Aufzugsteuerung – Simulation und Optimierung

Harald Schilly Richard Welke Moritz Wurnig

Zusammenfassung

Im Rahmen des Projektseminars Angewandte Mathematik im Wintersemester 2004/05 sollte eine Simulation von Aufzugsystemen entwickelt werden, mit der verschiedene Strategien zur Aufzugsteuerung verglichen werden können und ein Parameter für die verschiedenen Strategien optimal bestimmt werden kann. Das gesamte Projekt ist in JAVA™ 5.0 von SUN MICROSYSTEMS implementiert worden. Mit den implementierten Strategien wurden umfangreiche Testläufe durchgeführt, die einen intensiven Vergleich verschiedener Strategien ermöglichen. Die Versuchsergebnisse werden gedeutet und Aussagen darüber gemacht, welche der getesteten Strategien zur Aufzugsteuerung sich optimal verhält. Es stellt sich heraus, dass unter den implementierten Standardalgorithmen die *Random-Target* und die *Nearest-Neighbor*-Strategie zur Aufzugsteuerung am besten geeignet sind. Die im Rahmen des Projekts entstandenen Materialien finden sich unter [1].

1 Grundlagen der Aufzugsteuerung

1.1 Aufzugsysteme

Aufzugsysteme in Gebäuden bestehen in der Regel aus M Aufzügen, die Passagiere von ihrem Einstiegsstockwerk zu ihrem Zielstockwerk in einem Haus mit N Stockwerken bringen.

Die traditionellen Bedingungen, die man an das Verhalten der Aufzüge stellt (vgl. z. B. [2]), sind:

- Ein Aufzug darf nicht an einem Stockwerk vorbeifahren, an dem ein Passagier aussteigen möchte
- Ein Aufzug muss seinen gegenwärtigen Auftrag abarbeiten und darf nicht die Fahrtrichtung wechseln, solange er nicht leer ist
- Ein leerer Aufzug kann anhalten, hinauf- oder hinunterfahren

Die Aufzüge reagieren dabei auf zwei verschiedene Arten von Passagierwünschen:

- *car calls*, Fahrzielwunsch durch Tastendruck im Aufzug
- *hall calls*, Aufzugsruf in den Stockwerken.

1.2 Modellannahmen

Wie in der Literatur stets vorausgesetzt (vgl. z. B. [2]) nehmen wir an, dass die Ankunft neuer Passagiere einer Poisson-Verteilung folgt.

Um die Modellierung der Aufzugfahrten nicht zu komplex werden zu lassen, wird vorausgesetzt, dass die Aufzüge für die Fahrt eine konstante Zeit pro Stockwerk benötigen und eine bestimmte Zeit brauchen, um zu beschleunigen und abzubremsen. Diese wird zu der Zeit hinzugerechnet, die der Aufzug zum Öffnen bzw. Schließen der Türen benötigt. Jeder Passagier benötigt eine fixe Zeit zum Ein- bzw. Aussteigen. Diese Parameter sind in der Simulation einstellbar.

Kann ein Aufzug nicht alle wartenden Personen bedienen, die hinter einem Ruf in einem Stockwerk stehen, so wird der zugrundeliegende Ruf nicht gelöscht.

Für die Simulation des Verkehrsaufkommens sind zwei Parameter wählbar. Der Parameter *Lobby* gibt an, wieviel Prozent der Personen als Ziel- oder Startpunkt die Lobby haben. Der Parameter *hinauf–hinunter* gibt an, wie viel Prozent der Personen hinauf fahren. Der Komplementärwert zu diesem Parameter gibt damit an, wie viele Personen hinunter fahren wollen.

2 Die Simulation

Da die Simulation einem experimentellen Aufbau möglichst nahe kommen soll, ist sie als *Echtzeit*-Simulation ausgeführt. Diese Ausführung ermöglicht es zudem, die Interaktionen im Simulationsablauf mit Hilfe von *Threads* zu programmieren, was die Modellierung in vielen Fällen sehr viel eleganter macht (*Threads* sind unabhängig voneinander parallel ablaufende Teile eines größeren Programmes). Für Details der Implementierung siehe Abschnitt 4.

3 Die Strategien

Die Strategien werden als eigener *Thread* eingebunden. Bevor ein Aufzug ein Stockwerk weiterfährt, fragt er bei der Strategie nach, wie er sich im nächsten Stockwerk verhalten soll. Die Strategie teilt ihm dann mit, ob er anhalten, weiterfahren, die Richtung ändern soll usw.

Das Programm enthält vier Strategien, die im folgenden kurz beschrieben werden. Zwei der Strategien enthalten je zwei Untervarianten, die im Setupdialog ausgewählt werden können. Die *Random-Target-* und die *Nearest-Neighbor*-Strategie enthalten einen Parameter *Haltewahrscheinlichkeit*, mit dem ein Aufzug, der bereits mit anderen Passagieren unterwegs ist, für einen Fahrgast hält, der im aktuellen Stockwerk wartet und in die Aufzugsrichtung fahren möchte.

