Automatisierte Simulationsverteilung für SimpleSim

Diplomarbeit
von
Roland Krüger
aus
Dresden

vorgelegt am
Lehrstuhl für Praktische Informatik IV
Prof. Dr. W. Effelsberg
Fakultät für Mathematik und Informatik
Universität Mannheim

Juni 2006

Betreuer: Dipl.-Wirtsch.-Inf. Holger Füßler
Dipl.-Inf. Matthias Transier
Dipl.-Wirtsch.-Inf. Thomas King

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mannheim, den 30. Juni 2006 Roland Krüger

Danksagung

Die Erstellung einer Diplomarbeit ist ein Unterfangen, das in seinem ganzen Umfang natürlich nicht ohne die Hilfe und Unterstützung Dritter möglich ist. Daher möchte ich an dieser Stelle meinen Dank an die folgenden Personen richten.

Zunächst möchte ich mich bei den Betreuern meiner Arbeit am Lehrstuhl für Praktische Informatik IV Holger Füßler, Matthias Transier und Thomas King bedanken. In vielen fruchtbaren Gesprächen konnten ihre Anregungen und Tipps mir wertvolle Impulse geben, die einen wesentlichen Beitrag zur Entstehung der vorliegenden Arbeit leisteten.

Weiter gilt mein Dank meinen Eltern und Großeltern für ihre fortwährende Unterstützung. Durch den Rückhalt, den ich während meines Studiums durch meine Familie erfahren habe, ist es mir möglich gewesen, mich konzentriert und ungestört den mir gestellten Aufgaben zu widmen.

Meiner Freundin Susanne Koudela möchte ich besonders für ihre Geduld und ihre uneingeschränkte Unterstützung danken, die sie mir über die gesamte Zeit hinweg entgegenbrachte. Sie hatte für meine aufgabenbezogenen Probleme immer ein offenes Ohr, so dass man mit ihr auch ergiebig über für sie eigentlich fachfremde Dinge diskutieren konnte.

Inhaltsverzeichnis

Abbildungsverzeichnis iii

Tabellenverzeichnis v

Algorithmenverzeichnis vii

Abkürzungsverzeichnis ix

1 Einführung 1

1.1 Problemstellung . . . 2
1.2 Überblick über den weiteren Aufbau dieser Arbeit . . . 3

2 Grundlagen 7

2.1 Diskrete Ereignissimulationen . . . 7
2.1.1 Bedeutung des Zufallsmoments für Simulationen . . . 10
2.1.2 ns-2 und SimpleSim — Ein Vergleich . . . 11
2.1.3 SimpleSim als Simulator für diskrete Ereignissimulationen . . . 14
2.1.4 Beschaffenheit von Simulationsaufträgen . . . 15
2.2 Maschinen-Scheduling . . . 16
2.2.1 Klassifikation von Scheduling-Problemen . . . 18
2.2.2 Algorithmen für das Scheduling . . . 21

3 Konzeption und Analyse 37

3.1 Leistungsparameter der Scheduling-Algorithmen . . . 37
3.1.1 Vorbemerkungen . . . 37
3.1.2 Branch And Bound . . . 39
3.1.3 Greedy Scheduling . . . 45
3.1.4 Simulated Annealing . . . 46
3.1.5 List Scheduling . . . 48
3.1.6 Schlussfolgerungen . . . 49
3.2 Grundlegende Arbeitsweise der Simulationsumgebung . . . 49
3.2.1 Herkömmliche Methode der Simulationsdurchführung . . . 50
3.2.2 Simulation mit Hilfe von SimpleGrid . . . 50
3.3 Scheduling von Simulationsaufträgen . . . 52
3.3.1 Klassifikation des vorliegenden Problems . . . 52
3.3.2 Begriffsklärung . . . 53

