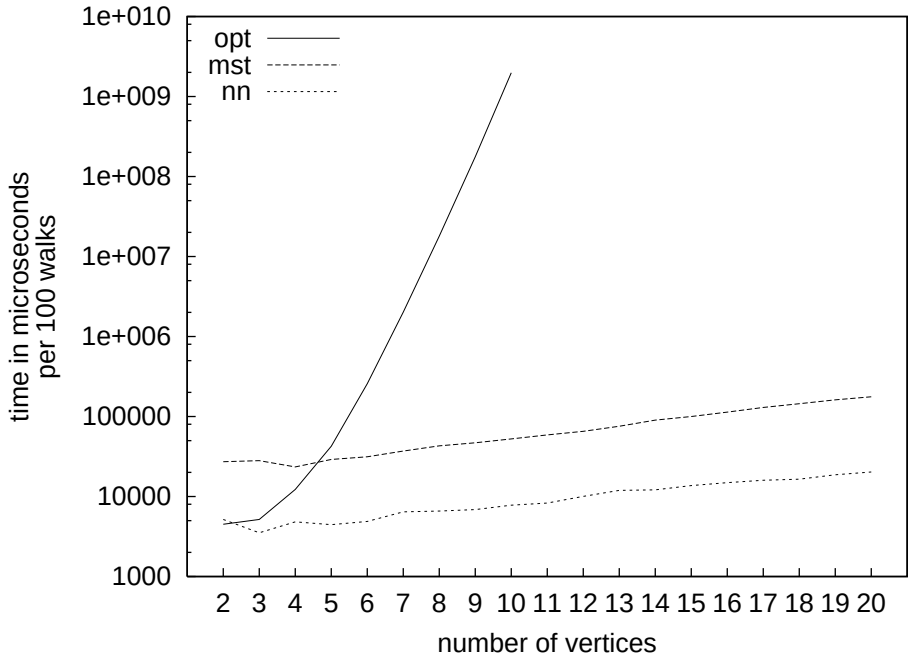

Abbildung 4.8: Vergleich der Laufzeiten

te: (8, 18, 24, 26). Der daraus resultierende Subgraph mit den kürzesten Pfaden ist in Abbildung 4.9 dargestellt. Im nächstem Schritt folgt die Abstrahierung über die Durchreiseknoten 5, 6, 7, 9, 12, 17, 19, 20, 21, 37 (Abbildung 4.10). Der kürzeste Weg, der verbleibenden (Ziel-)Knoten miteinander verbindet ist (24, 8, 18, 26) (Abbildung 4.11). Nach der Rückführung auf den Original-Infrastrukturgraphen werden letztendlich die Knoten (24, 12, 9, 8, 7, 18, 21, 37, 17, 26) besucht (Abbildung 4.12).

4.5 Ergebnisse eines Simulationslaufes

Die Erfassung der Simulationsergebnisse erfolgt auf zweierlei Arten: erstens kann sich der Benutzer der Simulation die bei den Simulationsläufen besuchten Knoten und abgefragten Informationseinheiten in einer *Log-Datei* (s. Abschnitt 6.13.4) protokollieren lassen.

Die andere Art der Erfassung sind die *Zähler*. Diese gibt es für sieben Fälle:

1. Für jeden Knoten des Infrastrukturgraphen wird mitgezählt, wie oft er während der Simulationsreihe besucht wurde.

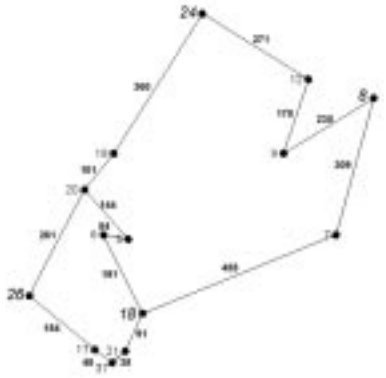

Abbildung 4.9: Subgraph mit kürzesten Pfaden zwischen den Zielknoten

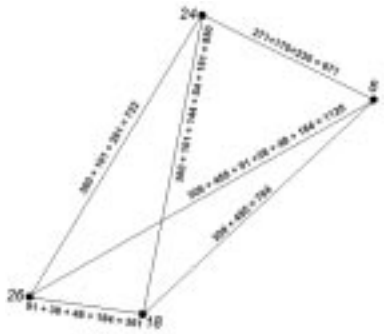

Abbildung 4.10: Nach Abstrahierung über die Durchreiseknoten

Abbildung 4.11: Kürzester Weg über die Zielknoten

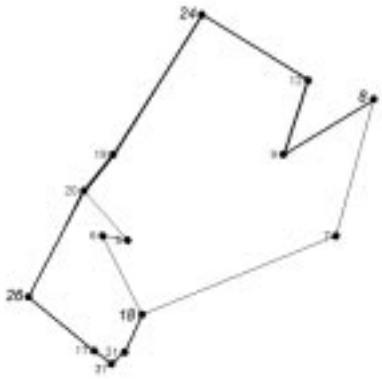

Abbildung 4.12: Nach Rückführung auf Originalgraph

2. Außerdem werden für jeden Knoten die Anzahl der Informationseinheiten, die dort abgefragt wurden, registriert.

3. Für jede Kante wird ermittelt, wie oft sie *insgesamt* während einer Simulationsreihe benutzt wurde.

4. Zusätzlich wird für jede Kante festgestellt, wie oft sie bei einem Simulationslauf *erstmals* während einer Simulationsreihe benutzt wurde.

5. Ebenfalls aufgezeichnet wird die Anzahl der Zugriffe auf jede Informationseinheit.

6. Es existiert auch ein Zähler für die Besuche von Feldern, in die das Gebiet, das der Infrastrukturgraph abdeckt, aufgeteilt wurde.

7. Wie bei den Kanten wird auch bei den Feldern - zusätzlich zu der Gesamtzahl von Besuchen - nur der *erstmalige* Besuch in einem Simulationslauf gezählt.

Alle diese Zähler können *textuell* dargestellt (s. Abschnitt 6.3.3) und in einer Datei abgespeichert (s. Abschnitt 6.13.5) werden. Zudem werden die Besuchshäufigkeiten der Felder in der Besuchswahrscheinlichkeitskarte (s. Abschnitt 6.10) *graphisch* dargestellt.

Kapitel 5

Einsatzbeispiel

Bei dem Einsatzbeispiel geht es darum, drei miteinander kooperierende Submodelle zu erstellen. Dabei wurde das Prinzip aus [KR00] der "dedicated" und "non-dedicated" Knoten aufgegriffen, d.h. es gibt zwei Arten von Knoten mit jeweils unterschiedlicher Besuchswahrscheinlichkeit. Momentan werden für jeden der drei Submodelltypen zwei verschiedene Submodell-Instanzen angeboten: eine einfache mit keinerlei Konfigurationsmöglichkeiten für die erste Testphase und eine kompliziertere, welche in den folgenden Abschnitten für jeden Submodelltyp einzeln beschrieben wird.

5.1 Mobilitätsmodell

Manche Knoten werden bevorzugt besucht und haben daher eine höhere Besuchswahrscheinlichkeit. Die Vorliebe für bestimmte Orte wird vom Benutzertypus bestimmt: ein *Durchreisender* favorisiert Orte mit Anbindung an das öffentliche Verkehrsnetz, ein *Tourist* hingegen hält sich überwiegend dort auf, wo es Sehenswürdigkeiten gibt. So werden die Knoten eines Infrastrukturgraphen in zwei Klassen aufgeteilt[1]: potentiell vorrangig besuchte Knoten erhalten vom Benutzer der Simulationsumgebung eine höhere Besuchswahrscheinlichkeit zugewiesen.

Der Nachteil dieser Art der Zuteilung ist, dass das Erfassen der bevorzugten Knoten im Quellcode zu unflexibel ist, da man fest an einen bestimmten Infrastrukturgraphen gebunden ist. Darum wurde bei der Entwicklung des Submodells auch die Möglichkeit vorgesehen, dass bei der Konfiguration des Submodells der Benutzer die IDs der Knoten über die Benutzungsoberfläche ändern (s. Abbildung 5.1) kann.

[1] Dies ist eine sehr simple Aufteilung; es sind sehr wohl differenziertere Mobilitätsmodelle denkbar, bis hin zur direkten Zuweisung einer bestimmten Besuchswahrscheinlichkeit an jeden einzelnen Knoten.

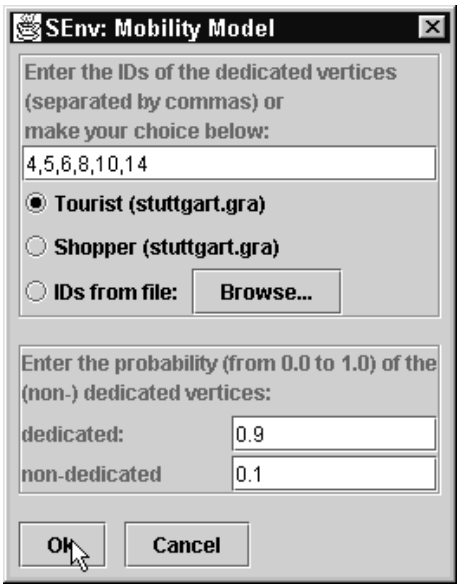

Abbildung 5.1: Konfiguration des Mobilitätsmodells

Da dies auf Dauer zu mühsam ist, können die Knoten-IDs auch über eine ASCII-Datei geladen werden (in jeder Zeile steht eine Knoten-ID). Gültige Besuchswahrscheinlichkeiten sind reelle Zahlen im Bereich von 0.0 bis einschließlich 1.0. Die IDs der Knoten müssen durch *Kommata* (keine Leerzeichen) getrennt werden.

5.2 Informationsmodell

Das Informationsmodell richtet sich in gewisser Weise nach dem Mobilitätsmodell. Die Anzahl der Informationseinheiten, die an einem Knoten abgefragt werden können, hängt davon ab, ob es sich um einen bevorzugt angesteuerten Ort handelt. Dann sind auch mehr Informationseinheiten verfügbar, die interessant sein könnten.

Bei der Konfiguration (s. Abbildung 5.2) muss die den Knoten zugeordnete Anzahl von Informationseinheiten und eine Grenze zur Unterscheidung zwischen den vorzugsweise besuchten und den anderen Knoten eingegeben werden.

5.3 Zugriffsmodell

Ein Teil des Zugriffsmodells (s. Abbildung 5.3) ähnelt dem Informationsmodell; auch hier muss eine Grenze für die Besuchswahrscheinlichkeit zur Unterscheidung beider

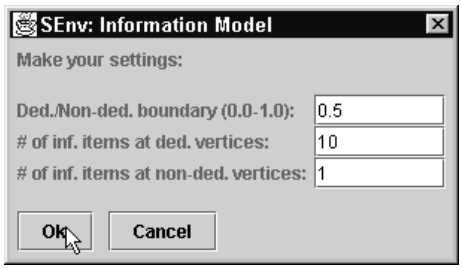

Abbildung 5.2: Konfiguration des Informationsmodells

Klassen von Knoten und eine Anzahl für die Informationseinheiten eingegeben werden. Nur handelt es sich eben hier nicht um die den Knoten zugeordneten Informationseinheiten, sondern um diejenigen, die abgefragt werden.

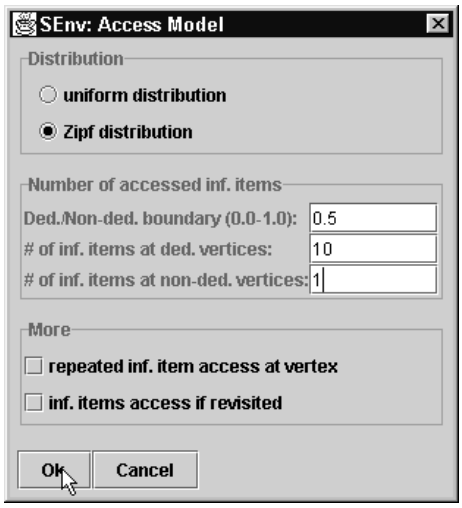

Abbildung 5.3: Konfiguration des Zugriffsmodells

Zudem wurden zwei unterschiedliche Verteilungsstrategien implementiert, von denen immer nur eine zum Einsatz kommen kann: eine Gleich- und eine "Zipf"-Verteilung. Bei der Gleichverteilung wird - wie der Name schon sagt - auf jede Informationseinheit, die das Informationsmodell für einen Knoten liefert, mit *derselben* Wahrscheinlichkeit zugegriffen. Es genügt also, entsprechend oft aus den möglichen Informationseinheiten eine beliebige herauszugreifen.

Die "Zipf"-Verteilung verhält sich komplizierter: auf die i.-beliebteste Informationseinheit wird mit der Wahrscheinlichkeit $1/i$ zugegriffen. Als beliebteste Informationseinheit wird diejenige unter den vom Informationsmodell gelieferten angenommen, die die

niedrigste ID besitzt. Voraussetzung ist, dass das Informationsmodell während einer Simulationsreihe einem Knoten immer dieselben Informationseinheiten zuordnet.

Es gibt noch zwei weitere Konfigurationsmöglichkeiten: zum einen kann der Simulationsbenutzer einstellen, ob beim wiederholten Besuch eines Knotens während eines Simulationslaufes nochmals Informationen abgefragt werden sollen oder nicht ("inf. items access if revisited"). Die andere Einstellungsmöglichkeit regelt, ob während des Besuchs eines Knotens eine bestimmte Informationseinheit mehrmals abgefragt werden kann ("repeated inf. item access at vertex").

5.4 Anwendung des Einsatzbeispiels

Zur Demonstration der Auswirkungen der verschiedenen Konfigurationsmöglichkeiten werden im Folgenden drei Simulationsreihen, die aus jeweils 1000 Simulationsläufen bestehen, beschrieben. Als Grundlage dient ein Infrastrukturgraph, der einen Teil des U- und S-Bahn-Liniennetzes der Stadt Stuttgart (Stuttgarter Straßenbahnen AG) widergibt (s. Abbildung 5.4).

Es werden beispielhaft zwei unterschiedliche Aspekte miteinander verglichen: der Einfluss, den das Mobilitätsmodell durch die Vorgabe der Zielknoten ausübt, und die Auswirkungen, die sich durch die Zugriffsverteilung der Informationseinheiten ergeben.

Folgende Simulationsreihen wurden durchgeführt:

1. Im Mobilitätsmodell wird ein Benutzertyp "Student, der öffentliche Verkehrsmittel benutzt" nachgestellt. Dabei wird angenommen, dass der Student mit 90%iger Wahrscheinlichkeit die vier Orte bzw. Haltestellen *Hauptbahnhof, Keplerstrasse, Universität* und *Vaihingen Bahnhof* besucht. Das erscheint plausibel, weil sich in der Nähe dieser Haltestellen die meisten Universitätseinrichtungen befinden. Alle restlichen Orte des Infrastrukturgraphen besitzen die Besuchswahrscheinlichkeit 10%.

2. Des weiteren wird ein Benutzertyp "Tagesausflügler" imitiert. Dieser bevorzugt Haltestellen, die nahe bei Ausflugszielen liegen: *Wilhelma, Staatsgalerie, Hauptbahnhof, Schlossplatz, Charlottenplatz, Stadtmitte* und *Rathaus*. Auch hier wird eine Besuchswahrscheinlichkeit von 90% und 10% angenommen.
   Das Informationsmodell teilt den Haltestellen der Ausflugsziele *zehn*, den restlichen Haltestellen *eine* Informationseinheit zu.
   Die Einstellungen des Verteilungsmodells nehmen eine *Gleichverteilung* des Zugriffs auf die Informationseinheiten an.