3.1 Simple Strategie

Im der simplen Strategie fahren alle Aufzüge stets von ganz unten bis ganz oben. Sie wechseln nur die Richtung, wenn sie im obersten bzw. untersten Stockwerk ankommen. Unterwegs halten die Aufzüge an, wenn es einen *hall call* gibt oder wenn ein Passagier aussteigen möchte.

Wartet kein Passagier, so wechseln alle Aufzüge in den Ruhestatus und fahren erst wieder an, wenn sie gebraucht werden. Für jeden zusätzlich wartenden Passagier wird dabei ein Aufzug gestartet, solange noch ruhende Aufzüge vorhanden sind.

Ein Nachteil beim simplen Algorithmus ist, dass Aufzüge oft fast gleichzeitig in einem Stockwerk ankommen. Dann erhält ein Aufzug alle Passagiere, der zweite Aufzug hält aber dennoch an. Dieses Verhalten wird in der Literatur als *bunching* bezeichnet [2].

3.2 Nearest-Neighbor

Die *Nearest-Neighbor*-Strategie verbessert das Verhalten, wenn ein Aufzug *leer* ist. In diesem Fall fährt der Aufzug den *hall call* an, der von seiner aktuellen Position am nächsten liegt, statt einfach in die Richtung weiterzufahren, aus der er gekommen ist.

Sind ein oder mehrere Aufzüge in Ruhe, so wird der Aufzug aktiviert, der dem *hall call* am nächsten liegt, und fährt diesen *hall call* an. Auch hier wechseln die Aufzüge in den Ruhestatus, wenn niemand wartet und sie leer sind.

Dieser Algorithmus ist im Programm in zwei Varianten implementiert, `NN_Stockwerk` und `NN_Zeit`, die sich in der Definition des nächsten *hall call*, also mathematisch gesprochen in der Abstandsfunktion unterscheiden. Die Stockwerk-Variante arbeitet mit dem Abstand der Stockwerke, die Zeit-Variante wählt den Fahrgast mit der längsten Wartezeit.

3.3 Random-Target

Ähnlich wie bei der *Nearest-Neigbor*-Strategie wird bei der *Random-Target*-Strategie das Verhalten bei einem leeren Aufzug verbessert. Wird der Aufzug leer, so sucht er sich ein neues Ziel aus, das noch nicht vergeben ist. Die Wahl des Ziels ist dabei zufällig bestimmt, was zum Namen dieser Strategie führt.

3.4 Sector

Beim *Sector*-Algorithmus werden die Stockwerke in (möglichst) gleich große Sektoren eingeteilt. Jedem Aufzug wird ein Sektor zugeteilt. Der Aufzug nimmt nur *hall calls* aus dem ausgezeichneten Stockwerk (in unserem Fall dem Erdgeschoss) und den ihm zugeteilten Sektoren an.

`SectorTyp1` teilt von unten in gleich große Sektoren, `SectorTyp2` verteilt die Stockwerke zyklisch. Vergleiche dazu Abbildung 1.

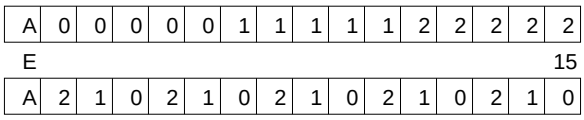

Abbildung 1: Stockwerkszuordung `SectorTyp1` (oben) und `SectorTyp2` (unten); A=alle Aufzüge; 15 Stockwerke

4 Die Implementierung

Die Implementierung ist komplett in JAVA™ 5.0 erfolgt. Für diese neue Version von JAVA™ wurde eine Vielzahl von Änderungen vorgenommen, etwa bei den im Projekt häufig verwendeten neuen Datentypen für Objektmengen und deren Behandlung im Multithread-Betrieb, weshalb die Entscheidung auf diese Plattform fiel. Für eine Zusammenstellung der Neuerungen in JAVA™ 5.0 siehe [5]. Als Entwicklungsumgebung wurde das frei erhältliche NETBEANS™ von SUN MICROSYSTEMS verwendet.

4.1 Die Struktur des Programms

Das Programm ist grob in drei Teile geteilt, die spezielle Klassen für die einzelnen Funktionen des Programms enthalten:

- `univie.mat.Aufzugsteuerung` enthält alle Datenstrukturen, die Personen oder Aufzüge betreffen sowie Methoden zur Initialisierung der Personen und der Aufzüge. Zudem werden hier die statistischen Daten gesammelt, ausgewertet und auf Wunsch gespeichert.
- `univie.mat.Aufzugsteuerung.strategie` enthält die Implementierung der Strategien.
- `univie.mat.Aufzugsteuerung.ui` enthält das *User Interface*, also die gesamte graphische Ausgabe des Programms.

Eine Übersicht hierzu zeigt die Grafik 2.