3.3.3 Schwierigkeiten beim Einplanen von Simulationsjobs auf freie Rechner . . . 55
3.3.4 Kapazitätsmessung der verfügbaren Rechner . . . 58
3.3.5 Komplexitätsmessung der Simulationsläufe . . . 59
3.3.6 Behandlung des Speicherbedarfs und Erkennung von exzessivem Memory Paging . . . 60
3.4 Einsatz der Scheduling-Algorithmen für die Simulationsverteilung . . . 61

4 Implementierung 65
4.1 Anforderungen an die Simulationsumgebung . . . 65
4.1.1 Transparenz . . . 65
4.1.2 Einfachheit . . . 66
4.1.3 Fehlertoleranz und Ausfallsicherheit . . . 66
4.1.4 Skalierbarkeit . . . 66
4.2 Architektur von SimpleGrid . . . 66
4.2.1 Gridserver . . . 67
4.2.2 Gridnode . . . 68
4.2.3 Client Tool . . . 68
4.2.4 Weitere Designentscheidungen für die Softwareimplementierung . 69
4.3 Arbeitsweise von SimpleGrid . . . 70
4.3.1 Durchführung von Benchmarks . . . 70
4.3.2 Scheduling von Simulationsaufträgen . . . 72
4.3.3 Schätzung der voraussichtlichen Simulationslaufzeiten . . . 74
4.3.4 Umgang mit Fehlersituationen . . . 78

5 Erweiterungsmöglichkeiten und Fazit 81
5.1 Komplexitätsschätzung durch Deduktion . . . 81
5.2 Vereinfachung des Schedulings durch Preemption . . . 81
5.3 Quality of Service und Benutzerverwaltung . . . 82
5.4 Einführung von Sicherheitsaspekten . . . 83
5.5 Messung der verfügbaren Netzwerkbandbreite für die Berücksichtigung beim Scheduling . . . 83
5.6 Fazit . . . 83

A Berechnung der Untergrenze für die Branch And Bound-Methode 87

B Konfiguration der SimpleGrid-Module 93
B.1 Gridserver . . . 93
B.2 Gridnode . . . 95
B.3 Client Tool . . . 97

C Anwendungsbeispiel für die Durchführung eines Simulationsprojekts 99

Literaturverzeichnis 103

Abbildungsverzeichnis

1.1 Organisation dieser Arbeit 4

2.1 Voranschreiten der Simulationszeit zum jeweils nächsten Ereignis 9
2.2 Voranschreiten der Simulationszeit um einen festen Betrag Δt mit den Ereignissen $e_1, \ldots, e_4$ 9
2.3 Beispiel für ein Gantt-Diagramm mit 9 Aufträgen $J_1, \ldots, J_9$ und 3 Maschinen $M_1, \ldots, M_3$ 17
2.4 Beispiel einer Probleminstanz für $P \mid prec; p_i = 1 \mid C_{max}$ 21
2.5 Beispiel für die Entwicklung der Zielfunktion beim Simulated Annealing 27
2.6 Beispiel für die Entwicklung der Zielfunktion bei der Branch And Bound-Methode 30
2.7 Beispielproblem für den Branch And Bound-Algorithmus 33
2.8 Beispiel für den Greedy Scheduling-Algorithmus 34

3.1 Ausführungsdauer und Fehlermaß für den Branch And Bound-Algorithmus bei kleinen Probleminstanzen. Die Startlösung wurde mit dem Zufallsscheduler berechnet. 41
3.2 Fehlermaß für den Branch And Bound-Algorithmus bei mittleren Probleminstanzen und unterschiedlichen Startlösungsalgorithmen 42
3.3 Fehlermaß für mittlere Probleminstanzen und dem Greedy Scheduler als Initialisierer für die Branch And Bound-Methode 42
3.4 Rechendauer der Branch And Bound-Methode mit dem Simulated Annealing-Verfahren als Initialisierer bei mittelgroßen Probleminstanzen 42
3.5 Fehlermaß für den Branch And Bound-Algorithmus bei großen Probleminstanzen und unterschiedlichen Startlösungsalgorithmen 43
3.6 Fehlermaß für den Branch And Bound-Algorithmus bei großen Probleminstanzen und dem Greedy Scheduler als Initialisierungsalgorithmus . . 44
3.7 Rechendauer der Branch And Bound-Methode mit dem Simulated Annealing-Verfahren als Initialisierer bei großen Probleminstanzen . . . 44
3.8 Leistungsparameter der Branch And Bound-Methode bei einem Toleranzwert von 2% 44
3.9 Fehlermaß für den Greedy Scheduling-Algorithmus bei kleinen und mittelgroßen Probleminstanzen 45
3.10 Fehler des Greedy Schedulers für große Problemgrößen 45
3.11 Leistungsparameter des Simulated Annealing-Verfahrens bei kleinen Problemgrößen 47