3. Wie 2., nur dass an die Stelle der Gleichverteilung eine *"Zipf"-Verteilung* tritt.

Die ersten beiden Simulationsreihen werden bezüglich der Bewegungsmuster verglichen. Als Algorithmus zur Wegesuche kam der *Nearest Neighbour*-Algorithmus zum Zuge. Besuchswahrscheinlichkeitskarten (s. Abschnitt 6.10) zeigen graphisch die relative Häufigkeit an, mit der bestimmte Stellen des Infrastrukturgraphen besucht wurden. Je öfters eine Stelle besucht wurde, desto heller erscheint sie. Anhand der Besuchswahrscheinlichkeitskarten (s. Abbildungen 5.5 und 5.6) sieht man, dass sich die Bewegungen der Ausflügler auf die Stadtmitte konzentrieren, während sich die Bewegungen der Studenten eher auf das ganze Gebiet verteilen.

Abbildung 5.4: Liniennetz

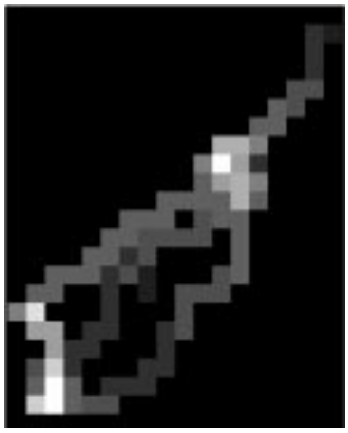

Abbildung 5.5: Student

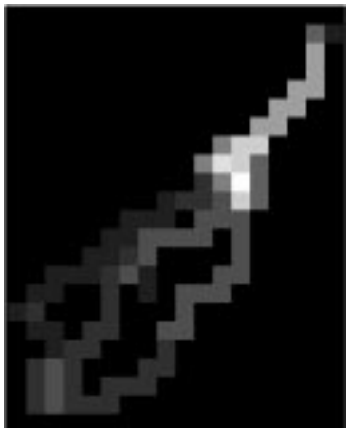

Abbildung 5.6: Ausflügler

Bei der zweiten und dritten Simulationsreihe werden die abgefragten Informationseinheiten nebeneinander gestellt. Die Abbildungen 5.7 und 5.8 zeigen, wie oft jede Informationseinheit abgefragt wurde[2]. Die Verteilung hat auf die Haltestellen, die nicht bevorzugt angefahren werden ("non-dedicated"), keinen Einfluss, da dort immer nur *eine* Informationseinheit abgefragt wird. Bei den anderen Haltestellen ("dedicated") lässt sich ein unterschiedliches Muster erkennen: bei der dritten Simulationsreihe ("Zipf"-Verteilung) werden diejenigen Informationseinheiten eines Ortes mit den niedrigeren IDs häufiger abgefragt als die mit den höheren.

[2] nicht vergebene IDs werden nicht angezeigt

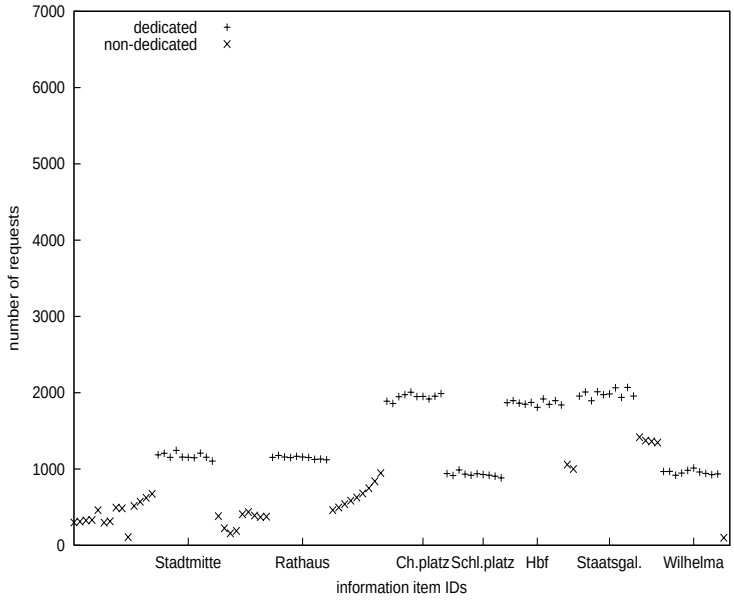

Abbildung 5.7: Gleichverteilung

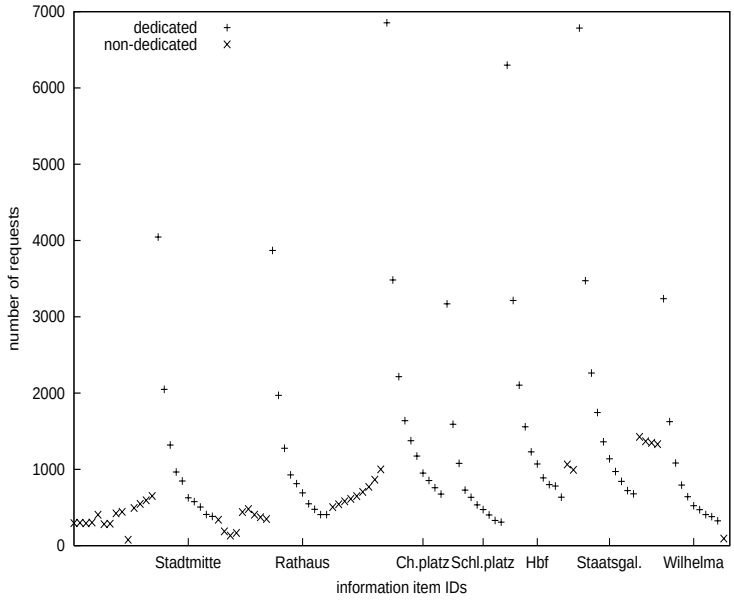

Abbildung 5.8: “Zipf”-Verteilung

Kapitel 6

Die Simulationsumgebung

6.1 Start der Applikation

Da das Programm mit der Programmiersprache *Java* entwickelt wurde, erfolgt seine Ausführung mit einem Java-Interpreter. Vorausgesetzt wird eine fehlerfrei verlaufene Kompilation der Quelltexte (s. Kapitel 7) und das Vorhandensein eines ordnungsgemäß installierten Java-Interpreters Im Folgenden sei `$CLASSES` das Verzeichnis, in dem sich das Verzeichnis `senv`, also das übergeordnete *Package* mit den Klassendateien befindet. Das Starten des Programmes erfolgt mit dem Kommandozeilen-Befehl

```
java -classpath $CLASSES senv.control.SEnv
```

Weitere Einstellungen für die *Java Virtual Machine (JVM)* haben sich - zumindest unter den Betriebssystemen *SunOS 5.7* und *Windows 95* - als nicht nötig erwiesen. Lediglich bei der Verwendung von Verzeichnis-"Separatoren" ("/" unter Unix und Linux, "\" unter Windows 95) unterscheiden sie sich.

Das Programm unterstützt drei optionale Kommandozeilen-Parameter:

-c(onfig) <file> Mit `-c`, bzw. `-config` kann eine zu verwendende Konfigurationsdatei angegeben werden. <`file`> ist ein Dateiname (mit Verzeichnis, wenn nötig). Das Format der Konfigurationsdatei wird in Abschnitt 6.13.1 beschrieben.

-d(ebug) Dieser Parameter ist nur für den Entwickler des Programmes nötig; bei unerwartetem oder fehlerhaftem Verhalten der Simulationsumgebung können Informationen über den Programmfluß und Variablenwerte ausgegeben werden. Sie beeinträchtigen die Funktionalität des Programmes nicht, verlangsamen es aber.

-h(elp) Fasst nochmal die möglichen Parameter zusammen (s. Abbildung 6.1), beendet dann aber sofort das Programm.

```
Usage:
java [JVM parameters] senv.control.SENV [options]

Options:
-c(onfig) <file> takes <file> as configuration file, default: SEnv.config
-d(ebug) turns debug mode on, default: off
-h(elp) prints this message
```

Abbildung 6.1: Hilfetext bei Kommandozeilen-Parameter "-h(elp)"

6.2 Hauptfenster

Nach dem Start des Programmes erscheint das *Hauptfenster*. Es untergliedert sich in fünf Bereiche (s. Abbildung 6.2), die in den folgenden Abschnitten erklärt werden:

(1) die Menüleiste (s. Abschnitt 6.3)

(2) das Submodell-Fenster (s. Abschnitt 6.4)

(3) das Infrastrukturgraph-Fenster (s. Abschnitt 6.5)

(4) das Simulationslauf-Fenster (s. Abschnitt 6.6)

(5) das Systeminformation-Fenster (s. Abschnitt 6.7)

6.3 Menüleiste

Die Menüleiste (s. Abbildung 6.3) erstreckt sich - wie üblich - horizontal über die gesamte Breite des Hauptfensters. Sie enthält die Titel der "Drop-down"-Menüs:

- File
- Graph

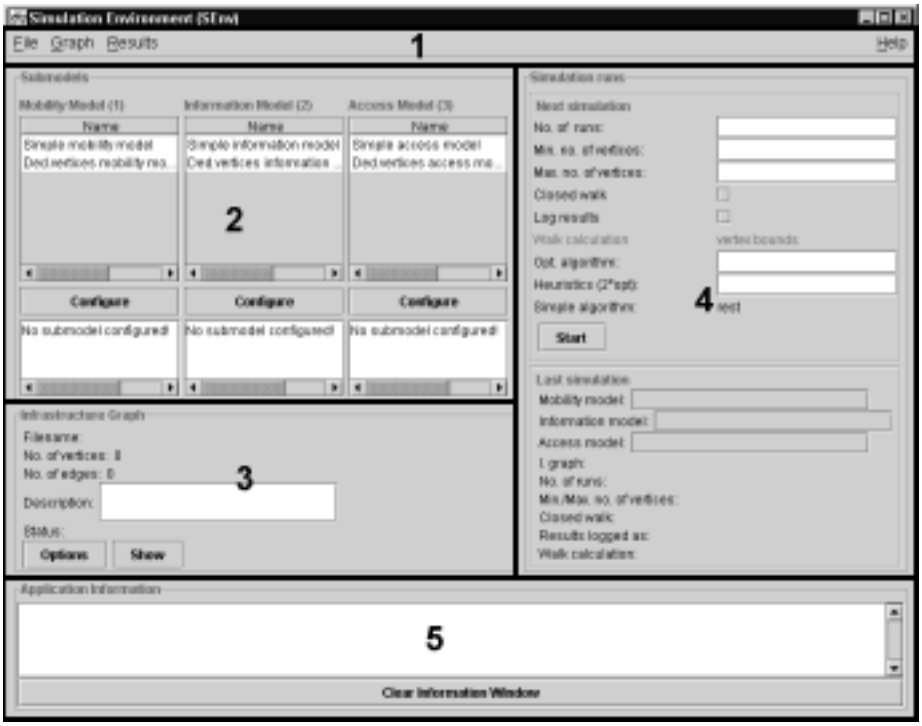

Abbildung 6.2: Hauptfenster

Abbildung 6.3: Die Menüleiste

- Results
- Help

Diese "Drop-down"-Menüs wiederum setzen sich zusammen aus Menüeinträgen. Die folgenden Abschnitte beschreiben die "Drop-down"-Menüs und ihre Einträge. Den meisten Menüeinträgen ist ein Tastatur-"Shortcut" zugeordnet, der die entsprechende Funktion ohne Umweg über die Menüleiste ausführt.

6.3.1 Menü *File*

Dieses Menü (s. Abbildung 6.4) enthält Einträge bezüglich Dateien und der gesamten Applikation:

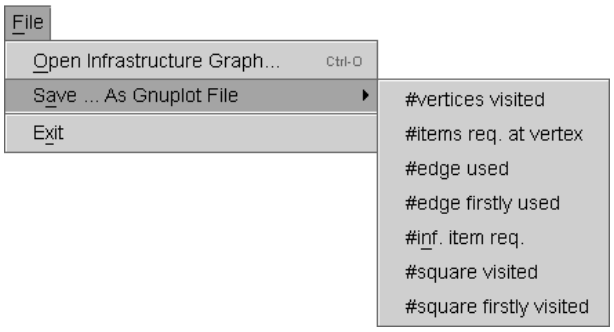

Abbildung 6.4: Menü *File*

Open Infrastructure Graph... (Ctrl-O) Öffnet ein Dateiauswahl-Fenster (s. Abbildung 6.5) und ermöglicht die Selektion und anschließende Öffnung einer Infrastrukturgraph-Datei (s. Abschnitt 6.13.3). Das System kann immer nur einen Infrastrukturgraphen verwalten. Mit dem Laden eines neuen Infrastrukturgraphen gehen der alte Infrastrukturgraph und die daraus gezogenen Simulationsergebnisse verloren (s. Abbildung 6.6).

Save ... As Gnuplot File Speichert die Resultate der zuletzt durchgeführten Simulationsreihe in einem Dateiformat, das vom externen Programm *Gnuplot* verstanden wird und visualisiert werden kann (s. Abschnitt 6.13.5). Ein Submenü gibt Auskunft darüber, welches der Simulationsresultate gespeichert werden soll:

 # vertices visited Wie oft wurde ein Knoten während einer Simulationsreihe besucht?

 # items req. at vertex Wieviele Informationseinheiten wurden an einem Knoten insgesamt abgefragt?

 # edge used Wie oft wurde eine Kante insgesamt während einer Simulationsreihe benutzt?

 # edge firstly used Wie häufig wurde eine Kante bei einem Simulationslauf erstmalig während einer Simulationsreihe benutzt?

 # inf. item req. Wie oft wurde eine Informationseinheit abgerufen?

 # square visited In wie vielen Fällen wurde ein Besuchsfeld insgesamt betreten?

 # square firstly visited Wie oft wurde während einer Simulationsreihe ein Besuchsfeld bei einem Simulationslauf erstmals betreten?

In Abschnitt 6.12 wird gezeigt, wie die Simulationsergebnisse dargestellt und ohne Umweg über das Menü abgespeichert werden können.

Exit Beendet die Applikation.

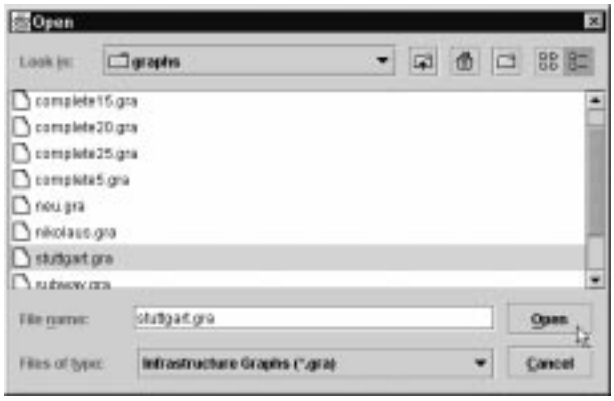

Abbildung 6.5: Laden eines Infrastrukturgraphen

6.3.2 Menü *Graph*

In diesem Menü kann die Anzeige des Infrastrukturgraphen beeinflusst werden:

Options (Ctrl-P) Öffnet ein Fenster, in dem Einstellungen über die graphische Darstellung des Infrastrukturgraphen gemacht werden können (s. Abschnitt 6.9).