4.2 Objektorientierung und *Threads*

Die einzelnen Mitspieler der Simulation sind – den Ideen der objektorientierten Programmierung folgend – als Objekte implementiert. Weiters laufen viele Prozesse der Simulation parallel ab, weshalb vieles in separaten *Threads* programmiert ist. Das entspricht genau der Realität, wo ja ebenfalls die einzelnen Mitspieler gleichzeitig existieren. Beim Programmieren muss man darauf achten, einzelne Ereignisse, die zwei *Threads* verknüpfen, wie auch den Informationsaustausch und den gemeinsamen Zugriff auf Daten, im Hinblick auf Konkurrenzsituationen abzusichern. Man stelle sich etwa vor, zwei voneinander unabhängige

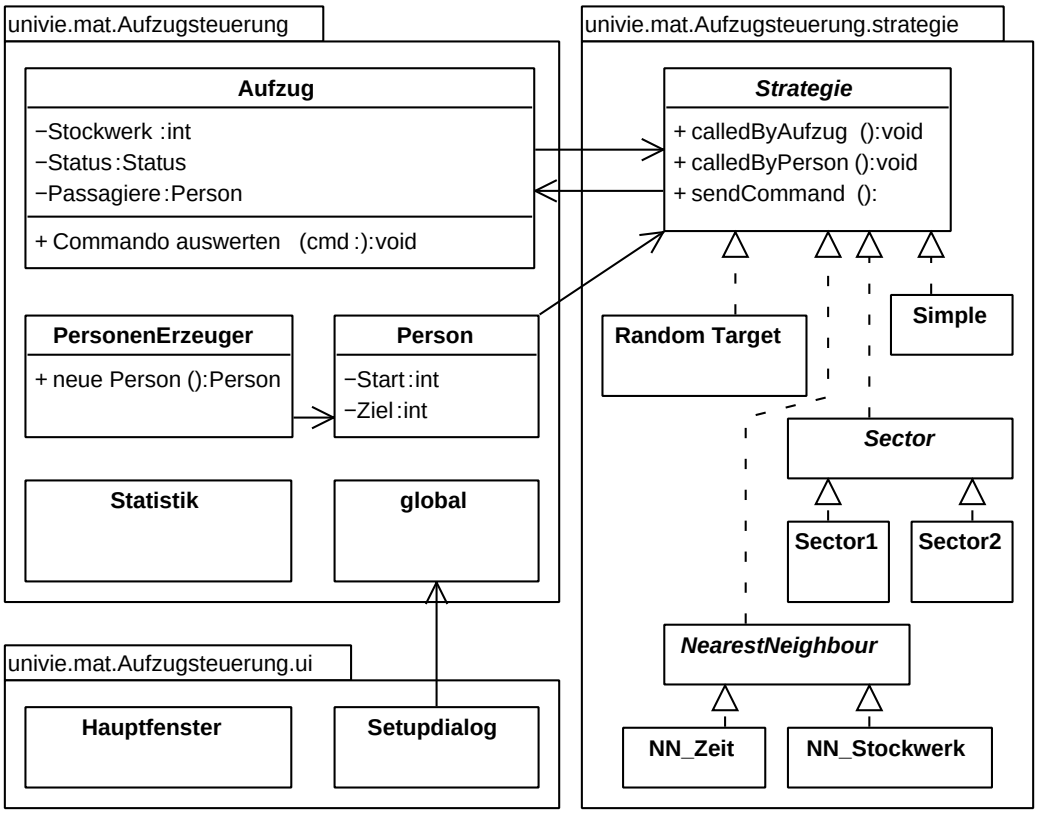

Abbildung 2: Grobes Klassendiagramm

Aufzüge wollen gleichzeitig ein und dieselbe Person einsteigen lassen: Das kann real nicht passieren und darf auch nicht zu Fehlern im Programm führen!

4.3 Kern der Simulation

Kern der Simulation bilden folgende drei Klassen:

- `Person` Diese Klasse bildet das Gerüst für jede Person, also jeden Fahrgast. Erzeugt werden diese entweder mittels eines *Threads*, der permanent einer Poissonverteilung folgend neue Personen erzeugt oder durch einen anderen *Thread*, der eine Datei einliest, die alle Personen inklusive Wartezeit gespeichert hat. Letzteres wird benötigt, um Simulationen wiederholt mit den selben Personendaten ablaufen lassen zu können, um verschiedene Algorithmen zuverlässig vergleichen zu können. Der Lebensablauf der Personen ist stets gleich und in Abbildung 3 dargestellt.

- `Aufzug` Für jeden Aufzug wird ein eigenes Objekt erstellt, das ein ausführbarer *Thread* ist. Jeder Aufzug läuft in einem separaten *Thread* und gemeinsam bilden sie einen Threadpool der in der Programmablaufsteuerungsklasse `runLogic` zentral

verwaltet wird. Über `runLogic` werden die Aufzüge beim Start der Simulation aktiviert.