3.12 Leistungsparameter des Simulated Annealing-Verfahrens bei mittleren Problemgrößen 47
3.13 Leistungsparameter des Simulated Annealing-Verfahrens bei großen Problemgrößen 48
3.14 Vorgang bei der Verteilung eines Simulationsauftrags 63

4.1 Beispiel für die Organisation einer SimpleGrid-Installation 67
4.2 Schema für die Durchführung eines Simulationsauftrags 69
4.3 Messungen der Kapazitätswerte zweier Rechner 71
4.4 Flussdiagramm für die Simulationsverteilung des Schedulers. Hier ist der Fall dargestellt, der eintritt, wenn ein Simulationsrechner den nächsten Simulationslauf anfordert. 75
4.5 Flussdiagramm für die Entscheidungen, die nach Beendigung des Scheduling-Vorgangs für die außer der Reihe zugewiesenen Simulationsläufe getroffen werden. 76

A.1 Variablen für die Berechnung der Untergrenze einer Teillösung des Branch And Bound-Verfahrens 88
A.2 Der Teilschedule S_3 des Branch And Bound-Algorithmus in der dritten Suchbaumebene 89
A.3 Teilschedule S_3^1 des Suchbaums 90
A.4 Zweites Beispiel für die Berechnung der Untergrenze 91

Tabellenverzeichnis

2.1 Zielfunktionen für das Optimalitätskriterium eines Scheduling-Problems 21

3.1 Beispiel einer Konfigurationsdatei für SimpleSim 54
3.2 Beispiel für die Distanz zweier Parametertupel mit $\delta_P(P_1, P_2) = 2$ 55

4.1 Mittelwerte und Standardabweichungen verschiedener Benchmarkkonfigurationen 72
4.2 Mittelwert, Standardabweichung und prozentuale Standardabweichung von Komplexitätswerten 74

C.1 Schablone für die Konfigurationsdateien des Anwendungsbeispiels 100
C.2 Konfigurationsdateien für das Anwendungsbeispiel 101

Algorithmenverzeichnis

1 Lokale Suche 23
2 Simulated Annealing 26
3 Branch And Bound 29
4 Greedy Scheduling 34
5 List Scheduling 35

Abkürzungsverzeichnis

AODV	Ad-hoc On-demand Distance Vector Routing
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
DARPA	Defense Advanced Research Projects Agency
DSR	Dynamic Source Routing
HDD	Hard Disk Drive
IP	Internet Protocol
JAR	Java Archive
LAN	Local Area Network
MANET	Mobile Ad-Hoc Network
ns	Network Simulator
NIO	New Input/Output
OTcl	Object Tool Command Language
QoS	Quality of Service
REAL	Realistic And Large
RMI	Remote Method Invocation
TCP	Transmission Control Protocol
VINT	Virtual InterNetwork Testbed
WAN	Wide Area Network
XML	Extensible Markup Language