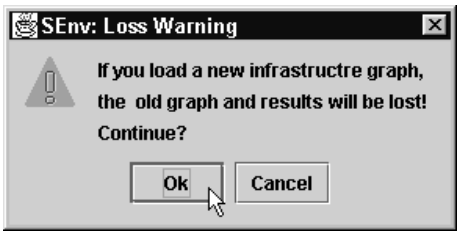

Abbildung 6.6: Warnung: Verlust der Simulationsergebnisse

Graphical Representation (Ctrl-G) Öffnet ein Fenster, in der Infrastrukturgraph graphisch gezeigt wird (s. Abschnitt 6.10). Im selben Fenster wird auch die *Besuchswahrscheinlichkeitskarte* dargestellt.

Textual Representation (Ctrl-T) Öffnet ein Fenster, in dem der Infrastrukturgraph textuell gezeigt wird (s. Abschnitt 6.11).

Abbildung 6.7: Menü *Graph*

6.3.3 Menü *Results*

Hier kann sich der Benutzer die Ergebnisse der Simulationsläufe anzeigen lassen:

Options Öffnet ein Fenster, in dem Einstellungen über die graphische Darstellung der Besuchswahrscheinlichkeitskarte getroffen werden können (s. Abschnitt 6.9).

Visit Probability Map (Ctrl-V) Öffnet ein Fenster, in dem die Besuchswahrscheinlichkeitskarte (s. Abschnitt 6.10) dargestellt wird. In demselben Fenster wird auch der Infrastrukturgraph graphisch dargestellt.

Textual Results Öffnet ein Fenster mit Angaben über Besuchshäufigkeit pro Knoten und Kante, Anzahl der abgefragten Informationseinheiten pro Knoten, Abfragehäufigkeit der Informationseinheiten und Besuchshäufigkeit der Felder (s. Abschnitt 6.12).

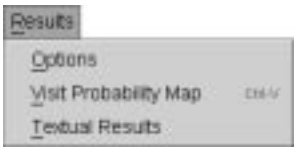

Abbildung 6.8: Menü *Results*

6.3.4 Menü *Help*

About Öffnet ein Fenster mit Informationen über die Applikation (s. Abbildung 6.9).

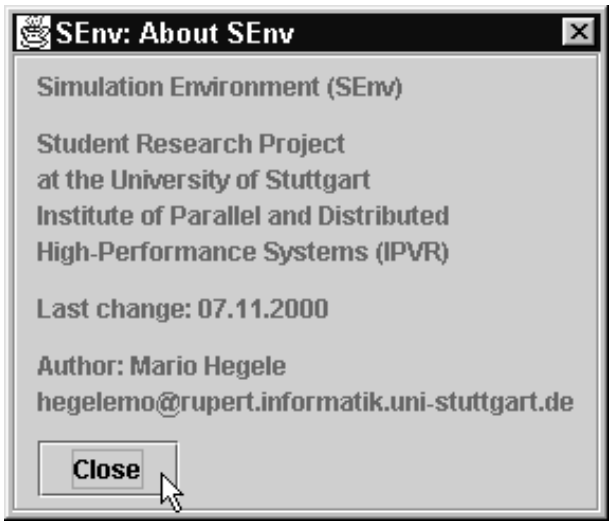

Abbildung 6.9: "About"-Fenster

6.4 Submodell-Fenster

Das Submodell-Fenster nimmt den größten Teil des Hauptfensters ein. Es unterteilt sich in drei Bereiche (für jeden Submodelltyp), die alle dieselbe Bedienung unterstützen und ähnlich aufgebaut sind (s. Abbildung 6.10). In der Auswahlliste kann eine Instanz des jeweiligen Submodells ausgesucht werden, mit der die nächste Simulationsreihe durchgeführt werden soll. Neben dem Namen ("Name") der Submodell-Instanzen stehen die Abhängigkeiten ("Dep.") von anderen Submodellen und die maximale Knotenanzahl ("Vertices"). Z.B. würde "-23" in der Spalte "Dep." bei einer Submodell-Instanz bedeuten, dass sie vom Informationsmodell (2) und vom Zugriffsmodell (3) abhängt. Die

Durchführung einer Simulationsreihe ist nicht möglich, wenn die gewählten Submodelle zyklisch voneinander abhängig sind. Ein Submodell kann für alle ("all") Knoten eines Infrastrukturgraphen definiert sein oder nur für einige.

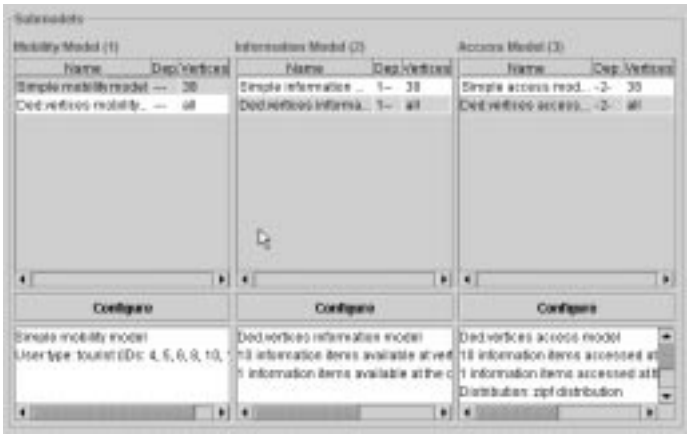

Abbildung 6.10: Submodell-Fenster

Wenn eine Instanz in der Liste zum ersten Mal angeklickt wird oder der "Configure"-Knopf gedrückt wird, dann erfolgt die Konfiguration der Submodell-Instanz. Es hängt sehr von der gewählten Instanz ab, wie die Konfiguration vor sich geht. Beispielsweise wird bei dem momentan bereits eingebundenen einfachen Submodell "Simple mobility model" die Konfiguration per Zufall zwischen den Benutzerprofilen "tourist" und "shopper" vorgenommen. Beim erweiterten Submodell "Ded.vertices mobility model" hingegen öffnet sich bei der Konfiguration ein Fenster, in dem der Simulationsbenutzer Einstellungen vornehmen kann (s. Abschnitt 5.1).

Das Ergebnis der Konfiguration kann dem Textfenster unterhalb des "Configure"-Knopfes entnommen werden. Der Text "No submodel configured!" weist auf einen abgebrochenen oder noch nicht vorgenommenen Konfigurationsvorgang hin. Nach einer Simulationsreihe erscheint dieser Text auch im unteren Teil ("Last simulation") des Simulationslauf-Fenster (s. Abschnitt 6.6) und in der Log-Datei (s. Abschnitt 6.13.4), damit nachvollzogen werden kann, mit welchen Einstellungen die Simulation durchgeführt wurde.

6.5 Infrastrukturgraph-Fenster

Das Infrastrukturgraph-Fenster (s. Abbildung 6.11) zeigt Informationen über den momentan geladenen Infrastrukturgraphen an, d.h. auf diesem Infrastrukturgraphen wird die nächste Simulationsreihe durchgeführt. Zu den Informationen gehört:

- Name der Datei, die den Infrastrukturgraphen enthält
- Anzahl der Knoten
- Anzahl der Kanten
- eine Beschreibung (s. Abschnitt 6.13.3 "description")
- Status: "appropriate" oder "not appropriate"

Abbildung 6.11: Infrastrukturgraph-Fenster

Der Status des Infrastrukturgraphen gibt an, ob er geeignet ("appropriate") ist, auf ihm eine Simulation auszuführen, d.h. der Graph *zusammenhängend* ist und mindestens einen Knoten enthält.

Außerdem werden zwei Knöpfe angeboten, die mit der graphischen Anzeige des Infrastrukturgraphen zusammenhängen und deren Funktionalität auch über die Menüleiste erreicht werden kann: der Knopf "Options" (Funktion wie Menüeintrag "Graph/Options") und der Knopf "Show" (Funktion wie Menüeintrag "Graph/Graphical Representation"), also die Anzeige des Optionen-Fensters (s. Abschnitt 6.9) und des Infrastrukturgraphen (s. Abschnitt 6.10).

6.6 Simulationslauf-Fenster

Dem Simulationslauf-Fenster (s. Abbildung 6.12) kann entnommen werden, mit welchen Einstellungen die *nächste* Simulationsreihe ("Next simulation") durchgeführt wird und wie die *letzte* Simulationsreihe ("Last simulation") stattgefunden hat.

Folgende Parameter können eingegeben werden:

- Aus wieviel Simulationsläufen die Simulationsreihe bestehen soll ("No. of runs").
- Wieviele Zielknoten *mindestens* besucht werden sollen ("Min. no. of vertices").

Abbildung 6.12: Simulationslauf-Fenster

- Wieviele Zielknoten *höchstens* besucht werden sollen ("Max. no. of vertices").
- Ob der Weg über die Knoten *geschlossen* sein soll ("Closed walk").
- Ob die Simulationsereignisse in einer Datei protokolliert werden sollen ("Log results", s. Abschnitt 6.13.4).
- Bis zu wieviel Zielknoten der Enumerationsalgorithmus verwendet werden soll ("Opt. algorithm").
- Bis zu wieviel Zielknoten eine Heuristik (Minimum Spanning Tree) verwendet werden soll ("Heuristics (2*opt)").

Mit dem "Start"-Knopf wird die Simulation begonnen (s. Abschnitt 6.8). *Nach* der Simulation erscheinen alle im oberen Teil des Fensters getroffenen Einstellungen im unteren Teil, damit der Benutzer nachvollziehen kann, wie die letzten Simulationsergebnisse zustande kamen.

6.7 Systeminformation-Fenster

Das Systeminformation-Fenster (s. Abbildung 6.13) protokolliert Ereignisse, die während des Programmablaufes auftreten:

- das Laden der Infrastrukturgraph-Datei (s. Abschnitt 6.13.3),

- das Speichern der Optionen-Datei (s. Abschnitt 6.13.2),
- die automatische Korrektur bei falsch ausgefüllten Eingabefeldern (s. Abschnitt 6.8),
- das Speichern der Simulationsergebnisse (s. Abschnitte 6.13.4 und
- das Beenden der Simulationsreihe

Abbildung 6.13: Systeminformation-Fenster

Die neu hinzukommenden Zeilen fügt das System unten an. Der Benutzer kann die Ausgabezeilen per Knopfdruck ("Clear Information Window") vollständig löschen. Außerdem führt das System selbständig eine Löschung der ältesten Zeilen durch, so dass die Anzahl der ausgegebenen Zeilen 100 nie übersteigt.

6.8 Durchführung der Simulation

6.8.1 Vor der Simulation

Wenn eine neue Simulation durchgeführt wird, dann gehen die alten Simulationsergebnisse verloren. Daran wird der Benutzer vor der Simulation mit einer Warnung erinnert (s. Abbildung 6.14). Er hat dann die Möglichkeit, die Simulation abzubrechen, um die alten Simulationsergebnisse zu sichern.

Abbildung 6.14: Warnung: Verlust der Simulationsergebnisse

Um überhaupt eine Simulation ablaufen zu lassen, müssen bestimmte Voraussetzungen erfüllt sein, auf deren Fehlen graphisch hingewiesen wird (in dieser Reihenfolge):

Voraussetzung	**Hinweis**
Ein geeigneter Infrastrukturgraph (zusammenhängend, mindestens ein Knoten) muss ausgewählt sein.	"You must choose an appropriate infrastructure graph"
Für jeden Submodelltyp muss eine Instanz ausgewählt und konfiguriert sein.	"You must choose the mobility/ information/access model(s)!"
Die Anzahl der zu simulierenden Läufe muss mindestens "1" betragen	"The number of simulation runs must be a legal number greater than 0!"
Die Abhängigkeiten der Submodell dürfen nicht zyklisch sein	"The submodel dependencies cannot be resolved!"
Die drei gewählten Submodell-Instanzen sollten mindestens für soviele Knoten definiert sein, wie der Infrastrukturgraph Knoten besitzt	"The mobility/information/access model(s) are not defined for some vertices!"

Des weiteren können im Simulationslauf-Fenster in vier Textfeldern Einstellungen getroffen werden, die automatisch korrigiert werden, wenn der Wert nicht in einem sinnvollen Wertebereich liegt. Eine erfolgte Korrektur kann im *Systeminformation-Fenster* (s. Abschnitt 6.7) nachgelesen werden.

Textfeld	**Wertebereich**	**automatische Korrektur zu**
"Min. no. of vertices" *min*	0 <= min <= #vertices	min := 0
"Max. no. of vertices" *max*	0 <= max <= #vertices	max := #vertices
"Opt. algorithm" *opt*	opt <= #vertices 0 <= opt	opt := #vertices opt := 0
"Heuristics (2*opt)" *heu*	heu <= #vertices 0 <= heu	heu := #vertices heu := 0

#vertices : Anzahl der Knoten des Infrastrukturgraphen

6.8.2 Während der Simulation

Wenn eine Simulation läuft, dann erscheint ein Fenster mit dem Titel "SEnv: Running Simulation" (s. Abbildung 6.15) im Vordergrund. Das Fenster zeigt den aktuellen Arbeitsschritt ("Status:...”), wieviele Simulationsläufe ausgeführt werden sollen und wieviel Prozent davon bereits bewältigt wurden. Während einer Simulation können keine anderen Benutzer-Interaktionen mit Programm erfolgen, außer dem Stoppen der

Simulation (s. Abbildung 6.16). Einzige Ausnahme bildet die Problematik, dass das Mobilitätsmodell eine Menge von Knoten finden muss, die mit den Zielknotengrenzen vereinbar ist. Wenn dieses Problem auftritt (s. Abbildung 6.17), muss der Benutzer entscheiden, ob er die Simulation abbrechen will, um eventuell andere Einstellungen vorzunehmen, oder ob das Programm weiter nach einer Lösung suchen soll.

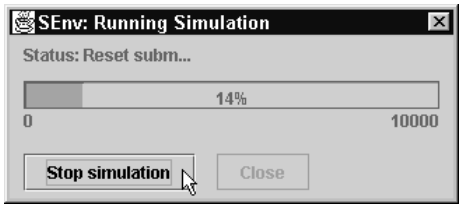

Abbildung 6.15: Laufende Simulation

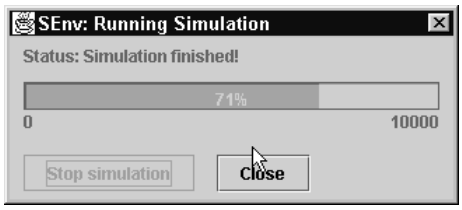

Abbildung 6.16: Gestoppte Simulation

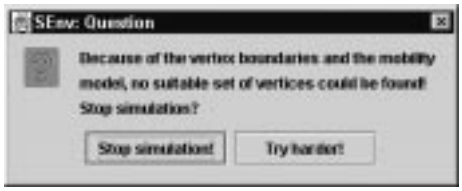

Abbildung 6.17: Mobilitatsmodell/Zielknoten-Problem

Verläuft die Durchführung der Simulationsreihe "normal", dann endet sie mit dem Abschluss des letzten Simulationslaufes (s. Abbildung 6.16). Der einzige Unterschied zum benutzerinitiierten Stoppen der Simulation liegt in der vollzähligen Abarbeitung *aller* Simulationsläufe.

6.8.3 Nach der Simulation

Zur Ermittlung der Simulationsergebnisse werden nur die vollständig durchgeführten Simulationsläufe herangezogen. Die genaue Anzahl kann dem *Simulationslauf-Fenster* (s. Abbildung 6.6, "No. of runs:") entnommen werden. Die Simulationsereignisse können folgendermaßen abgerufen werden:

- Mit einem Texteditor kann die Log-Datei (s. Abschnitt 6.13.4) betrachtet werden.
- Die Simulationsergebnisse werden textuell in einem eigenen Fenster dargestellt (s. Abschnitt 6.12).
- Die Simulationsergebnisse können in Dateien abgespeichert (s. Abschnitt 6.13.5) und mit dem externen Programm "Gnuplot" z.B. als Balkengrafik betrachtet werden.
- An der Besuchswahrscheinlichkeitskarte (s. Abschnitt 6.10) können die Bewegungen des Benutzers auf dem Infrastrukturgraphen nachvollzogen werden.