- `Strategie` Da wir mehrere Strategien verwenden, gibt es dementsprechend mehrere Klassen. Jede Strategie wird dabei von einer abstrakten Strategieklasse abgeleitet. Alle Strategieklassen haben ein gemeinsames Interface. Je nachdem, welche Strategie man auswählt, wird dann in der schon oben erwähnten `runLogic`-Klasse ein zentrales Strategieobjekt mittels der ReflectionAPI erzeugt und der Strategie-*Thread* zu Beginn der Simulation gestartet.

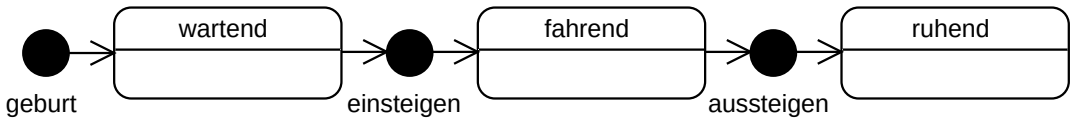

Abbildung 3: Lebensablauf einer Person

4.4 Koordination

Das Wechselspiel dieser Klassen ist natürlich keinesfalls einfach. Jeder *Thread* muss auf gemeinsame Daten (etwa die Anzahl der Stockwerke oder die wartenden Fahrgäste) und auf Methoden, die diese Daten betreffen, zugreifen und die Daten natürlich auch verändern können (z. B. nimmt ein Aufzug-*Thread* eine *wartende* Person auf, die dann in den Status *fahrend* versetzt wird. Deshalb kümmern sich die anderen Aufzug-*Threads* dann nicht mehr um diese Person). Dadurch kommt es zu einem permanenten Informationsfluss zwischen den *Threads*. Damit diese Vorgänge auch beobachtet und ausgewertet werden können, müssen parallel dazu noch laufend Berechnungen erfolgen, die dann über das Interface sichtbar werden. Erst dadurch sieht man, wo gerade jemand wartet oder komplexere Daten wie die momentane durchschnittliche Wartezeit oder das Histogramm der Statistik.

4.5 Die Steuerung

Der schwierigste Teil der Implementierung war die reibungslose und sinnvolle Steuerung der Aufzüge. Die Aufzüge selbst beinhalten dabei nur die rein physikalischen Vorgänge (z. B. Türe öffnen, dann Personen ein- und aussteigen lassen, diverse Verzögerungen, Stockwerkwechsel, Ruhezustand). Die Aufzüge benützen also keine global verfügbaren Informationen und empfangen ihre Befehle nur über die Strategie. Die Befehle der Strategie bestehen aus drei Teilen: dem nächsten Status des Aufzugs (ruhend oder fahrend), ob er im nächsten Schritt hinauf oder hinunter fahren soll oder im momentanen Stockwerk bleibt und ob er die Türe öffnen soll oder nicht. Diese Steuerinformation braucht der Aufzug in jedem Schritt – also mindestens in jedem Stockwerk – und verlangt sie auch ständig von der Strategie. Dabei ist zu bemerken, dass er die Anfrage nach einem neuen Befehl nicht erst

dann losschickt, wenn er sie braucht, sondern bereits etwas früher, bevor er ein Stockwerk weiterfahren muss. Dadurch hat die Strategie im separaten *Thread* genug Zeit, um den nächsten Befehl zu berechnen, und es gibt keine Verzögerungen für den Aufzug in der Simulation.

Die Strategie wird aber nicht nur durch die Aufzüge aufgerufen, sondern auch durch die neu entstehenden Personen (genau genommen durch das Erzeugen eines *hall calls*). Das ist notwendig, damit ein Aufzug über einen Befehl aktiviert werden kann, wenn alle Aufzüge in Ruhe sind und die Strategie nicht befragen.

Die Kommunikation zwischen Strategie und Aufzügen erfolgt über *MessageQueues*, die als `LinkedBlockingQueue` implementiert sind (vgl. [6]). Jeder Aufzug verfügt über eine eigene Queue, die, wenn sie leer ist, die Ausführung des Aufzugs-*Threads* blockiert. Die Strategie wird über eine einzige MessageQueue aufgerufen, indem das aufrufende Objekt eine Referenz auf sich selbst in die Queue stellt. Die Strategie arbeitet diese Queue, wenn sie nicht leer ist, der Reihe nach ab (FIFO, first-in-first-out). Dabei muss die Strategie zuerst filtern, ob sie von einer neuen Person oder einem Aufzug aufgerufen wurde, und dann wird gemäß der instanzierten Strategieart der Algorithmus aufgerufen (Abbildung 4).