Kapitel 1

Einführung

Der Entwicklung und Untersuchung von Kommunikationsprotokollen für Computernetzwerke wird ein breites wissenschaftliches Interesse entgegengebracht. Für die Beurteilung der Leistungsfähigkeit solcher Protokolle ist es notwendig, diese in realistischen, anwendungsbezogenen Umgebungen zu testen und auszuwerten. Eine Analyse von Kommunikationsprotokollen mit Hilfe realitätsnaher Tests setzt eine entsprechende Infrastruktur voraus, mit der sich die erforderlichen Testumgebungen konstruieren lassen. Eine solche Infrastruktur muss mindestens aus eine Menge von Hardware-Instanzen, wie z. B. den Netzwerkknoten, und den entsprechenden Software-Implementierungen der untersuchten Protokolle bestehen. Darauf aufbauend lassen sich beliebige Testsituationen gestalten und untersuchen. Es stellen sich hierbei für den Forscher jedoch eine ganze Reihe von Problemen. Zum einen ist es während der Entwicklung neuartiger Protokolle nicht immer möglich, einsatzfähige Software-Implementierungen für die verwendete Hardware zu erstellen. Dies kann bspw. schon daran scheitern, dass die eingesetzten Geräte keine Unterstützung dafür anbieten, ihre internen Steuerprogramme zu verändern. Zum anderen besteht der gravierendste Nachteil solcher realitätsnahen Tests in der Tatsache, dass sie in den meisten Fällen einen erheblichen Kosten- und Zeitaufwand erzeugen. So wäre z. B. die Durchführung von Experimenten mit drahtlosen Ad-Hoc-Netzwerken, deren Netzknoten von Fahrzeugen gebildet werden, nur unter der Verfügbarkeit eines entsprechenden Fuhrparks und eines Testgeländes möglich. Man sieht leicht, dass dies für die Entwicklung von Kommunikationsprotokollen in der Regel nicht praktikabel ist.

Um nun umfangreiche, vielfältige und gleichzeitig kostengünstig und in kurzer Zeit ermittelbare Untersuchungsergebnisse erhalten zu können, greift man daher auf Netzwerksimulationen zurück. Mit ihnen bildet man die gesamte Versuchsanordnung auf entsprechende logische Objekte eines Netzwerksimulators ab. Der Simulator dient dann als Steuerungsumgebung für die durchgeführten Simulationen. Er bildet die Umwelt, in der eine Simulation durchgeführt werden soll, weitgehend realitätsnah nach und ermöglicht somit Ergebnisse, die dem tatsächlichen Verhalten der Protokolle in realistischer Umgebung ausreichend nahe kommen.

Eine solche Nachbildung der Umwelt innerhalb eines Computerprogramms bietet gegenüber der direkten Untersuchung realer Versuchsanordnungen eine ganze Reihe von Vorteilen. Neben einer deutlichen Verringerung der Kosten und des Zeitaufwands für die Untersuchungen, ist vor allem eine weitaus größere Flexibilität hervorzuheben. Es ist problemlos und schnell möglich, einzelne Parameter einer Simulation zu variieren, den Simulationsaufbau kurzfristig abzuändern oder Modifikationen der grundlegenden

Algorithmen vorzunehmen. Somit lassen sich in relativ kurzer Zeit und mit geringem Aufwand umfangreiche Untersuchungen anstellen, die sonst nur sehr schwierig durchzuführen wären.

Im wissenschaftlichen Betrieb hat sich der Netzwerksimulator ns-2 etabliert [29]. Dieser wird als Gemeinschaftsprojekt u. a. durch das VINT-Projekt der DARPA getragen. Ns-2 ermöglicht die Durchführung von Simulationen mit einer Vielzahl unterschiedlicher Kommunikationsprotokolle und verschiedener Typen kabelloser und kabelgebundener Netzwerke. Ns-2 ist ein Simulator für diskrete Ereignissimulationen [24]. Dies bedeutet, dass hier die nachgebildete Umwelt nicht kontinuierlich mit der Zeit simuliert wird. Es werden vielmehr eine endliche Menge von verschiedenen Ereignissen erzeugt, die während einer Simulation zu bestimmten, festgelegten Zeitpunkten zur Ausführung kommen.