6.9 Optionen-Fenster

Das Optionen-Fenster (s. Abbildungen 6.18, 6.19) öffnet sich, wenn der Benutzer den Menüeintrag "Graph/Options" wählt oder per Menüeintrag "Results/Options". Je nachdem, wie das Fenster geöffnet wurde, erscheint die entsprechende Karteikarte. Hier macht der Benutzer Einstellungen über die graphische Darstellung des Infrastrukturgraphen und der Besuchswahrscheinlichkeitskarte (s. Abschnitt 6.10), deren Anzeige miteinander verknüpft ist:

Infrastrukturgraph

Vertex Size Eingabefeld für die Größe der Knoten (in *pixel*)

Vertex Color Auswahlbox (rot, gelb, grün, blau, schwarz, weiß) für die Farbe der Knoten

Edge Width Eingabefeld für die Breite der Kanten (in *pixel*)

Edge Color Auswahlbox (rot, gelb, grün, blau, schwarz, weiß) für die Farbe der Kanten

Position of Vertex ID Auswahlbox (links, rechts, über, unter) für die Position der Beschriftung (die ID) der Knoten

Position of Vertex description Auswahlbox (links, rechts, über, unter) über die Position der Beschriftung (die Beschreibung) der Knoten

Font Size Eingabefeld für die Größe der Schrift

Font Color Auswahlbox (rot, gelb, grün, blau, schwarz, weiß) für die Farbe der Schrift

Show Vertex ID Checkbox, die bestimmt, ob die ID der Knoten angezeigt werden soll

Show Vertex Description Checkbox, die bestimmt, ob die Beschreibung der Knoten angezeigt werden soll

Scale Eingabefeld für die Skalierung des Infrastrukturgraph-Fensters in Prozent, d.h. hiermit kann das Fenster verkleinert, bzw. vergrößert werden. Ausgegangen wird von einer fest im Programm verankerten Maximalgröße (z.B. 200 Pixel) für die Breite/die Höhe des Fensters. Die Größe der anderen Dimension richtet sich je nach Form des Infrastrukturgraphen danach aus. Diese Anfangsgröße des Fensters wird als 100% betrachtet. Die anderen Einstellungen bleiben davon unbeeinflusst.

Besuchswahrscheinlichkeitskarte

Vertical Grid Width Eingabefeld für den vertikalen Abstand der Gitterlinien (in *pixel*).

Horizontal Grid Width Eingabefeld für den horizontalen Abstand der Gitterlinien (in *pixel*).

Color Steps Eingabefeld für die Anzahl der Farbabstufungen, mit denen die durch die Gitterlinien entstandenen Felder eingefärbt werden sollen.

Scale siehe oben

Die Ausgangswerte werden bei Start des Programmes der Optionen-Datei (s. Abschnitt 6.13.2) entnommen. Für jeden Eintrag gibt es Default-Werte.

Zusätzlich zu den Einstellungsmöglichkeiten gibt es im unteren Bereich des Fensters noch vier Knöpfe:

OK Übernimmt die Einstellungen, wendet sie an, falls das Infrastrukturgraph-Fenster geöffnet ist, und schließt das Optionen-Fenster.

Cancel Verwirft die Einstellungen, die seit dem letzten Drücken von "OK", "Apply" oder "Save" (muss nicht stattgefunden haben) gemacht worden sind und schließt das Optionen-Fenster.

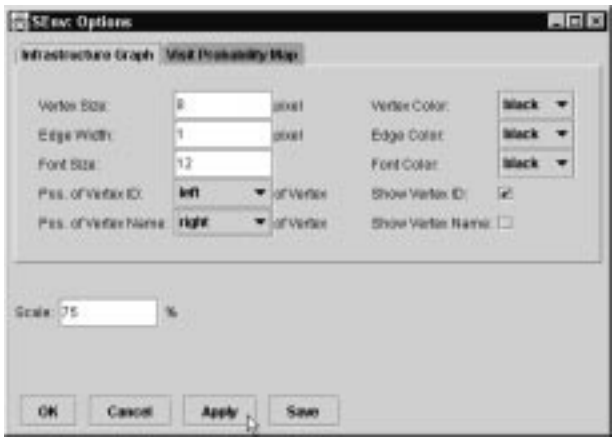

Abbildung 6.18: Optionen-Fenster (Infrastrukturgraph)

Abbildung 6.19: Optionen-Fenster (Besuchswahrscheinlichkeitskarte)

Apply Übernimmt die Einstellungen und wendet sie an, falls das Infrastrukturgraph-Fenster geöffnet ist.

Save Speichert die Einstellungen in einer Datei (s. Abschnitt 6.13.2) und verwendet sie beim nächsten Start des Programmes.

Die Voreinstellungen der Werte hängen davon ab, ob beim Start der Applikation die Optionen-Datei (s. Abschnitt 6.13.2) ordnungsgemäß geladen werden konnte.

6.10 Infrastrukturgraph und Besuchswahrscheinlichkeitskarte

In diesem Fenster werden der Infrastrukturgraph und die Besuchswahrscheinlichkeitskarte und verschiedene Zwischenformen angezeigt. Geöffnet wird das Fenster (s. Abbildung 6.20) über den Menüeintrag "Graph/Graphical Representation" bzw. "Results/Visit Probability Map". Das Fenster zeigt für den Infrastrukturgraphen und die Besuchswahrscheinlichkeitskarte dasselbe Gebiet. Über Radio-Buttons kann über die Art der Darstellung entschieden werden:

Infrastructure Graph Darstellung des Infrastrukturgraphen: Knoten als Kreise, Kanten als Linien. Weitere Einzelheiten können über das Optionen-Fenster (s. Abschnitt 6.9) festgelegt werden.

I.Graph+Grid Darstellung wie "Infrastructure Graph". *Darüber* wird das Gitter für die Besuchswahrscheinlichkeitskarte gelegt (d.h. nur die Linien).

Vis.Prob.Map+I.Graph Darstellung der Besuchswahrscheinlichkeitskarte, d.h. eingefärbter Rechtecke ohne Gitterlinien. *Darüber* wird der Infrastrukturgraph gelegt.

Visit Prob. Map Darstellung der Besuchswahrscheinlichkeitskarte entsprechend den Einstellungen des Optionen-Fensters.

Eine Ankreuzbox ("show only first square visits") lässt die Wahl zwischen zwei Varianten der Besuchswahrscheinlichkeitskarte (s.u.).

Die Darstellungsform beim Öffnen des Fensters entscheidet sich danach, über welche Menüpunkte das Fenster geöffnet wurde.

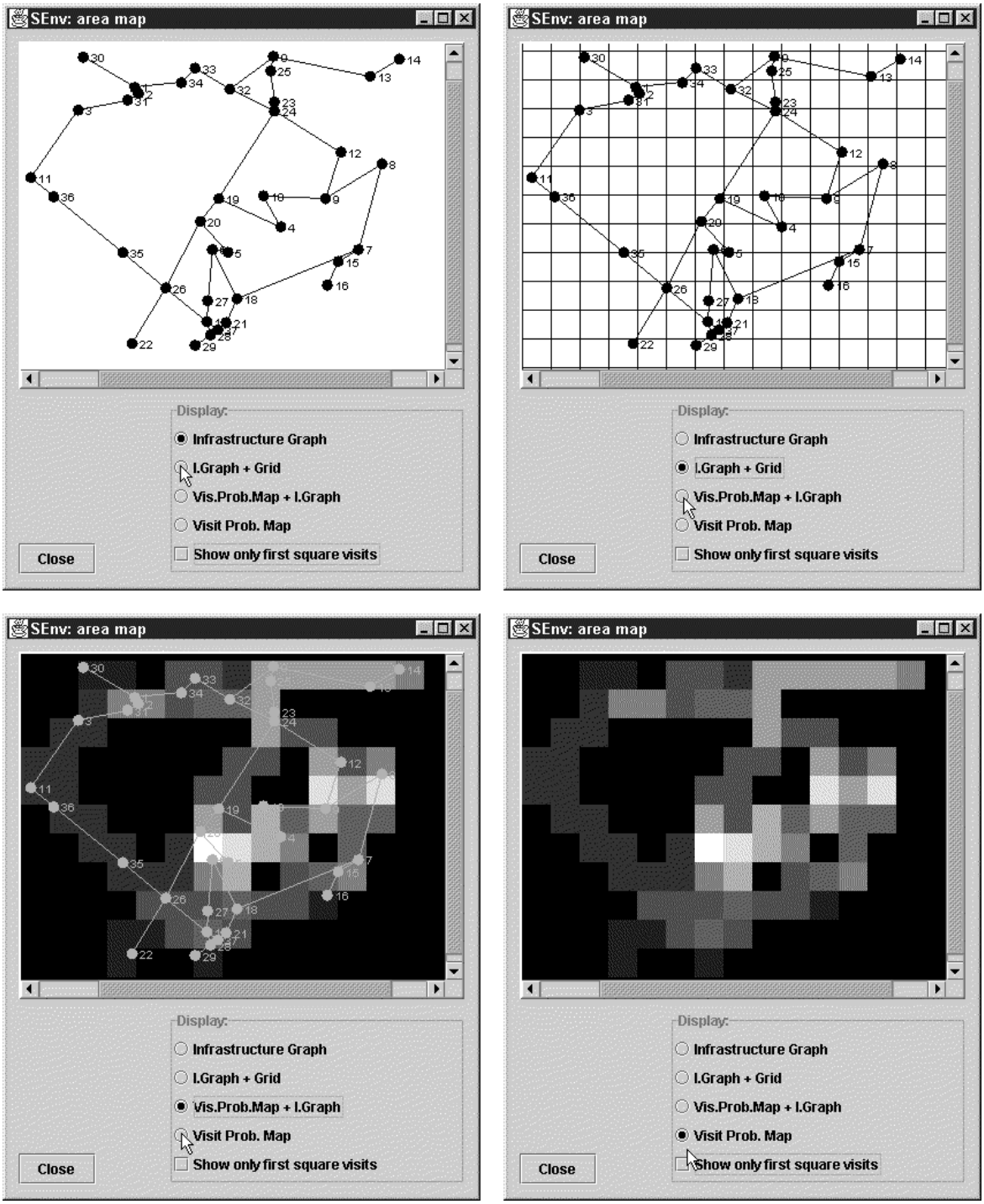

Abbildung 6.20: Darstellung von Infrastrukturgraph und Besuchswahrscheinlichkeitskarte

6.10.1 Einfärbung der Besuchswahrscheinlichkeitskarte

Die durch die Gitterlinien entstehenden Rechtecke werden im Folgenden *Besuchsfelder* (oder kurz: *Felder*) genannt. Die Größe und Position der Besuchsfelder der Besuchswahrscheinlichkeitskarte hängen von den in den Optionen (s. Abschnitt 6.9) getroffenen Einstellungen ab. In der oberen linken Ecke liegt das erste Besuchsfeld, die folgenden richten sich danach aus. Jeder Punkt im dargestellten Gebiet kann eindeutig *einem* Besuchsfeld zugeordnet werden. Punkte, die scheinbar *auf* einer Gitterlinie liegen, gehören zum rechten, bzw. unteren Besuchsfeld.

Für die Besuchswahrscheinlichkeitskarte werden die simulierten Bewegungen des Benutzers eines mobilen Informationssystems von den Kanten des Infrastrukturgraphen auf die Besuchsfelder übertragen. Dabei wird angenommen, dass der Benutzer sich stets nur vertikal oder horizontal von Feld zu Feld fortbewegt, aber nie diagonal (s. Abbildung 6.21). Wenn während der Benutzung einer Kante ein Feld betreten wird, dann gilt das als *Besuch* des Feldes.

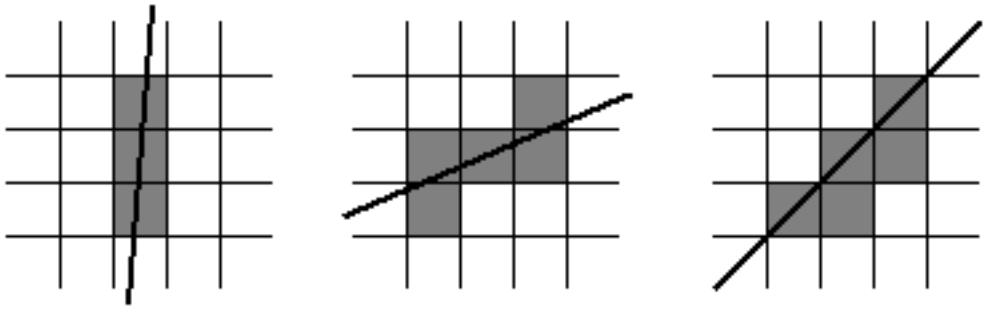

Abbildung 6.21: Von der Kantenbenutzung zum Feldbesuch

Es gibt zwei Varianten der Besuchswahrscheinlichkeitskarte: bei der einen werden *alle* Besuche während einer Simulationsreihe gezählt. Bei der zweiten Variante (Box "show only first square visits" ist angekreuzt) wird der Zähler für ein Besuchsfeld nur dann um eins erhöht, wenn das Feld zum *ersten* Mal bei einem Simulationslauf besucht wird. Alle weiteren Besuche während eines Simulationslaufes werden ignoriert. Damit wird festgestellt, bei wievielen Simulationsläufen das Feld überhaupt besucht wurde. Der Status der Ankreuzbox entscheidet darüber, welche der beiden Varianten angezeigt wird.

Die Einfärbung eines Feldes wird davon bestimmt, wie oft ein Besuch gezählt wurde, wie häufig das am meisten besuchte Feld betreten wurde und wieviele Farbabstufungen zwischen schwarz (nie besuchtes Feld) und weiß (am häufigsten besucht) gemacht werden sollen (s. Abschnitt 6.9, "color steps"). Abbildung 6.22 zeigt die Möglichkeiten,

wie ein Feld eingefärbt sein kann, wenn es fünf Farbabstufungen (ohne schwarz) gibt und das am meisten frequentierte Feld 120 mal besucht wurde.

interval	(0-0)	(1-24)	(25-48)	(49-72)	(73-96)	(97-120)
color						

Abbildung 6.22: Einfärbung von Feldern

6.11 Textuelle Darstellung des Infrastrukturgraphen

Da die genauen Knotenkoordinaten und Kantenlängen nicht aus der graphischen Darstellung entnommen werden können, kann der Benutzer den Infrastrukturgraphen alternativ auch textuell betrachten. Es gibt je eine Karteikarte für die Knoten (s. Abbildung 6.23) und die Kanten (s. Abbildung 6.24).