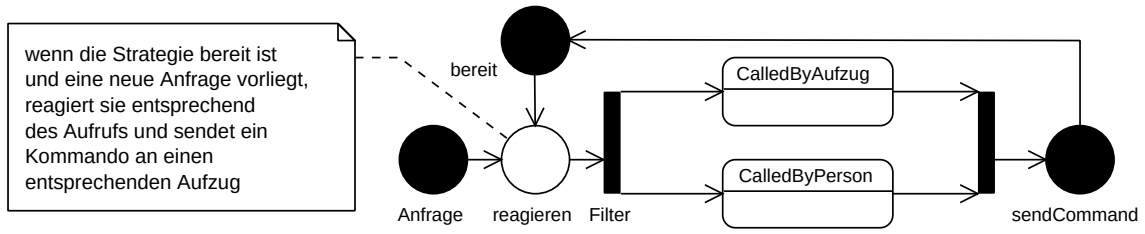

Abbildung 4: Ablaufdiagramm für die Strategie

Als Verbesserung dieser Implementierung kann man sich überlegen, diese Queue mit Prioritäten zu versehen, so dass die Aufrufe der Aufzüge wichtiger sind und vor denen der Personen behandelt werden müssen. Allerdings läuft das Programm offensichtlich auch ohne Priorisierung zuverlässig.

4.6 Ausgabe

Als Hilfe zum Beurteilen der Vorgänge gibt es verschiedene Ausgabemöglichkeiten. Diese sind in der Klasse `Statistik` und beinhalten eine Anbindung an GnuPlot [7] für grafische und PostScript Ausgaben, die Ausgabe eines Simulationsprotokolls in einer Textdatei und eine Auflistung aller abgehandelten Personen in eine `CSV`-Datei. Dadurch kann man im nachhinein sehr flexibel die gewonnen Daten analysieren.

4.7 javadoc

Für weitergehende Informationen empfiehlt es sich das javadoc zu benützen. Ein Großteil der Klassen und Methoden ist dort genau erklärt.

5 Die Statistik

Um die einzelnen Strategien vergleichen zu können, werden die Warte- und Fahrzeiten für die einzelnen Personen, ihr Durchschnittswert (d. h. statistisch gesehen der Stichproben-Erwartungswert) und die Stichprobenvarianz gesammelt. Mit Hilfe dieser Werte kann man sowohl die einzelnen Varianten miteinander vergleichen, als auch den Parameter für die einzelnen Strategien optimal bestimmen.

5.1 Szenarien

Für beide Zwecke ist die Auswahl typischer Szenarien nötig, für die eine Simlation erfolgt. Wir haben folgende Szenarien als Testfälle erstellt:

S1: Wohn-Hochhaus

S2: Büro-Hochhaus

S3: Institutsgebäude

S4: Altbauwohnung

mit verschiedenen Aufzugsparametern:

Szenario	S1	S2	S3	S4
Stockwerke	20	40	6	7
Lifte	6	10	8	1
Personen/Stockwerk	40	100	60	5
Tür auf und zu [s]	1	1	1.5	2
einsteigen/Person [s]	2	2	2	2
aussteigen/Person [s]	1.5	1.5	1.5	2
Zeit/Stockwerk [s]	0.5	0.5	1.5	3
Verzögerung hinauf/hinunter [s]	0.5	0.5	2	0.1
Kapazität/Lift	6	15	5	3

Tabelle 1: Szenarien und Aufzugsparameter

Zu diesen Szenarien kommen ebensoviele Muster von Verkehrsaufkommen:

P1: normal

P2: hoch

P3: nur hinunter

P4: nur hinauf

mit der zugehörigen Tabelle der Verkehrsparameter:

Muster	P1	P2	P3	P4
Fahrender Anteil der Gesamtpopulation	10%	50%	100%	100%
Hinauf–hinunter	50%	50%	100%	0%
Lobby	1/Stockwerkzahl	50%	100%	100%

Tabelle 2: Verkehrsmuster und -parameter

5.2 Latin Squares

Als Latin Square oder lateinisches Quadrat mit Seitenlänge p bezeichnet man eine quadratische Anordnung von p Buchstaben derart, dass in jeder Reihe und Spalte jeder Buchstabe nur einmal auftritt.

A	B	C	D
D	A	B	C
C	D	A	B
B	C	D	A

Abbildung 5: Beispiel: 4×4–Latin Square

Eine solche Anordnung von Buchstaben kann bei faktoriellen Experimenten, wie in unserem Fall, die Anzahl der nötigen Experimente stark verkleinern. Statt $4 \times 4 \times 4$ (4 Algorithmen, 4 Umgebungs-Setups, 4 Personen-Setups) genügte es 4×4 Experimente durchzuführen.

Die Idee bei der Verwendung von Latin Squares ist folgende: Jeder Algorithmus (entsprechend den Buchstaben) „sieht“ jedes Personen-Setup und jedes Gebäude-Setup (entsprechend Zeilen und Spalten) ein Mal. Durch die Anordnung im Latin Square werden dabei systematische Fehler vermieden (etwa dass bei einem Setup das Verkehrsaufkommen höher ist und dieses immer auf den selben Algorithmus trifft).

Latin Squares sind ein wertvolles Hilfsmittel in der Planung von Experimenten und deren statistischer Auswertung, vgl. [4].