Ns-2 zeigt trotz seiner vielfältigen Möglichkeiten eine Reihe von Schwächen. Aus diesem Grund wird am Lehrstuhl für Praktische Informatik IV an der Universität Mannheim ein alternativer Simulator mit dem Namen SimpleSim entwickelt, der sich ebenfalls für die Simulation von Computernetzwerken einsetzen lässt, dabei aber die Nachteile des ns-2 vermeiden möchte.

1.1 Problemstellung

Die vorliegende Arbeit behandelt SimpleGrid, eine Ergänzung für SimpleSim zur automatisierten Simulationsverteilung in einem Rechnerverbund.

Die Durchführung von Netzwerksimulationen ist üblicherweise mit einem sehr hohen Rechenaufwand verbunden. Dies liegt darin begründet, dass zum einen ein einzelner Simulationsrechner die gesamte Versuchsanordnung koordinieren und berechnen muss. So wird z. B. ein großer Teil der Rechenkapazität dafür verwendet, die Protokollstacks jedes einzelnen Netzwerkknotens in einer Simulation zu verwalten. Zusätzlich ist der Simulationsrechner für die Berechnung der Umgebung verantwortlich, in der sich die logischen Netzknoten befinden. Es müssen also auch die physischen Gegebenheiten der Testumgebung simuliert werden, wie z. B. das Übertragungsmedium für den Datenverkehr.

Eine weitere Ursache für den hohen Rechenaufwand besteht in der Tatsache, dass man üblicherweise viele einzelne Parameter einer Simulation variiert. Man möchte dadurch herausfinden, wie sich diese auf das Verhalten der Kommunikationsprotokolle auswirken. Zusätzlich führt man Simulationen mit dem gleichen Parametersatz auch zumeist mehrmals in Folge aus und berechnet den Mittelwert der entsprechenden Ausgabewerte, um eine gewisse statistische Stabilisierung der Ergebnisse zu erreichen.

Durch diese umfangreichen Simulationskonfigurationen ergeben sich selbst auf schnellen Maschinen Berechnungszeiten, die sich über mehrere Wochen erstrecken können. Es liegt daher die Überlegung nahe, den Rechenaufwand nicht einer einzelnen Maschine aufzulasten, sondern diesen auf mehrere zur Verfügung stehende Rechner zu verteilen. SimpleGrid als Simulationsumgebung macht sich diese Idee zur Aufgabe.

Hintergrund der Überlegung ist der Wunsch, brachliegende Rechenkapazität sinnvoll

zur Verkürzung der Gesamtdauer einer Simulation einzusetzen. Heutige Computersysteme, wie sie z. B. als Bürorechner eingesetzt werden, sind üblicherweise die meiste Zeit weitgehend untätig. Sie werden in diesem Zustand entweder gerade nicht benutzt oder an ihnen werden Arbeiten verrichtet, die nur einen sehr geringen Teil der vorhandenen Rechenkapazität beanspruchen. So sind moderne Prozessoren bspw. bei der Arbeit mit Textverarbeitungsprogrammen größtenteils unbeschäftigt.

Das Thema dieser Arbeit ist die Frage, wie mit Hilfe der automatisierten Simulationsumgebung SimpleGrid die Rechenlast eines kompletten Simulationsprojekts unter frei verfügbaren Computern verteilt werden kann. Das Hauptaugenmerk liegt dabei besonders auf dem Scheduling von Simulationsjobs. Maschinen-Scheduling bezeichnet die Aufgabe, Aufträge (die auch als *Tasks* oder *Jobs* bezeichnet werden) dergestalt auf freie Maschinen zu verteilen, dass unter Berücksichtigung einer speziellen Zielfunktion ein gegebenes Zielkriterium optimiert wird [8, 6]. Im vorliegenden Fall wird die Zielfunktion durch die Gesamtdauer einer vollständigen Netzwerksimulation bestimmt. Es soll nun versucht werden, diese Zeit durch die Verwendung von Scheduling-Algorithmen zu minimieren.