SEmv: Infrastructure Graph

Vertices Edges

ID	Description	X	Y	Neighbours
0	Engelboldstr.	27.2	9.2	1,5
1	Vaihingen Viadukt	26.4	8.8	0,2
2	Faustistr.	26.2	8.6	1,3
3	Vaihingen Schillerplatz	26.2	8.4	2,4
4	Vaihingen Bf	26.2	8.0	3,18,19,32
5	Kaltental	27.4	9.4	0,6
6	Waldeck	27.6	10.0	5,7
7	Heslach Vogelrain	27.8	10.8	6,8,9
8	Suedheimer Platz	28.0	11.0	7,10
9	Waldfriedhof	28.4	10.6	7

Close

Abbildung 6.23: Textuelle Darstellung der Infrastrukturgraph-Knoten

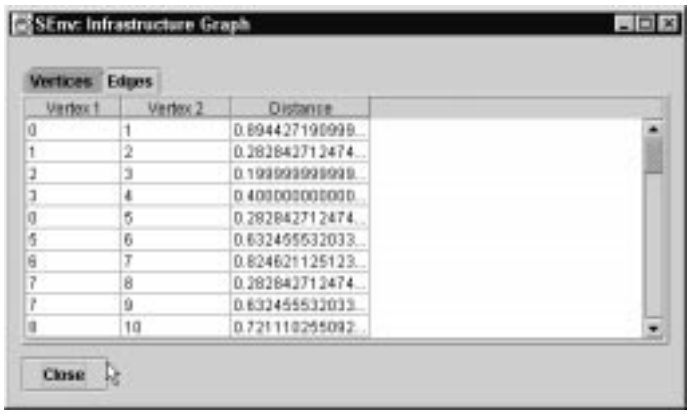

Abbildung 6.24: Textuelle Darstellung der Infrastrukturgraph-Kanten

6.12 Textuelle Darstellung der Simulationsergebnisse

In vier Kategorien werden bei den Simulationsläufen *Zähler* mitgeführt:

- Für die *Knoten* des Infrastrukturgraphen,
- für die *Kanten* des Infrastrukturgraphen,
- für die während der Simulation abgefragten *Informationseinheiten* und
- für die *Besuchsfelder*, die das vom Infrastrukturgraphen abgedeckte Gebiet unterteilen.

Für jede der Kategorien gibt es bei der Darstellung der Simulationsergebnisse eine Karteikarte ("Vertices", "Edges", "Inf. Items" und "Squares"). Hier können die Simulationsergebnisse auch per Knopdruck in einer Datei (s. Abschnitt 6.13.5) festgehalten werden. Die Beschriftung des Knopfes gibt an, *welches* der Simulationsergebnisse, d.h. welche Spalte abgespeichert wird. Voraussetzung für das Abspeichern ist eine durchgeführte Simulationsreihe. Hat noch kein Simulationslauf stattgefunden, dann enthalten die Tabellen entweder keine Zeilen ("Inf. Items", "Squares"), oder die entsprechenden Felder sind dem Wert "-1" ausgefüllt ("Vertices", "Edges").

6.12.1 Zähler für die Kategorie *Knoten*

Wie der Abbildung 6.25 entnommen werden kann, werden für jeden Knoten ("vertex ID") die Besuchshäufigkeiten ("# visited") und die Anzahl der dort abgefragten

Informationseinheiten ("# inf. items req.") erfasst. Es werden alle Knoten des Infrastrukturgraphen aufgeführt, die IDs ergeben sich aus der Infrastrukturgraph-Datei (s. Abschnitt 6.13.3).

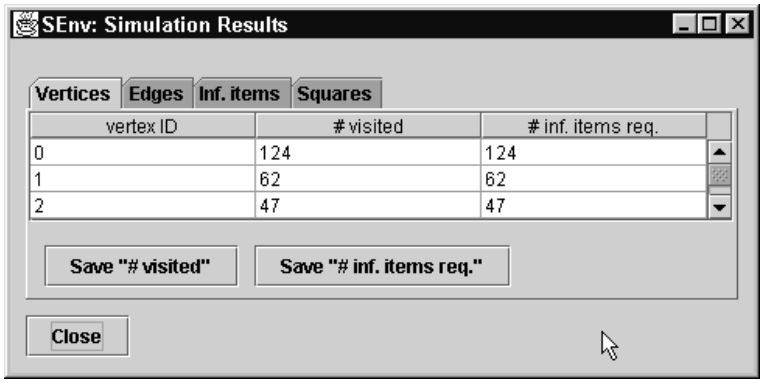

Abbildung 6.25: Zähler für die Knoten

6.12.2 Zähler für die Kategorie *Kante*

Die Tabelle für die Kanten (s. Abbildung 6.26) besitzt fünf Spalten: die ersten drei dienen der Identifizierung der Kante, die letzten beiden enthalten die Zähler. Die zweite und dritte Spalte ("vertex 1", "vertex 2") beschreiben, welche beiden Knoten eine Kante miteinander verbindet. Die ID in der ersten Spalte ("edge ID") ergibt sich aus der bloßen Durchnumerierung der Kanten und tritt in der Simulationsumgebung sonst nicht mehr auf. Sie wird bei der Ausgabe der Simulationsergebnisse als Datei (s. Abschnitt 6.13.5) benötigt. Die Reihenfolge der Kanten spiegelt die Reihenfolge ihres Auftretens in der Infrastrukturgraph-Datei wider.

Der erste Zähler ("# used") bezeichnet, wie oft eine Kante während einer Simulationsreihe *insgesamt* benutzt wurde. Auch der zweite Zähler ("# firstly used") registriert die Benutzung der Kanten. Er erfasst, *ob* während einer Simulationsreihe eine Kante in einem Simulationslauf benutzt wird[1] und addiert diese Kantenbenutzungen für die gesamte Simulationsreihe.

6.12.3 Zähler für die Kategorie *Informationseinheit*

Die Anzahl und IDs ("inf. item ID", s. Abbildung 6.27) der während eines Simulationslaufes abfragbaren Informationseinheiten werden vom Informationsmodell bestimmt. In

[1] gleichgültig ob ein oder mehrere Male

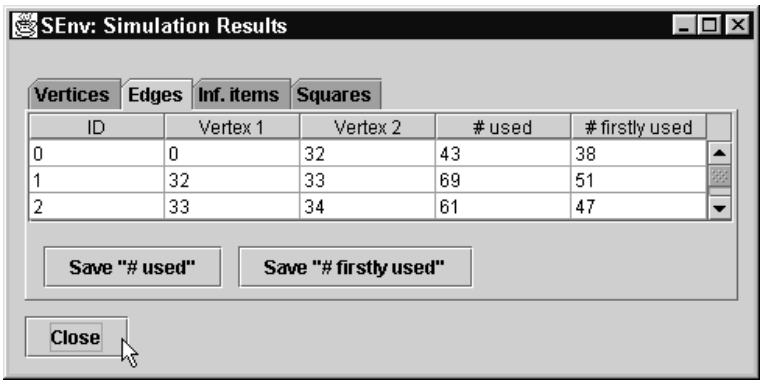

Abbildung 6.26: Zähler für die Kanten

der zweiten Spalte ("# requested") steht, wie oft eine bestimmte Informationseinheit während einer Simulationsreihe abgefragt wurde.

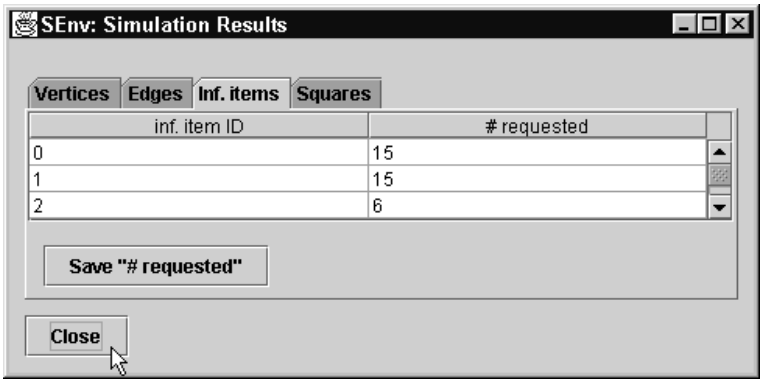

Abbildung 6.27: Zähler für die Informationseinheiten

6.12.4 Zähler für die Kategorie *Besuchsfeld*

Besuchsfelder wurden in Abschnitt 6.10.1 eingeführt. Ihre Anzahl und Größe hängt von den in den Optionen getroffenen Einstellungen (s. Abschnitt 6.9) ab. Je dichter dort der Abstand der vertikalen und horizontalen Gitterlinien eingerichtet wird, desto mehr Besuchsfelder gibt es. Dadurch kann die Darstellung der Simulationsergebnisse - im Gegensatz zu den anderen Kategorien - auch *nach* Durchführung einer Simulationsreihe variiert werden.

Die ersten drei Spalten identifizieren ein bestimmtes Besuchsfeld. Es werden die Spalte ("columns") und die Reihe ("rows") angegeben, in denen sich ein Feld befindet. Die ID eines Besuchsfeldes ("square ID") ist - wie bei den Kanten - nur für die Ausgabe der Simulationsergebnisse als Datei nötig. Spalte vier ("# visited") gibt an, wie oft ein Feld während einer Simulationsreihe betreten wurde. In der fünften Spalte ("# firstly visited") wird - wie bei den Kanten - nur der *erstmalige* Besuch eines Feldes erfasst.

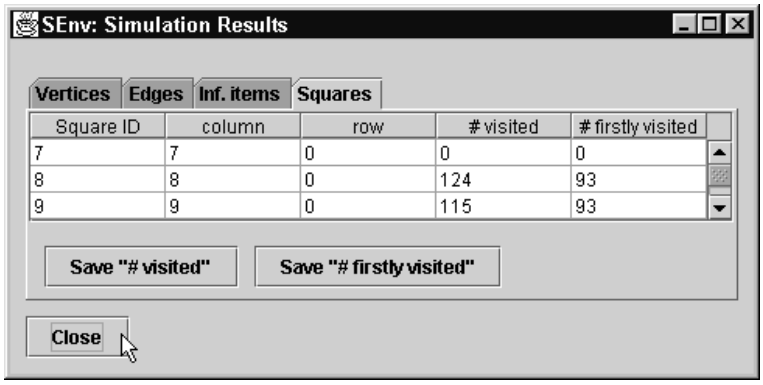

Abbildung 6.28: Zähler für die Besuchsfelder

6.13 Dateiformate

Im Zusammenhang mit der Simulationsumgebung treten fünf Dateitypen auf. Jeweils zwei, die nur geöffnet (Konfigurations-, Infrastrukturgraph-Datei) oder geschrieben (Log-, Resultats-Datei) werden, und eine, die sowohl geöffnet als auch geschrieben (Optionen-Datei) wird:

Konfigurations-Datei Enthält Pfadangaben für die Simulationsumgebung (s. Abschnitt 6.13.1).

Optionen-Datei Bestimmt die Darstellung des Infrastrukturgraphen und der Besuchswahrscheinlichkeitskarte (s. Abschnitt 6.13.2).

Infrastrukturgraph-Datei Beschreibt die Knoten und Kanten, aus denen sich der Infrastrukturgraph zusammensetzt (s. Abschnitt 6.13.3).

Log-Datei Protokolliert Knotenbesuche und Informationsabfragen während einer Simulationsreihe (s. Abschnitt 6.13.4).

Resultats-Datei Enthält Besuchs- und Abfragehäufigkeiten für Knoten, Kanten, Informationseinheiten und Besuchsfelder im *gnuplot*-Format (s. Abschnitt 6.13.5).

6.13.1 Konfigurations-Datei

Die Konfigurations-Datei beinhaltet Pfadangaben für die Applikation. Ihr Vorhandensein ist optional. Der Benutzer kann sie beim Start als Parameter angeben; wenn er das nicht tut, oder die Datei dort nicht gefunden wird, dann wird im aktuellen Arbeitsverzeichnis des Benutzers danach gesucht. Die Datei wird - wenn vorhanden - einmal beim Start des Programmes geöffnet. Es handelt sich um eine Textdatei; es gibt drei Arten von Zeilen:

Kommentare Kommentarzeilen beginnen mit einem Semikolon, das sich an der ersten Position befinden muss. Der weitere Inhalt der Zeile ist beliebig und wird ignoriert.

Zeilen mit Attribut/Werte-Paaren Die Attribute geben an, *welche* Parameter eingestellt werden sollen, und die Werte geben an, *womit*. Die Zeilen haben das Format
ATTRIBUT=wert,
d.h. dieser Ausdruck muss an der ersten Position der Zeile beginnen, es sind keine Leerzeichen erlaubt, Groß- und Kleinschreibung bei dem Attribut ist wichtig.

sonstige Zeilen Das sind z.B. Leerzeilen oder Zeilen, die nicht dem vorgeschriebenen Format genügen und deshalb vom Programm nicht bearbeitet werden können und ignoriert werden.

Es werden folgende Attribute unterstützt:

LOADPATH Gibt die Voreinstellung für den Pfad an, in dem nach den Infrastrukturgraphen gesucht werden soll. Default-Wert: aktuelles Arbeitsverzeichnis

SAVEPATH Gibt die Voreinstellung für den Pfad an, in dem die Simulationsergebnisse abgespeichert werden sollen. Default-Wert: aktuelles Arbeitsverzeichnis

OPTIONPATH Gibt den Pfad an, in dem bei Start des Programmes nach der Optionen-Datei (s. Abschnitt 6.13.2) gesucht werden soll. Default-Wert: aktuelles Arbeitsverzeichnis

LOGPATH Gibt den Pfad an, in dem die Log-Dateien (s. Abschnitt 6.13.4) abgespeichert werden sollen. Default-Wert: aktuelles Arbeitsverzeichnis

Das Vorhandensein und die Reihenfolge der Parameter ist beliebig. Ein Abschluß der Pfadangaben mit dem Schrägstrich "/" (oder "\" je nach Betriebssystem) ist empfehlenswert. Eine typische Konfigurations-Datei wird in Abbildung 6.29 gezeigt.

```
;config file for the simulation application
;author: Mario Hegele
;date: 24.07.2000

;path for ``open infrastructure graph'' window
LOADPATH=/home/hegelemo/SEnv/graphs/

;path for ``save simulation results as...'' window
SAVEPATH=/home/hegelemo/SEnv/results/

;path for options file
OPTIONPATH=/home/hegelemo/SEnv/config/

;path for the log files
LOGPATH=/home/hegelemo/SEnv/log/
```

Abbildung 6.29: Beispiel für eine Konfigurations-Datei

6.13.2 Optionen-Datei

Die Optionen-Datei beinhaltet Einstellungen bezüglich der Darstellung des Infrastrukturgraphen und der Besuchswahrscheinlichkeitskarte. Ihr Name lautet "SEnv.inf", ihr Vorhandensein ist optional. Der Benutzer kann ihre Position in der Konfigurations-Datei angeben; wenn er das nicht tut, oder die Datei dort nicht gefunden wird, dann wird im aktuellen Arbeitsverzeichnis des Benutzers danach gesucht. Die Datei wird - wenn vorhanden - einmal beim Start des Programmes (nach dem Öffnen der Konfigurationsdatei) geöffnet. Für das Dateiformat gelten dieselben Forderungen, wie bei der Konfigurationsdatei.

Es werden folgende Attribute unterstützt:

VERTEX_SIZE Größe eines Knoten (in Pixel). Voreinstellung: 10

VERTEX_COLOR Farbe eines Knoten. Voreinstellung: black

EDGE_WIDTH Breite einer Kante (in Pixel). Voreinstellung: 1

EDGE_COLOR Farbe einer Kante. Voreinstellung: black

ID_POSITION Position der Knoten-ID. Voreinstellung: left

DESCR_POSITION Position der Knotenbeschreibung. Voreinstellung: right

FONT_SIZE Größe der Schrift. Voreinstellung: 10

FONT_COLOR Farbe der Schrift. Voreinstellung: black

SHOW_ID Anzeige der Knoten-ID. Voreinstellung: yes

SHOW_DESCRIPTION Anzeige der Knotenbeschreibung. Voreinstellung: yes

SCALE Skalierung des Infrastrukturgraphen (in %). Voreinstellung: 100

GRIDWIDTH_V Vertikaler Abstand der Gitterlinien (in Pixel). Voreinstellung: 5

GRIDWIDTH_H Horizontaler Abstand der Gitterlinien (in Pixel). Voreinstellung: 5

COLOR_STEPS Anzahl der Farbabstufungen. Voreinstellung: 10

Das Vorhandensein und die Reihenfolge der Attribute ist beliebig. Bei fehlenden oder fehlerhaften Attributen wird der Voreinstellungswert angenommen. Da der Benutzer jedoch die Möglichkeit hat, die Optionen und damit auch den Inhalt der Optionen-Datei im Programm festzulegen, ist eine manuelle Bearbeitung selten nötig. Die vom Benutzer erstellte Optionen-Datei würde im Falle eines Abspeicherns der Optionen durch das Programm überschrieben. Abbildung 6.30 zeigt eine typische Optionen-Datei.