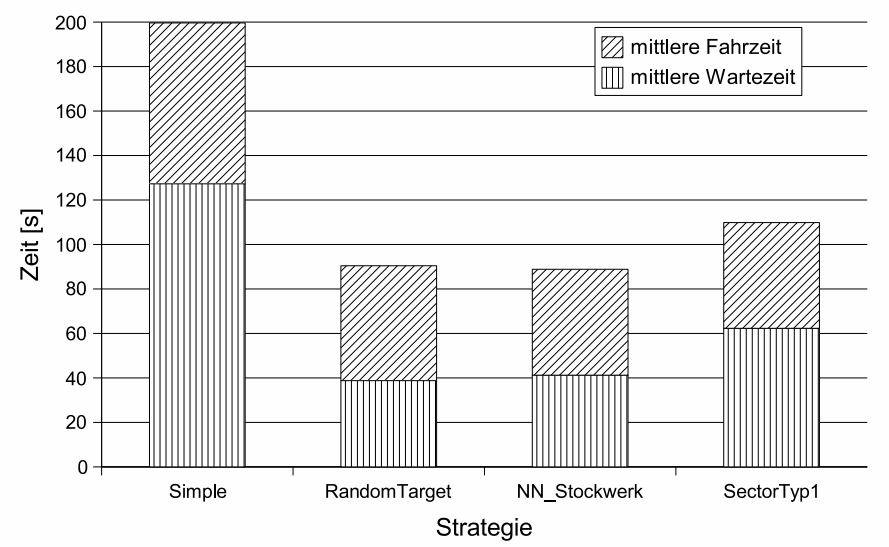

Abbildung 6: Vergleich der Warte- und Fahrzeit für die Algorithmen

5.3 Vergleich der verschiedenen Algorithmen

In einem Latin-Square wurde eine entsprechende Anzahl an Testläufen durchgeführt. Eine genaue Beschreibung des durchgeführten Versuchs gibt Anhang A [8]. Es wurden die vier Personen-Setups zusammen mit den vier Umgebungs-Setups und den Algorithmen `Simple`, `RamdomTarget`, `NN_Stockwerk` und `SectorTyp1` getestet. Aussagen über die Güte der Algorithmen geben die Summen der Warte- und Fahrzeiten aus den vier Versuchsdurchgängen, die in Abbildung 6 graphisch dargestellt sind.

Wie man sofort sieht, führt insbesondere die simple Strategie nicht zu zufriedenstellenden Ergebnissen. Durch das ineffiziente Verhalten benötigen die Aufzüge entscheidend länger als in den anderen Algorithmen. *Nearest-Neighbor* und *Random-Target* schneiden wesentlich besser ab. Gerade im heikelsten Szenario, dem Peak-Down-Traffic (bei dem die gesamte Population des Hauses nur abwärts fahren möchte) führen beide Algorithmen zu wesentlich verbesserten Werten.

Interessanterweise schneidet der *Nearest-Neighbor*-Algorithmus nicht besser ab als der *Random-Target*-Algorithmus. Zwar hat der *Nearest-Neighbor*-Algorithmus in der Regel die kürzeren Fahrzeiten, dafür tritt aber der Effekt des bunching auf, d. h. die Aufzüge fahren oft nah beieinander und verlieren dadurch an Effizienz.

Der getestete *Sector*-Algorithmen liegt nicht schlecht, verliert jedoch gegenüber *Random-Target* und *Nearest-Neighbor* dadurch an Effizienz, dass durch die starre Zuordnung von Stockwerken oftmals Personen nicht mitgenommen werden können, die auf dem Weg des Aufzugs liegen. Diese Aussage wird auch durch die Bestimmung des optimalen Parameters im nächsten Abschnitt untermauert:

5.4 Bestimmung des optimalen Parameters

Ein weiteres Ziel des Projekts war es, einen Parameter optimal zu bestimmen. Dieser Parameter gibt die Wahrscheinlichkeit an, mit der ein Aufzug für einen Fahrgast hält, der „auf dem Weg liegt“ (z. B. Aufzug 0 ist mit einem Passagier von Stockwerk 10 auf dem Weg ins Erdgeschoss und ist auf Höhe von Stockwerk 5, wo ein Passagier darauf wartet, nach Stockwerk 2 zu fahren. Der Aufzug hält für diesen Passagier dann mit der festgelegten Wahrscheinlichkeit an oder nicht).

Steigt die Mitnahmewahrscheinlichkeit, so werden die Passagiere auf dem Weg öfter mitgenommen. Da bei Mitnahmewahrscheinlichkeit 0 der Aufzug immer nur einen Passagier ans Ziel bringt und an allen anderen vorbeifährt, ist zu erwarten, dass die Wartezeit zunimmt.

Andererseits führen die häufigen Stopps natürlich zu längerer Fahrzeit.

Über die Summe aus Warte- und Fahrzeit, die ja für einen Passagier am wichtigsten ist, kann zunächst keine Aussage getroffen werden.

Mit einem zweiten Experiment (Anhang B, [9]) sollte nun der Parameter hierfür optimal bestimmt werden.