Der Name SimpleGrid wurde als Anlehnung an den Bereich des Grid Computings gewählt, bei dem – ähnlich wie in dieser Arbeit – geographisch und logisch verteilte Ressourcen verbunden werden, um sie gemeinsam an einer Aufgabe arbeiten zu lassen oder sie als Einheit zusammengefasst Endanwendern für Dienstleistungen zur Verfügung zu stellen [5].

Es existiert eine Reihe von Anwendungen, die eine ähnliche Zielsetzung wie SimpleGrid haben. So werden im Rahmen des SETI@home-Projekts der Universität Berkeley gewöhnliche Arbeitsrechner von Privatleuten eingesetzt, um mit einem Radioteleskop eingefangene Signale auf Hinweise außerirdischer Intelligenz abzusuchen [50]. Um an dem Projekt teilzunehmen, lädt man sich einen Bildschirmschoner von der SETI@home-Internetseite herunter und installiert diesen auf seinem Rechner. Wird der Bildschirmschoner aktiv, so nimmt der Computer an den Berechnungen des Projekts teil.

Andere verteilte Rechenanwendungen, die ähnlich arbeiten, sind das Folding@home-Projekt [47] oder das Cancer Research Project [45]. Bei ersterem helfen Privatrechner bei der Erforschung von Krankheiten und ihren Heilungsmöglichkeiten durch die Berechnung der Faltung von Proteinen. Bei letzterem Projekt geht es um die verteilte Berechnung von Krebsheilungsmöglichkeiten.

SimpleGrid hat mit diesen Anwendungen gemein, dass hier mit Hilfe von verteiltem Rechnen eine große Anzahl von Computern an einer gemeinsamen Aufgabe arbeiten [11].

1.2 Überblick über den weiteren Aufbau dieser Arbeit

Die weiteren Kapitel sind wie folgt strukturiert. Kapitel 2 stellt zwei wichtige Themenkomplexe vor, auf denen diese Arbeit aufbaut. Hier werden zum einen diskrete Ereignissimulationen besprochen. Diese Simulationen bilden die Grundlage der Rechenaufgaben, deren Parallelisierung in einem Rechnerverbund die vorliegende Arbeit zur Aufgabe hat. Zum anderen wird eine Einführung in das Gebiet des Maschinen-Schedulings gegeben.