6.13.3 Infrastrukturgraph-Datei

In der Infrastrukturgraph-Datei gibt der Benutzer die Knoten und Kanten, aus denen ein Infrastrukturgraph besteht, an. Die Syntax einer Infrastrukturgraph-Datei lehnt an die XML-Syntax an, sie ist jedoch eingeschränkter. Die Verwendung von *Tags* macht die Infrastrukturgraph-Datei flexibler als ein festes Format, außerdem ist es für den Menschen besser lesbar. Ein *Element* besteht aus einem sogenannten *Start-Tag* (z.B. "<vertex>"), dem *Inhalt* (z.B. "Hauptbahnhof", "13.54" oder anderen Elementen, s.u.) und einem zum Start-Tag passenden *End-Tag* (z.B. < /vertex>). Im Folgenden werden die Elemente mit den Namen ihrer Tags bezeichnet. Außerdem gibt es noch *Leerzeilen*, welche vollkommen ignoriert werden, und *Kommentare*. Kommentare beginnen mit "<!−−", dann kann bis Zeilenende Beliebiges stehen. Es wird aber empfohlen, den Kommentar mit "−− >" abzuschließen.

Für das Format gilt:

- In einer Zeile darf nur stehen
  - ein Start-Tag oder
  - ein End-Tag oder

```
;options file for the Simulation ENVironment (SENV)
;author: automatically generated
;date: 24.07.2000

VERTEX_SIZE=10
VERTEX_COLOR=black
EDGE_WIDTH=1
EDGE_COLOR=black
ID_POSITION=left
DESCR_POSITION=right
FONT_SIZE=10
FONT_COLOR=black
SHOW_ID=yes
SHOW_DESCRIPTION=no
SCALE=100
GRIDWIDTH_V=5
GRIDWIDTH_H=5
COLOR_STEPS=10
```

Abbildung 6.30: Beispiel für eine Optionen-Datei

- der Inhalt eines Elements oder
- ein Element oder
- ein Kommentar oder
- eine Leerzeile.

➢ Leerzeichen und Tabs zu Beginn einer Zeile oder vor und nach einem Elementinhalt werden ignoriert

➢ Es muss bei den Tags auf Groß- und Kleinschreibung geachtet werden.

➢ Wenn eine Zeile nicht interpretiert werden kann, dann bleibt sie unbeachtet. Der Rest der Datei bleibt davon unberührt.

➢ Innerhalb des Inhaltes eines Elements ist kein "<" erlaubt, es sei denn es handelt sich um ein dort erlaubtes Tag.

Es gibt zwei Elemente, bei denen der Inhalt seinerseits wiederum nur aus Elementen bestehen darf: "vertex" und "edge". Manche Elemente sind nur erlaubt, wenn sie innerhalb bestimmter anderer Elemente auftreten, also verschachtelt sind. Es gibt drei "übergeordnete" Elemente, die nicht in anderen Elementen verschachtelt werden. Sie dürfen in beliebiger Reihenfolge stehen:

description Es beinhaltet eine Beschreibung des Infrastrukturgraphen. Dieses Tag kann einmal oder keinmal stehen; weitere Vorkommen werden ignoriert. Der Inhalt kann keine Tags enthalten.

vertex Beschreibt einen Knoten (s.u.). Dieses Element kann mehrmals vorkommen. Infrastrukturgraphen sollten mindestens aus einem Knoten bestehen. Der Inhalt muss bestimmte Tags enthalten.

edge Beschreibt eine Kante (s.u.). Dieses Element kann mehrmals vorkommen. Der Inhalt muss bestimmte Tags enthalten.

Die Angabe eines Knoten beginnt mit <*vertex*> und endet mit < /*vertex*>). Dazwischen müssen Elemente stehen, die den Knoten näher beschreiben. Die Elemente dürfen in beliebiger Reihenfolge, aber höchstens einmal vorkommen:

id Gibt die ID eines Knoten an, die aus einer eindeutigen natürlichen Zahl bestehen muss.

name (optional) Eine Beschreibung dieses Knotens.

x Die x-Koordinate des Knotens. Kann eine natürliche oder reelle Zahl (Format: Ziffern, optional gefolgt von einem Punkt und weiteren Ziffern) sein.

y Die y-Koordinate des Knotens. Für sie gelten die gleichen Forderungen wie für die x-Koordinate.

Die Beschreibung einer Kante wird von <*edge*> und < /*edge*> umschlossen. Auch hier gilt: die Reihenfolge der Elemente ist belanglos.

v1 ID des ersten Knotens.

v2 ID des zweiten Knotens. Da der Graph ungerichtet ist, ist die Reihenfolge egal.

Abbildung 6.31 fasst übersichtlich das Format einer Infrastrukturgraph-Datei zusammen. Abbildung 6.32 zeigt ein Beispiel für eine Infrastrukturgraph-Datei.

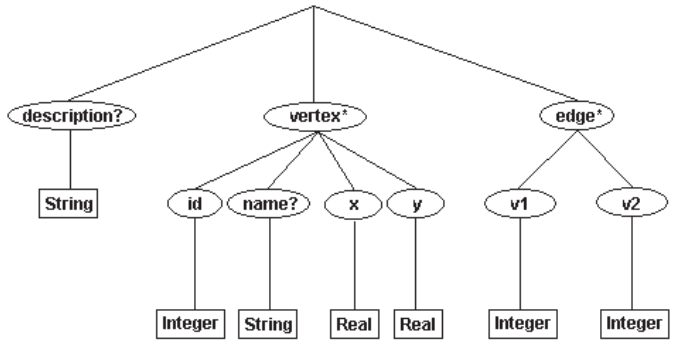

Abbildung 6.31: Format einer Infrastrukturgraph-Datei

6.13.4 Log-Datei

Das Verzeichnis, in dem die Log-Datei abgespeichert werden soll, kann in der Konfigurations-Datei festgelegt werden. Es kann nicht mehr geändert werden, während das Programm läuft. Der Name, unter dem die Log-Datei abgespeichert wird, lautet
`senvlog<nr>.txt`,
wobei `<nr>` um eins höher ist als die bislang höchste Nummer in dem Verzeichnis oder "1", wenn noch keine Log-Datei in diesem Verzeichnis existiert. Eine Log-Datei besteht aus dem *Header* und dem *Content*. Die *Header*zeilen stehen am Anfang der Datei und beginnen mit einem *Semikolon* (";"). Sie beinhalten die *Einstellungen*, mit denen die Simulationsreihe durchgeführt wurde. Es schließen sich die *Content*zeilen an.

```
<!-- created on 24.07.2000, by Mario Hegele -->
<description>
An infrastructure graph of Stuttgart's city centre
</description>
<!-- an example for an extremly space-wasting vertex definition -->
<vertex>
<id>
1
</id>
<name>
Hauptbahnhof
</name>
<y>
5
</y>
<x>
5
</x>
</vertex>
<!-- an example for a condensed vertex definition -->
<vertex><name>Hotel</name><id>3</id><x>4</x><y>7</y></vertex>
<!-- suggested format -->
<vertex>
  <id> 4 </id>
  <name> Fast Food-Tempel </name>
  <x> 23 </x>
  <y> 12.34 </y>
</vertex>
<!-- connection between the station and the hotel -->
<edge>
  <v1> 1 </v1>
  <v2> 3 </v2>
</edge>
```

Abbildung 6.32: Beispiel für eine Infrastrukturgraph-Datei

Darin stehen die IDs der Knoten und der Informationseinheiten. Die Ergebnisse der einzelnen Simulationsläufe sind durch eine Leerzeile voneinander getrennt. Am Anfang einer jeden Zeile steht die ID des Knoten, gefolgt von zwei Leerzeichen und den IDs der dort abgefragten Informationseinheiten (jeweils durch ein Leerzeichen getrennt). In Abbildung 6.33 wird ein Beispiel für eine Log-Datei gezeigt.

```
;Simulation ENVironment (SENV) log file
;created at: Friday, September 15, 2000 02:41:00 PM GMT+02:00
;Mobility Model: Simple mobility model
;User type: tourist (dedicated: 4, 5, 6, 8, 10, 14)
;Information Model: Simple information model
;10 disjunctive information items per location
;Access Model: Simple access model
;uniform distribution, 1/10 item(s) at (not) dedicated locations
;Infrastructure Graph: stuttgart.gra
;Number of simulation runs: 1
;Min. number of visited vertices: 0
;Max. number of visited vertices: 38
;Closed Walk: no
;Walk calculation: optimal algorithm is used up to 0 vertices
;Walk calculation: a heuristics is used up to 0 vertices
;Walk calculation: simple algorithm (nearest neighbour) for more
;vertices
;
;The simulation runs are separated by a blank line
;Each line starts with the vertex ID followed by the inf. items ID
;
4 41 41 49 48 45 41 49 47 42 49
10 104 108 102 109 101 106 106 107 102 103
9 93
12 120
9 97
```

Abbildung 6.33: Beispiel für eine Log-Datei

6.13.5 Resultats-Datei

Die Simulationsergebnisse werden in eine Datei in einem Format ausgegeben, das vom Zeichenprogramm *Gnuplot* verstanden und visualisiert werden kann (s. Abbildung 5.7). Zu einem Eintrag ("data point" in der Gnuplot-Sprache) gehören jeweils ein x- und

ein y-Wert, die durch ein Tabulatorzeichen getrennt auf einer Zeile stehen. Zeilen, die mit dem Zeichen "#" beginnen werden als Kommentar verstanden und beim Zeichnen ignoriert.

Abschnitt 6.12 kann entnommen werden, welche Simulationsergebnisse abgespeichert werden können. Als x-Wert eines Datensatzes dient immer die ID des Knoten, der Kante, der Informationseinheit oder des Besuchsfeldes (erste Spalte, s. Abbildungen 6.25-6.28). Den y-Wert stellt der Zähler (letzte bzw. vorletze Spalte).

An die Datensätze schließen sich - als Kommentar gekennzeichnet - wie bei den *Header*zeilen einer Log-Datei (s. Abschnitt 6.13.4) Informationen über die Einstellungen an, mit denen die Simulationsreihe durchgeführt wurde.

Kapitel 7

Implementierung

Die Implementierung der Simulationsumgebung nimmt einen großen Anteil an der Studienarbeit ein, da nicht nur eine prototypmäßige Programmierung, sondern eine "komfortable, graphische Benutzungsoberfläche", "eine umfangreiche Dokumentation des Quellcodes" und ein "Einsatzbeispiel" in der Aufgabenstellung gefordert waren. Wegen des zu erwartenden Aufwands wurden bei der Programmierung bestimmte Konzepte des *Software Engineerings* angewandt: die zu erstellende Software wurde grob in die drei Pakete *Benutzungsoberfläche*, *Simulatorkern* (Datenstrukturen, Simulationsablauf) und *Ausgabe der Ergebnisse* eingeteilt und in dieser Reihenfolge bearbeitet. Der Software-Prozess für jedes der Pakete umschließt die Aufgaben *Analyse*, *Spezifikation*, *Entwurf*, *Codierung* und *Test*.

Da zu erwarten ist, dass die Software um neue Submodelle erweitert wird, wird im nächsten Abschnitt beschrieben, wie man vorgehen muss, um eigene Submodelle einzubinden. Abschnitt 7.2 gibt einen Überblick über die in dieser Studienarbeit erstellte Software.

7.1 Einbindung eigener Submodelle

Eine Eigenschaft der Simulationsumgebung ist der flexible und modulare Aufbau, so dass einzelne Teilaspekte des Benutzerverhaltens (Mobilität, Anzahl der Informationszugriffe, etc.) voneinander unabhängig geändert werden können.

Wenn man ein neues Submodell in die Simulationsumgebung einbringen will, genügt nicht nur die Programmierung der Funktionalität des Submodells. Auch die Darstellung, Auswahl und Konfiguration müssen implementiert werden.

Die Einbindung eines neuen Submodells erfolgt grob in drei Schritten:

1. Erstellen des Submodells gemäß den Schnittstellen (s. Abschnitt 7.1.1).

2. Erstellen eines "Submodell-Managers", der das Submodell instantiiert und konfiguriert (s. Abschnitt 7.1.2).

3. Einbinden des Submodell-Managers in das Programm (s. Abschnitt 7.1.3).

Vor der Integration eines eigenen Submodells empfiehlt sich auch das Studium der bereits vorhandenen sechs Submodelle und zwei Submodell-Managern.

7.1.1 Erstellen eines Submodells

Für das reibungslose Zusammenspiel der Submodelle ist das Einhalten von bestimmten Schnittstellen nötig. Die Programmiersprache *Java* unterstützt sogenannte *Interfaces*. Jeder der drei Submodelltypen (Mobilitäts-, Informations- und Zugriffsmodell) muss ein bestimmtes Interface unterstützen, d.h. gewisse Methoden implementieren. Darüberhinaus gibt es noch ein Interface, das *alle* Submodelltypen unterstützen müssen.

Interface *Submodel*

Jeder Submodelltyp muss die einzige Methode, die dieses Interface bietet, implementieren:

```
public interface Submodel {
   /**
   * make all the necessary settings before a new simulation run begins
   */
   public void reset();
}
```

Diese Methode wird jedesmal aufgerufen, bevor ein neuer Simulationslauf beginnt. Da die Abhängigkeiten der Submodelle untereinander berücksichtigt werden, kann davon ausgegangen werden, dass zum Zeitpunkt des Aufrufs dieser Methode, die anderen Submodelle instantiiert und vollständig konfiguriert sind.

Interface *MobilityModel*

Jedes Mobilitätsmodell unterstützt die Schnittstelle *MobilityModel*. Die Methoden betreffen die Knoten des Infrastrukturgraphen, die physikalische Orte ("Locations") repräsentieren:

```
public interface MobilityModel extends Submodel {
   /**
    * Determines a set of locations which absolutely must be visited
    * during a walk through the infrastructure graph.
    *
    * @return the locations to be visited
    */
   public Vertex[] getLocations();
   /**
    * Determines the visit probability for a location.
    * @param location the visited location
    * @return returns the probability with which a location is visited
    */
   public double getVisitProbability(Vertex location);
}
```

Die erste Methode wird zu Beginn eines jeden Simulationslaufes aufgerufen und liefert dann diejenigen Knoten/Orte des Infrastrukturgraphen, die auf jeden Fall besucht werden müssen (d.h. die Zielknoten). Doppelt vorkommende Knoten werden vor der Suche nach einem Weg über die Zielknoten entfernt. Die Übergabe aller Knoten des Infrastrukturgraphen muss *vor* der Simulation durch den Submodell-Manager erfolgen.

Die zweite Methode wird eventuell (je nach Abhängigkeit) von den anderen beiden Submodellen aufgerufen. Sie gibt die Besuchswahrscheinlichkeit für einen Knoten zurück. Es wird erwartet, dass die Methode während eines Simulationslaufes immer dieselbe Besuchswahrscheinlichkeit für einen Knoten zurückgibt. Abweichungen davon sollten im Kommentar der Methode kenntlich gemacht werden.