Aufgrund der Ähnlichkeit der *Random-Target* und der *Nearest-Neighbor*-Strategie und ihrem sehr ähnlichen Abschneiden reicht es, diesen Parameter für die *Random-Target*-Strategie zu optimieren, um Rückschlüsse für die andere zu erhalten. Abhängig vom Parameter erhält man Abbildung 7 als Grafik für den Zusammenhang von Parameterwerten und Warte- und Fahrzeiten.

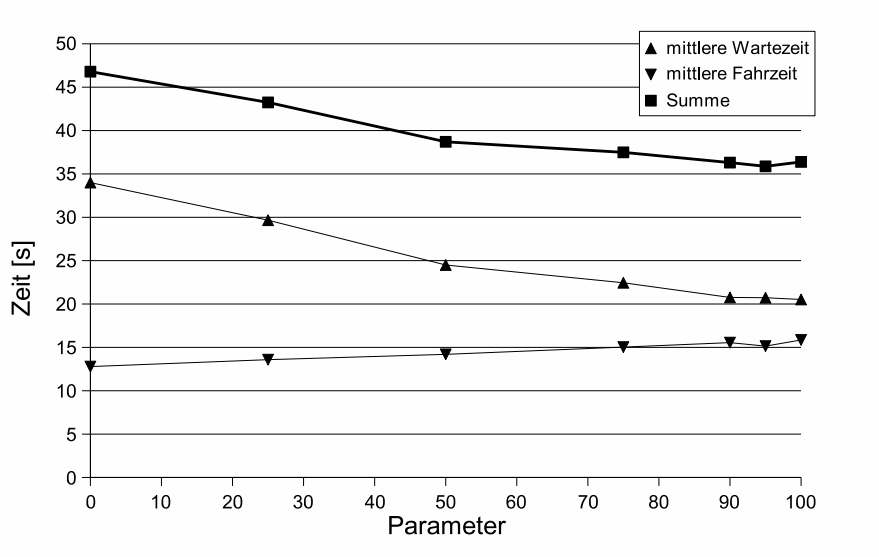

Abbildung 7: Warte und Fahrzeiten nach Parameter

Man sieht also, dass der Parameter mit etwa 95% optimal gewählt ist, um die Summe von von Fahr- und Wartezeit zu minimieren. Während die Wartezeit tatsächlich mono-

ton abnimmt, hat die Fahrzeit bei ungefähr 95% ein Minimum, das auch auf die Summe durchschlägt und somit zum optimalen Wert von 95% führt. Aufgrund der geringen Unterschiede zu 100%, also der Mitnahme aller auf dem Weg liegenden Passagiere, lässt sich mit geringer Vereinfachung sagen, dass man auf die Implementierung dieses Parameters in der Regel verzichten kann und alle auf dem Weg liegenden Passagiere mitnehmen sollte, falls es einem auf ein effizientes Verhalten in Situationen mit hohem Verkehrsaufkommen ankommt.

5.5 Ergänzende Bemerkungen

In einer vorhergehenden Version des Latin Squares wurden statt des `NN_Stockwerk` der `NN_Zeit`-Algorithmus verwendet. Dadurch kam es zu einer geringfügigen Verschiebung in Richtung geringerer Warte- und erhöhter Fahrzeit. Was die Systemzeit, also die Summe von Fahr- und Wartezeit angeht, war praktisch keine Änderung festzustellen.

Durch die Berücksichtigung mehrerer verschiedener Setups und die Bildung der Summe der Fahr- und Wartezeiten über alle Setups gelten alle oben erhaltenen Ergebnisse natürlich nicht für einen konkreten Einzelfall, sondern als Maß dafür, für welche Algorithmen man im Allgemeinen eine gute Performance erwarten darf. Weiß man bereits, dass zum weit überwiegenden Teil nur ein bestimmtes Verkehrsaufkommen auftritt, so lohnt es sich eventuell, den Vergleich der Algorithmen für ausschließlich diese Umweltbedingungen durchzuführen. Für das Ein-Aufzug-System im Wohnhaus bei geringem Verkehrsaufkommen werden etwa alle Algorithmen ähnliche Werte liefern.

6 Fazit und mögliche Verbesserungen

Unter den von uns getesteten Standardalgorithmen schneiden `NN_Stockwerk` und`Random-Target` und am besten ab. Zwar war hier auch der Programmieraufwand am höchsten, die erhaltenen Ergebnisse rechtfertigen allerdings den höheren Aufwand für die Implementierung gerade in Situationen mit hohem Verkehrsaufkommen.

Wie sich in der Untersuchung des optimalen Parameters der Mitnahmewahrscheinlichkeit zeigt, ist es für Situationen mit mittlerem bis hohem Verkehrsaufkommen am günstigsten, alle auf dem Weg eines Aufzugs liegenden Passagiere, die in die aktuelle Aufzugsrichtung fahren wollen, mitzunehmen. Da für Situationen mit niedrigem Verkehrsaufkommen die Aufzüge ohnehin effizient genug arbeiten, kann also auf die Implementierung dieses Parameters in der Regel verzichtet werden. Der Parameter liefert aber, wie in Abschnitt 5.3 gezeigt, eine gute Erklärung für das schlechtere Abschneiden des *Sector*-Algorithmus.