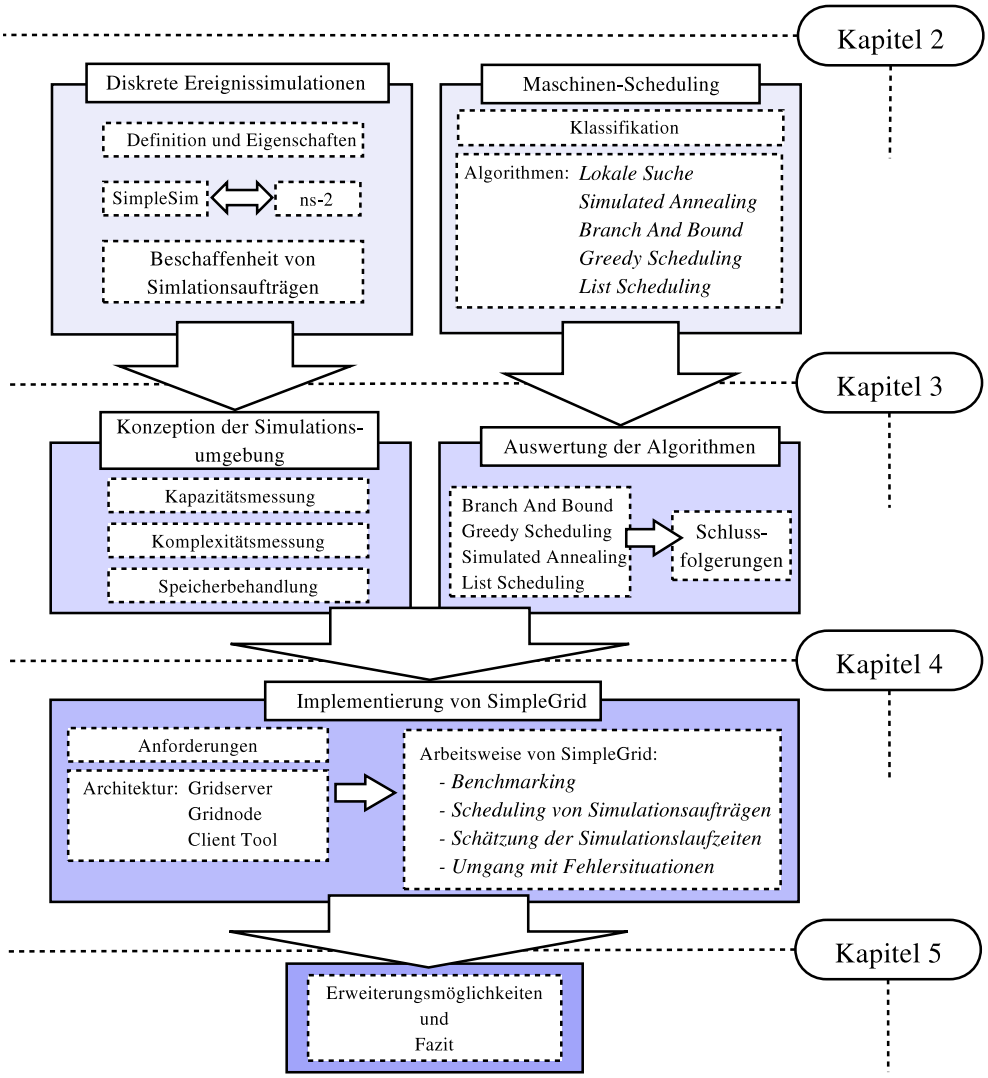


Abbildung 1.1: Organisation dieser Arbeit

Hier werden insbesondere die Algorithmen vorgestellt, mit deren Hilfe die Problemstellungen gelöst werden sollen, die Thema dieser Arbeit sind.

Der erste Teil von Kapitel 3 besteht aus einer umfassenden Diskussion der im zweiten Kapitel vorgestellten Konzepte. Es werden die Eigenschaften der zu Beginn der Arbeit eingeführten Scheduling-Algorithmen analysiert und ihre jeweiligen Einsatzmöglichkeiten beschrieben. Im letzten Teil des dritten Kapitels befindet sich eine Besprechung der Konzepte, die für die Lösung der bearbeiteten Problemstellung benötigt werden. Es werden die genauen Strukturen des Problems untersucht und Strategien entwickelt, wie man auf die dabei analysierten Besonderheiten eingehen kann.

In Kapitel 4 wird beschrieben, wie die zuvor angestellten Überlegungen in die konkrete Implementierung von SimpleGrid geflossen sind. Es wird die grundlegende Architektur der Simulationsumgebung beschrieben und auf einige wichtige Eigenschaften und Details der Implementierung eingegangen.

Schließlich werden im letzten Kapitel einige Ideen und Vorschläge diskutiert, die für die Erweiterung der Simulationsumgebung denkbar sind. Den Schluss bildet ein Fazit, welches die vorher besprochenen Punkte zusammenfasst.

Einen Überblick über die weitere Organisation dieser Arbeit bietet Abbildung 1.1.

Kapitel 2

Grundlagen

In diesem Kapitel sollen einige grundlegende Erläuterungen zu den Konzepten gemacht werden, die im weiteren Verlauf dieser Arbeit benötigt werden. In den folgenden Abschnitten werden zunächst diskrete Ereignissimulationen und anschließend Algorithmen für die Lösung komplexer Optimierungsprobleme besprochen.