Interface *InformationModel*

Das Informationsmodell legt alle Informationseinheiten fest, die während einer Simulationsreihe angeboten werden, und von welchen Orten aus darauf zugegriffen werden kann. Dazu muss es die Schnittstelle *InformationModel* unterstützen:

```
public interface InformationModel extends Submodel {
   /**
    * Determines the information items which are assigned to the
    * vertices.
    *
    * @param location the location from where the information items
    * are requested
    * @return returns all the information items which are associated
    * to a location
    */
   public int[] getAssociatedItems(Vertex location);
   /**
    * @return returns the total number of information items
    * (information item
    * IDs range from 0 to totalNumber-1
    */
   public int getTotalNumberOfItems();
}
```

Informationseinheiten in der Simulationsumgebung, bzw. deren IDs, bestehen aus einer natürlichen Zahl. Die Methode *getAssociatedItems(Vertex location)* gibt für einen bestimmten Knoten diejenigen Informationseinheiten zurück, die dort abgefragt werden können. Der Aufruf erfolgt für jeden besuchten Knoten durch das Zugriffsmodell. Die gelieferten Informationseinheiten müssen nicht disjunkt sein, es wird aber erwartet, dass während einer Simulationsreihe für einen Knoten immer *dieselben* Informationseinheiten zurückgegeben werden.

Die Methode *getTotalNumberOfItems()* gibt an, wieviele verschiedene Informationseinheiten höchstens zu erwarten sind. Diese Information wird vor der Simulation abgerufen und an einen Zähler weitergegeben. Die IDs der Informationseinheiten müssen zwischen 0 und der maximalen Anzahl liegen.

Interface *AccessModel*

Das Zugriffsmodell bestimmt aus allen Informationseinheiten, die einem Knoten zugeordnet sind, diejenigen, die abgefragt werden:

```
public interface AccessModel extends Submodel {
   /**
    * Determines the information items which are requested when
```

```
    * visiting a location.
    *
    * @param location the location from where the information items
    * are requested
    * @return returns the information items which are requested
    * at a location
    */
   public int[] getAccessedItems(Vertex location);
}
```

Die Methode wird während des Simulationslaufes für jeden Knoten des Weges, der über Ziel- und Durchreiseknoten führt, aufgerufen. Die Informationseinheiten sind eine Teilmenge aller Informationseinheiten, die das Informationsmodell für diesen Knoten liefert.

7.1.2 Erstellen eines Submodell-Managers

Eine Klasse, die Submodelle verwaltet, muss das Interface *SubmodelManager* unterstützen:

```
public interface SubmodelManager {
   /**
    * dependency constant: independent
    */
   public final static int INDEPENDENT = 0;
   /**
    * dependency constant: dependent on the mobility model
    */
   public final static int MOBILITY_MODEL_DEPENDENT = 1;
   /**
    * dependency constant: dependent on the information model
    */
   public final static int INFORMATION_MODEL_DEPENDENT = 2;
   /**
    * dependency constant: dependent on the access model
    */
   public final static int ACCESS_MODEL_DEPENDENT = 4;
   /**
    * submodel-type constant : mobility model
```

```
 */
public final static int MOBILITY_MODEL = 0;

/**
 * submodel-type constant : information model
 */public final static int INFORMATION_MODEL = 1;

/**
 * submodel-type constant : access model
 */public final static int ACCESS_MODEL = 2;

/**
 * Indicates that a submodel is defined for any number of vertices
 */
public final static int ALL_VERTICES = Integer.MAX_VALUE;

/**
 * @param modelKind see submodel constants
 * @return returns a short description for the specified submodel
 * @see #MOBILITY_MODEL
 * @see #INFORMATION_MODEL
 * @see #ACCESS_MODEL
 */
public String getDescription(int modelKind);

/**
 * @param modelKind see submodel constants
 * @return returns the number of supported vertices
 * for the specified submodel
 * @see #MOBILITY_MODEL
 * @see #INFORMATION_MODEL
 * @see #ACCESS_MODEL
 */
public int getNoOfSupportedVertices(int modelKind);

/**
 * The dependency is indicated by a logical OR-combination
 * of the dependency constants.
 *
 * @param modelKind see submodel constants
 * @return returns a value which indicates the dependencies of
 * the submodel
 * @see #MOBILITY_MODEL
 * @see #INFORMATION_MODEL
 * @see #ACCESS_MODEL
 * @see #INDEPENDENT
 * @see #MOBILITY_MODEL_DEPENDENT
```

```
 * @see #INFORMATION_MODEL_DEPENDENT
 * @see #ACCESS_MODEL_DEPENDENT
 */
public int getDependencies(int modelKind);

/**
 * @return returns the supported mobility model
 */
public MobilityModel getMobilityModel();

/**
 * is called once before the whole simulation starts
 * and after the infrastructure graph and the other (not completely
 * configured) submodels are known and instantiated
 */
public void makeSettingsMobilityModel(InfrastructureGraph i,
   InformationModel iModel,
   AccessModel aModel)
   throws Exception;

/**
 * @return returns the supported information model
 */
public InformationModel getInformationModel();

/**
 * is called once before the whole simulation starts
 * and after the infrastructure graph and the other (not completely
 * configured) submodels are known and instantiated
 */
public void makeSettingsInformationModel(InfrastructureGraph i,
   MobilityModel mModel,
   AccessModel aModel)
   throws Exception;

/**
 * @return returns the supported access model
 */
public AccessModel getAccessModel();

/**
 * is called once before the whole simulation starts
 * and after the infrastructure graph and the other (not completely
 * configured) submodels are known and instantiated
 */
public void makeSettingsAccessModel(InfrastructureGraph i,
   MobilityModel mModel,
```

```
      InformationModel iModel)
      throws Exception;
   /**
    * Is called when the user has selected a submodel and pressed
    * the ``configure'' button.
    * After this call the managed submodels must be available (get...Model),
    * i.e. instantiated.
    * @param modelKind which kind of model should be configured
    * @return Returns any kind of String data (e.g.
    * the configuration parameters) - which is displayed - ,
    * if the submodel has been configured successfully. Returns ``null'',
    * if the user has not configured the submodel successfully (e.g.
    * if the user has cancelled the configuration)
    * @see #MOBILITY_MODEL
    * @see #INFORMATION_MODEL
    * @see #ACCESS_MODEL
    */
   public String configureSubmodel(int modelKind);
}
```

Die erste Methode *getDescription(int modelKind)* liefert diejenige Textzeile, die in den Auswahllisten des Submodell-Fensters (s. Abschnitt 6.4) dem Benutzer angezeigt wird. Hier - wie in den meisten folgenden Methoden - bezeichnet der Parameter *modelkind* den Submodelltyp (Mobilitäts-, Informations- oder Zugriffsmodell), der über die Konstanten angegeben wird. Durch Einbinden eines Submodell-Managers in das richtige Submodelltyp-Array (s. Abschnitt 7.1.3) stellt man sicher, dass der Submodell-Manager nur für diejenigen Submodelltypen aufgerufen wird, die er auch verwaltet.

Ein Submodell muss nicht für jede Anzahl von Knoten definiert sein, z.B. wenn es nur für einen speziellen Infrastrukturgraphen geschrieben wurde. Die Anzahl der unterstützten Knoten wird über die Methode *getNoOfSupportedVertices(int modelKind)* ermittelt. Wenn jede Anzahl von Knoten unterstützt wird, dann wird *ALL_VERTICES* zurückgegeben.

Die dritte Methode, *getDependencies(int modelKind)*, zeigt die Abhängigkeit eines Submodells von anderen Submodellen an. Dies geschieht durch die Rückgabe eines drei-Bit-Vektors, bei dem eine "1" eine Abhängigkeit bedeutet (s. "dependency constant"). Z.B. bedeutet "010", dass das Submodell vom Informationsmodell ("INFORMATION_MODEL_DEPENDENT") abhängt. Eine Abhängigkeit eines Submodells von sich selbst wird implizit angenommen. Die Erstellung des Bit-Vektors wird durch eine ODER-Verknüpfung der entsprechenden Konstanten erreicht.

Die drei *get...model*-Methoden werden vor Simulationsbeginn aufgerufen und müssen eine Instanz des jeweiligen Submodells liefern, die dann auch in der Simulation verwen-

det wird. Die vollständige Konfiguration erfolgt in den *makeSettings...Model*-Methoden.

Eine vorläufige[1] Konfiguration findet statt, wenn der Benutzer der Simulationsumgebung im Submodell-Fenster ein Submodell auswählt oder den "Configure"-Knopf drückt. Wie die Auswahl der Konfigurationsparameter für ein Submodell geschieht (per Zufall, per Bedieneroberfläche, ...), ist nicht festgelegt. Dafür können auch zusätzliche Klassen, z.B. für eine graphische Eingabemaske (s. Einsatzbeispiel in Kapitel 5) eingestezt werden. Wenn die Methode "null" zurückliefert, wurde der Konfigurationsvorgang abgebrochen, und es liegt kein zur Simulation geeignetes Submodell vor.

7.1.3 Einbinden eines Submodell-Managers

Das Einbinden eines Submodell-Managers erfolgt in der Klasse *SEnv*. Damit wird gesichert, dass die angebotenen Submodelle im Submodell-Fenster ausgewählt und konfiguriert werden können. Vor Simulationsbeginn liefert der Submodell-Manager eine Instanz des jeweiligen Submodells. In der Methode *initSubmodelManagers()* muss für jeden unterstützten Submodelltyp eine Instanz des Submodell-Managers[2] in das entsprechende Array von Submodell-Managern eingebunden werden. Ein Submodell-Manager muss nicht für jeden Submodelltyp ein Submodell anbieten. Es folgt ein Beispiel für die Methode:

```
private void initSubmodelManagers() {
   SubmodelManager simpleManager = new SimpleSubmodelManager();
   SubmodelManager myManager1 = new mySubmodelManager();
   SubmodelManager myManager2 = new mySubmodelManager();
   SubmodelManager dedVerticesSubmodelManager =
      new DedVerticesSubmodelManager();

   // array for the mobility model managers
   mobilityModelManager = new SubmodelManager[]
      {simpleManager, myManager1, dedVerticesSubmodelManager};

   // array for the information model managers
   informationModelManager = new SubmodelManager[]
      {simpleManager, dedVerticesSubmodelManager};

   // array for the access model managers
   accessModelManager = new SubmodelManager[]
      {simpleManager, dedVerticesSubmodelManager, myManager2};
}
```

[1] *Vorläufig*, weil die anderen Submodelle noch nicht instantiiert sind und der Infrastrukturgraph noch nicht festgelegt ist.

[2] Notwendig ist ein Import des enthaltenden Packages oder eine vollständige Referenzierung mit den Package-Namen.

7.2 Überblick über die erstellte Software

7.2.1 Umfang der Software

Tabelle 7.1 zeigt, wie sich die Software zum Zeitpunkt des Studienarbeit-Abschlusses zusammensetzt. Spätere Erweiterungen oder Änderungen sind nicht auszuschließen. Eine detaillierte Angabe über die Packages und deren Klassen findet sich in Anhang B.

Package	**enthält .. Packages**	**enthält .. Klassen**	**Größe in kByte**	**Beschreibung**
senv.control	0	1	35	enthält Klasse mit main()-Methode
senv.core	0	9	61	Simulatorkern
senv.gui	0	15	183	Graphische Oberfläche
senv.model	0	8	38	Submodelle und Submodell-Manager
senv.utils	0	21	123	Zusätzliche Klassen
senv	5	54	440	Übergeordnetes Package

Tabelle 7.1: Software-Übersicht

7.2.2 Verwendete Tools

Wie schon erwähnt, wurde die Software in der Programmiersprache *Java* von *Sun Microsystems Inc.* geschrieben, die den Vorteil einer Portabilität der Klassen besitzt. Es wurde die Version *JDK-1.2.2* unter *SunOS 5.7* und *Windows 95* eingesetzt.

Eine Konvention bei Java ist es, dass die Verzeichnis-Struktur die Package-Struktur widerspiegelt. Es muss also ein Verzeichnis geben, in dem sich das Verzeichnis `senv` mit den Quelltexten befindet. Dieses Verzeichnis wird im Folgenden mit $SRC bezeichnet. Die Klassendateien, die beim Kompilieren entstehen, sollen in das bereits vorhandene Verzeichnis $CLASSES geschrieben werden. Eine Neukompilierung einer Quelltextdatei, z.B. der Hauptdatei, erfolgt mit dem Befehl (ohne Zeilenumbruch)

```
javac -d $CLASSES -classpath $CLASSES -sourcepath $SRC
$SRC/senv/control/SEnv.java
```

Der Start des Programmes (s. Abschnitt 6.1) kann dann mit dem Interpreter erfolgen.

Zum Schreiben der Quelltexte wurde der *Emacs*-Editor verwendet, der durch eine Syntax-Analyse imstande ist, Einrückungen vorzunehmen, Schlüsselwörter farblich hervorzuheben und andere Verstöße gegen die Syntax zu erkennen. Zusätzlich wurden Teile des Quelltextes von einer graphischen Entwicklungsumgebung generiert (*forte*, auch von Sun Microsystems).

Kapitel 8

Zusammenfassung

In dieser Studienarbeit wurden ein neues Benutzermodell und eine darauf aufbauende Simulationsumgebung vorgestellt. Die Studienarbeit basiert auf einem bislang noch nicht veröffentlichten Artikel ([KR00]).

Das Benutzermodell setzt sich aus den drei Submodellen Mobilitätsmodell, Informationsmodell und Zugriffsmodell zusammen. Es stellt auf einem Graphen gleichermaßen die Bewegungen und Informationszugriffe eines Benutzers eines mobilen Informationssystems nach. Es wurden verschiedene Alternativen zu den gewählten Modellierungen gezeigt. Anhand eines Einsatzbeispiels wurde demonstriert, wie die Submodelle konkret aussehen können und wie eine Simulation abläuft.

Die programmierte Simulationsumgebung baut auf einem Prototyp auf, der gewisse Schwächen im Gebrauch offenbart hat. Es wurden vor allem Wert auf eine verbesserte Benutzerinteraktion (mehr Komfort durch eine graphische Oberfläche) und die Aufarbeitung der Simulationsergebnisse gelegt.

Durch ihre Konfigurationsmöglichkeiten kann die Simulationsumgebung an spezielle Bedürfnisse angepasst werden. Sollten die bereits vorhandenen Submodelle sich als nicht ausreichend erweisen, so können eigene Submodelle programmiert und leicht in das Programm eingebunden werden. Mögliche künftige Erweiterungen könnten die Implementierung eines graphischen Editors für Infrastrukturgraphen oder die Vergrößerung der Submodell-Bibliothek sein.

Die Simulation von Mobilität und Informationszugriff bei mobilen Informationssystemen ermöglicht neben der Analyse solcher Systeme auch die Evaluierung von Mechanismen zur Überwindung der Nachteile von drahtloser Kommunikation beim mobilen Informationszugriff. Solche Mechanismen sind z.B. das *Caching* oder *Hoarding* (vorsorgliches Laden) von Informationen. Da mobile Computer immer mehr Verbreitung finden, wird auch die Nachfrage nach Benutzermodellen und deren Simulation bestehen bleiben.

Literaturverzeichnis

[BCFPS99] Breslau, Lee, Pei Cao, Li Fan, Graham Phillips and Scott Shenker, "Web Caching and Zipf-like Distributions: Evidence and Implications", *Proceedings of the IEEE INFOCOM '99*, New York City, NY, USA, 1999.