Das Projekt war wesentlich aufwändiger als zunächst angenommen. Gerade die Implementierung der Simulation und der Strategien nahm sehr viel Zeit in Anspruch. Es war deshalb nicht möglich, einige in der Literatur vorgestellte Algorithmen zu implementieren, etwa die in [2], [3] vorgeschlagenen. Zwar sind unsere Ergebnisse mit den in [3] erwähnten nicht direkt vergleichbar, weil die vorliegende Arbeit für andere und eine geringere Zahl von Verkehrsaufkommen und Gebäudetypen getestet wurden. Der in [3] erwähnte LQF-

Algorithmus entspricht aber in etwa unserem `NN_Zeit`, und hat dort eine etwa 10% höhere Wartetzeit als der vorgeschlagene Ansatz auf der Basis von neuronalen Netzen. Man darf also bei der Implementierung komplizierterer Algorithmen mit weiter verbesserten Werten rechnen, allerdings auf Kosten einer wesentlich erhöhten Komplexität des Programms.

Die Konzeption des Programms war von Beginn an auf vergleichsweise einfache Szenarien ausgelegt, um den Aufwand nicht zu sehr zu erhöhen. So sind Spezialfälle wie das in Japan gängige Aufzugmodell, bei dem man beim Absetzen eines *hall calls* schon erfährt, mit welchen Aufzug man fahren muss (weshalb diese Zuordnung sofort berechnet und später ständig berücksichtigt werden muss) oder der steuerbare Ausfall eines Aufzugs nicht eingebaut. Auch auf die Aufteilung von Abschnitten des Gebäudes auf Aufzüge, etwa, dass manche Aufzüge nur die untere Hälfte der Stockwerke des Gebäudes bedienen und manche ausschließlich die oberen (und nicht das Erdgeschoss!) wurde verzichtet.

Weitere Verbesserungen des Programms wären möglich: Die Versuche, die bisher selbst geplant und nachher separat ausgewertet werden müssen, ließen sich automatisieren, so dass man nach Vorgabe von Gebäude- und Verkehrsaufkommen-Setups bereits ein fertiges Ergebnis erhält und somit leichter kleine Änderungen durchführen kann. Weiters könnte man sich überlegen, nicht nur Einzelpersonen, sondern auch Gruppen von Personen zu erzeugen, wie sie im realen Leben auftreten: Oft fahren Teams etwa gleichzeitig in einen Besprechungsraum. Solche Fälle sind im Programm, aber auch in der Literatur, bisher nicht berücksichtigt. Schließlich behandelt unsere Versuchsauswertung nur die durchschnittliche Fahr- und Wartezeit. Eventuell sind aber auch weitere Werte wie maximale Fahr- und Wartezeiten, der Prozentsatz von Personen, die länger als eine vorgegebene Zeit warten oder der Stromverbrauch des Systems interessant. Eine Vielzahl solcher zusätzlicher Werte wurde für den Versuch nicht ausgewertet – unser Hauptaugenmerk lag auf dem Vergleich der durchschnittlichen Effizienz der Aufzüge – ist aber im Programm schon implementiert. Eine Erweiterung in dieser Richtung ist also sehr leicht möglich.

Abschließend lässt sich sagen, dass der Vergleich verschiedener Strategien zur Aufzugsteuerung mittels Simulation durchaus sinnvoll ist, weil es hohe Unterschiede in der Effizienz der verschiedenen Strategien gibt. Über die durchschnittliche Systemzeit (also die Summe von Fahr- und Wartezeit) lässt sich so schon im Vorfeld eine Aussage treffen und Verbesserungen in den Strategien sind leicht möglich.

Diese Dokumentation, die genaue Beschreibung der durchgeführten Versuche, das Programm und ein Präsentationsfoliensatz finden sich auf [1].

Literatur

[1] `http://www.aufzugsteuerung.at.tt`

[2] A. Rong et al. *Estimated Time of Arrival (ETA) Based Elevator Group Control Algorithm with More Accurate Estimation.* TUCS Report No 584, Turku Center for Computer Science, 2003

[3] R. H. Crites und A. G. Barto. *Elevator Group Control Using Multiple Reinforcement Learning Agents.* Machine Learning 33 (1998), 235–262.

[4] Peter W. M. John. *Statistical Design and Analysis of Experiments.* siam, Philadelphia 1998.

[5] `http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html`

[6] `http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/LinkedBlockingQueue.html`

[7] `http://www.gnuplot.info/`

[8] Versuchsprotokoll 1 (Vergleich der Algorithmen).[1]

[9] Versuchsprotokoll 2 (Bestimmung des optimalen Parameters).[1]

[1] verfügbar auf `http://www.aufzugsteuerung.at.tt`