[C76] N. Christofides, "Worst-case analysis of a new heuristic for the traveling salesman problem", *Technical report, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh PA*, 1976.

[CDMF00] Cheverst, K., N. Davies, K. Mitchell and A. Friday, "Experiences of Developing and Deploying a Context-Aware Tourist Guide: the GUIDE project", *Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking (MobiCom '00)*, p. 20-31, Boston, MA, USA, Aug. 2000.

[DGM98] de Nitto Personè, Vittoria, Vincenzo Grassi and Antonio Morlupi, "Modeling and Evaluation of Prefetching Policies for Context-Aware Information Services", *Proceedings of the 4th Annual International Conference on Mobile Computing and Networking (Mobicom '98)*, p. 55-65, Dallas, Texas, USA, Oct. 1998.

[KR00] Kubach, Uwe and Kurt Rothermel, "A Simulation Framework for Mobile Information Access", 2000.

[LBC98] Liu, T., P. Bahl and I. Chlamtac, "Mobility modeling, location tracking and trajectory prediction in wireless atm networks, *IEEE Journal on Selected Areas in Communications 16(6)*.

[LK73] Lin, S. and B. W. Kernighan, "An effective heuristic algorithm for the traveling salesman problem", *Operations Research 21 (1973)*, p. 498-516.

[O95] Oppenheim, Norbert, "Urban Travel Demand Modeling", *John Wiley & Sons, Inc.*, 1995.

[P57] Prim, R.C., "Shortest connection networks and some generalizations", *Bell System Technical Journal*, 36:1389-1401, 1957.

[SDK83] Syslo, M., N. Deo and J. Kowalik, "Discrete Optimization Algorithms with Pascal Programs", *Prentice Hall, Englewood Cliffs NJ*, 1983.

[sun1] Sun Microsystems, Inc., "Java Look and Feel Design Guidelines"

[sun2] Sun Microsystems, Inc., "Java API Specification v.1.2.2"

[sun3] Sun Microsystems, Inc., "The Java Tutorial: A Practical Guide for Programmers"

[sun4] Sun Microsystems, Inc., "Code Conventions for the Java Programming Language"

[SWH98] Sistla, A., Ouri Wolfson and Yixiu Huang, "Minimization of Communication Cost Through Caching in Mobile Environments", *IEEE Transaction on Parallel and Distributed Systems 9(4)*

[S98] Skiena, Steven S., "The Algorithm Design Manual", *Springer-Verlag New York, Inc.*, 1998.

Anhang A

Glossar

Das Glossar enthält für die Begriffe, die innerhalb der Studienarbeit eine wichtige Rolle spielen, eine kurze Definition. Mit dem Glossar sollen neue Begriffe geklärt und ein gemeinsames Verständnis erreicht, sowie die Konsistenz gesichert werden. Dazu gehört auch ein Wörterbuch, das in der Literatur und in der zu programmierenden Applikation auftretende englische Begriffe übersetzt. Diese englischen Wörter sind *rekursiv* hervorgehoben und verweisen auf die deutsche Übersetzung. Verweise sind mit einem Symbol ➢ gekennzeichnet.

Abstrahierung Vorgang, bei dem aus den ➢ `Zielknoten` eines ➢ `Infrastrukturgraphen` und ihren kürzesten Verbindungen miteinander kleinerer Graph gebildet wird. Die ➢ `Pfade` zwischen den Zielknoten werden durch Kanten mit den Längen der Pfade ersetzt. Die ➢ `Durchreiseknoten`, die auf den Pfaden zwischen den Zielknoten liegen, werden dabei vorläufig ignoriert. Siehe ➢ `Expandierung`.

access model ➢ `Zugriffsmodell`

Benutzer 1. B. der Applikation, d.h. der Simulationsumgebung
2. B. eines ➢ `Informationssystems`, dessen Verhalten mit der Simulation nachgeahmt wird
Aus dem Kontext sollte hervorgehen, welche dieser beiden Deutungen gemeint ist.

Benutzermodell Das B. modelliert das Verhalten der Benutzer eines Informationssystems: ihre Bewegungen innerhalb des ➢ `Infrastrukturgraphen` und ihre Zugriffe auf Informationen. Es setzt sich aus den drei ➢ `Submodellen` ➢ `Mobilitätsmodell`, ➢ `Informationsmodell` und ➢ `Zugriffsmodell` zusammen.

Beschreibung B. eines ➢ `Knoten`. Gibt optional an, welchen physikalischen Ort der Knoten repräsentiert, z.B. "Bahnhof". Siehe auch ➢ `ID`.

Besuchsfeld Das Gebiet, das der ➢ `Infrastrukturgraph` abdeckt, wird durch vertikale und horizontale Linien in Rechtecke, bzw. B. eingeteilt.

Besuchswahrscheinlichkeit B. eines ➢ `Knoten`. Gibt an, mit welcher Wahrscheinlicheit ein Knoten bei einem ➢ `Simulationslauf` zum ➢ `Zielknoten` wird. Ist etwa die B. 0 (0%), dann wird der Knoten nur besucht, wenn er als ➢ `Durchreiseknoten` auf einem ➢ `Weg` liegt, bei 1 (100%) wird er auf jeden Fall besucht, weil er immer ein ➢ `Zielknoten` ist.

Besuchswahrscheinlichkeitskarte Diese B. zeigt dasselbe Gebiet wie der ➢ `Infrastrukturgraph`, allerdings ist ein Liniengitter darüber gelegt, so dass ➢ `Besuchsfelder` entstehen. Die Besuchsfelder werden je nach Besuchshäufigkeit eingefärbt. Die B. kann erst nach der Durchführung von mindestens einem ➢ `Simulationslauf` erstellt werden.

Durchreiseknoten Die D. sind Knoten, die während eines *Simulationslaufes* besucht werden, weil sie auf der kürzesten oder gar einzigen Verbindung zwischen zwei ➢ `Zielknoten` liegen.

Expandierung Nachdem ein ➢ `Weg` über die ➢ `Zielknoten`, die den kleineren Graphen bilden, bestimmt wurde, werden alle ➢ `Kanten` zwischen diesen Zielknoten durch die ursprünglichen ➢ `Pfade` des Originalgraphen ersetzt. Die *Knoten*, die zwischen den Zielknoten liegen, werden *Durchreiseknoten* genannt. Siehe ➢ `Abstrahierung`.

Graph, kompletter Ein Graph, bei dem jeder Knoten direkt mit jedem anderen Knoten verbunden ist.

Graph, vollständiger ➢ `Graph, kompletter`

ID ID eines ➢ `Knoten`. Jeder Knoten im ➢ `Infrastrukturgraphen` besitzt eine eindeutige Identität in Form einer natürlichen Zahl. Siehe auch ➢ `Beschreibung`.

information item ➢ `Informationseinheit`

information model ➢ `Informationsmodell`

Informationseinheit Eine I. kann z.B. eine Web-Seite sein, ein Kartenausschnitt oder sonst eine Datei. I. haben eine eindeutige ➢ `ID`.

Informationsmodell Weist den Orten eines Infrastrukturgraphen ➢ `Informationseinheiten` zu.

Informationssystem Ein I. gestattet den Zugriff auf eine Menge von bestimmten ➢ `Informationseinheiten`.

Infrastruktur Mit I. sind hier die Orte und Verbindungen zwischen diesen Orten in dem betrachteten Gebiet gemeint. Orte können z.B. Sehenswürdigkeiten oder Haltestellen sein, die durch Straßen oder Schienen miteinander verbunden sind.

Infrastrukturgraph Ein Graph, der die Infrastruktur des Gebiets, in dem sich der Benutzer eines ➢ `Informationssystems` aufhält, repräsentiert.

Kante K. verbinden immer zwei ➢ `Knoten` und repräsentieren im ➢ `Infrastrukturgraphen` z.B. Straßen- oder Zugverbindungen. Der Benutzer verwendet die K., um von einem Ort zur anderen zu kommen.

Knoten Im ➢ `Infrastrukturgraphen` repräsentieren K. Orte.

Mobilitätsmodell Legt die ➢ `Besuchswahrscheinlichkeit` eines ➢ `Knoten` bzw. eines Ortes fest und bestimmt, *welche* Knoten der ➢ `Benutzer` während eines ➢ `Simulationslaufes` besucht.

mobility model ➢ `Mobilitätsmodell`

Pfad ➢ `Weg`, in dessen Knotenfolge kein ➢ `Knoten` doppelt vorkommt.

Simulationslauf Simulation einer einzelnen Benutzerreise: der ➢ `Benutzer` besucht bestimmte ➢ `Knoten` und ruft dort ➢ `Informationseinheiten` ab.

Simulationsreihe Zu einer Reihe von Simulationsläufen gehören alle Simulationsläufe, die unmittelbar hintereinander mit exakt denselben Einstellungen ausgeführt werden.

Simulatorkern Der S. koordiniert den Simulationsprozess. Er bedient sich dabei des ➢ `Infrastrukturgraphen` und der ➢ `Submodelle`.

Submodell Bestandteil des ➢ `Benutzermodells`

Submodelltyp Jedes ➢ `Submodell` kann einem der Typen ➢ `Mobilitätsmodell`, ➢ `Zugriffsmodell` oder ➢ `Informationsmodell` zugeordnet werden.

visit probability map ➢ `Besuchswahrscheinlichkeitskarte`

Weg Eine Folge von ➢ `Knoten`, die über ➢ `Kanten` miteinander verbunden sind.

Weg, geschlossener Ein Weg, bei dem Anfangs- und Endknoten identisch sind.

Zielknoten Aufgrund der ➢ `Besuchswahrscheinlichkeit` von ➢ `Knoten` bestimmt das ➢ `Mobilitätsmodell` bei jedem ➢ `Simulationslauf` neu die sogenannten Z. Diese Knoten sind die Ziele der Reise eines mobilen ➢ `Benutzers` und müssen während des Simulationslaufes auf jeden Fall besucht werden.

Zugriffsmodell Bestimmt, welche ➢ `Informationseinheiten` vom ➢ `Benutzer` beim Besuch eines Ortes abgefragt werden.

Anhang B

Klassenübersicht

Die Simulationsumgebung setzt sich aus einem Haupt-Package (*senv*) und fünf untergeordneten Packages (*core, control, gui, model, utils*) zusammen (s. Abbildung B.1).

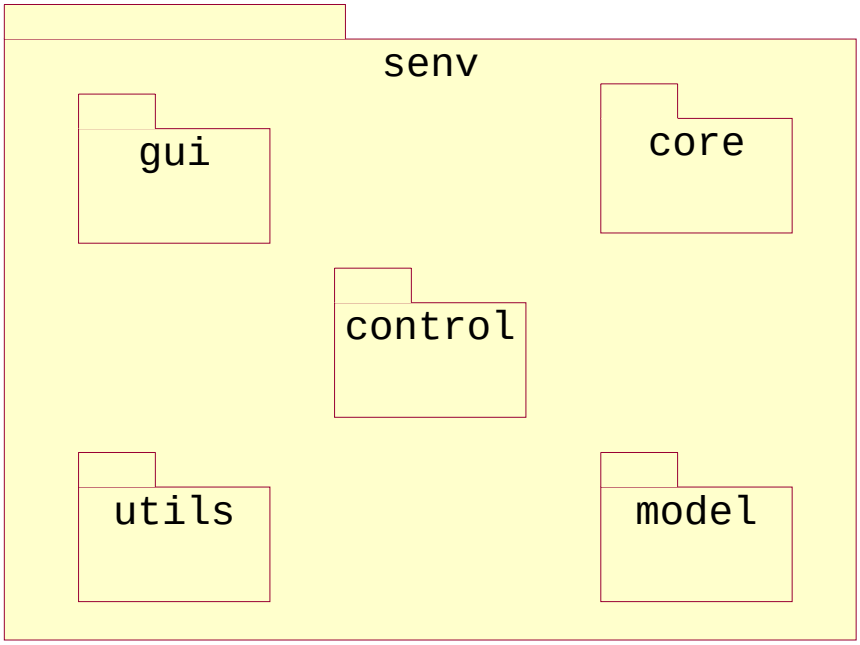

Abbildung B.1: Package-Übersicht

Quelltexte bzw. Klassendateien befinden sich nur in den untergeordneten Packages.

Package *control*

Enthält die Klasse mit der *main()*-Methode.

Klassenname	Größe in Byte
senv/control/SEnv.java	36192

Package *core*

Enthält Klassen, die hauptsächlich den eigentlichen Simulationsvorgang und die daran beteiligten Datenstrukturen betreffen.

Klassenname	Größe in Byte
senv/core/AccessModel.java	1394
senv/core/Edge.java	4391
senv/core/InformationModel.java	1790
senv/core/InfrastructureGraph.java	17313
senv/core/MobilityModel.java	1549
senv/core/Simulator.java	26137
senv/core/Submodel.java	1021
senv/core/SubmodelManager.java	5415
senv/core/Vertex.java	4275

Package *gui*

Die Klassen dieses Packages sind für die graphische Benutzungsoberfläche zuständig.

Klassenname	Größe in Byte
senv/gui/AboutFrame.java	3165
senv/gui/DedVerticesAccModel.java	10760
senv/gui/DedVerticesInfModel.java	8003
senv/gui/DedVerticesMobModel.java	12989
senv/gui/GraphMapFrame.java	38045
senv/gui/InformationPanel.java	3163
senv/gui/InfrastructureGraphPanel.java	
senv/gui/MainFrame.java	4176
senv/gui/OptionFrame.java	18980
senv/gui/SEnvMenuBar.java	13309
senv/gui/SimulationRunPanel.java	17472
senv/gui/StopDialog.java	5042
senv/gui/SubmodelPanel.java	17362
senv/gui/TextualGraphFrame.java	9780
senv/gui/TextualResultsFrame.java	18441

Package *model*

Dieses Package beinhaltet die eingebundenen Submodelle und Submodell-Manager.

Klassenname	Größe in Byte
senv/model/DedVerticesAccModel.java	6893
senv/model/DedVerticesInfModel.java	3187
senv/model/DedVerticesMobModel.java	3283
senv/model/DedVerticesSubmodelManager.java	11905
senv/model/SimpleAccessModel.java	2764
senv/model/SimpleInformationModel.java	2846
senv/model/SimpleMobilityModel.java	3234
senv/model/SimpleSubmodelManager.java	5304

Package *utils*

In diesem Package sind zusätzliche Klassen enthalten, die Exceptions, Dateioperationen und Containerklassen betreffen.

Klassenname	Größe in Byte
senv/utils/ConfigFileLoader.java	4751
senv/utils/Counter.java	9610
senv/utils/EdgeExistsException.java	1188
senv/utils/EdgeNotLegalException.java	1178
senv/utils/EdgeVector.java	2298
senv/utils/GraphConverter.java	6002
senv/utils/GraphFileFilter.java	1356
senv/utils/GraphLoader.java	19229
senv/utils/HeuristicsWalk.java	8656
senv/utils/MyFileFilter.java	1271
senv/utils/Options.java	27017
senv/utils/OptWalkAlgorithm.java	6968
senv/utils/Path.java	4952
senv/utils/ResultsLogger.java	6422
senv/utils/ResultsSaver.java	13697
senv/utils/SimpleWalkAlgorithm.java	3188
senv/utils/Square.java	1373
senv/utils/VertexExistsException.java	1213
senv/utils/VertexNotLegalException.java	1210
senv/utils/VertexVector.java	2406
senv/utils/WalkAlgorithm.java	2141

Ich versichere, dass ich diese Arbeit selbständig verfaßt
und nur die angegebenen Hilfsmittel verwendet habe.

_____