2.1 Diskrete Ereignissimulationen

Simulationen gehören in der Forschung zu einem der wichtigsten Hilfsmittel, um Aussagen über komplexe Systeme und Prognosen über deren Verhalten machen zu können [24, 13]. Als System wird hierbei ein Prozess oder ein Konzept bezeichnet, das als Untersuchungsgegenstand für eine Simulation dient. Ein solches System soll in der Regel auf bestimmte Eigenschaften hin untersucht werden, um dadurch Aussagen über die Zukunft treffen zu können. Beispielhaft für ein System, das einen Prozess repräsentiert, sei die Flugzeugabfertigung eines Flughafens genannt. Hier können Simulationen bei der Suche nach Optimierungsmöglichkeiten des Abfertigungsprozesses helfen. Die Gesamtheit eines Computernetzwerkes ist ein weiteres Beispiel. Ein solches System ist für die Bestimmung der Eigenschaften und Einsatzmöglichkeiten von unterschiedlichen Netzwerkprotokollen von Interesse.

Für die Untersuchung eines interessierenden Konzeptes bieten sich mehrere Möglichkeiten an. Ein Untersuchungsobjekt in seiner natürlichen Umwelt zu studieren ist das nahe liegendste. Man betrachtet dafür, wie sich das Objekt in der Realität verhält und variiert von außen zielgerichtet die Parameter, die einen Einfluss auf dieses Verhalten ausüben.

Eine solche Herangehensweise ist in der Regel mit einem sehr hohen Kosten- und Zeitaufwand verbunden. Daher greift man üblicherweise auf die Untersuchung eines Modells der Wirklichkeit zurück. Ein Modell ist eine Menge von Annahmen, mit denen die Beschaffenheit eines Systems beschrieben werden kann. In ihm vereinen sich alle Variablen und Teilkonzepte, die einen Einfluss auf das Funktionieren eines Systems und dessen Interaktion mit seiner Umwelt haben [24]. Eine solche Abbildung der Realität kann durch ein physikalisches Modell dargestellt werden. So könnte z. B. die Miniatur eines Automobils aus Ton zur Messung von Strömungseigenschaften verwendet werden.

Daneben kann ein Konzept auch als ein mathematisches Modell repräsentiert werden. Hier werden alle Bestandteile in Form von Variablen und Beziehungsgleichungen ausgedrückt. Bei einer solchen mathematischen Darstellung bieten sich zwei Methoden der

Untersuchung an. Ist der Untersuchungsgegenstand – und damit das Modell – von eher einfacher Struktur, kann es durch analytische Methoden untersucht werden. Beispielhaft seien hierzu mathematische Verfahren, wie die Lineare Programmierung [12, 28], genannt. Übersteigt die Komplexität des Modells aber eine bestimmte Grenze, ist es nicht mehr so einfach möglich, ausschließlich analytische Methoden zur Anwendung kommen zu lassen. Die rein mathematische Behandlung aller Variablen wird in einem solchen Fall zu komplex und macht die Verwendung von Simulationen notwendig.

Während der Durchführung einer Simulation werden sämtliche Daten gesammelt, die über das Verhalten eines Modells in seiner simulierten Umgebung anfallen. Das Modell wird dabei durch eine Menge von Simulationsobjekten konstruiert. Ein solches Objekt stellt die logische Repräsentation eines realen Gegenstandes oder Konzeptes dar, das Teil des untersuchten Systems ist. Simulationsobjekte haben die Fähigkeit, Informationen über ihren internen Zustand und andere allgemeine Messdaten über eine vordefinierte Schnittstelle auszugeben. Diese Daten werden gesammelt, damit sie später numerisch ausgewertet werden können. Mit ihrer Hilfe ist es dann möglich, Schätzungen über die tatsächlichen Eigenschaften des Systems und Aussagen über dessen erwartetes Verhalten in der Realität zu erhalten.

Für die Klassifizierung von Simulationen lassen sich *stetige* von *diskreten* Systemen unterscheiden. In einem stetigen System verändern sich die relevanten Variablen kontinuierlich mit der Zeit. Bspw. lässt sich die Flugbahn eines Flugzeugs sehr gut durch ein stetiges System abbilden, da diese über eine Funktion der Zeit definiert ist.

Die Variablen der Simulationsobjekte eines diskreten Systems nehmen im Gegensatz dazu immer nur zu festen Zeitpunkten und ohne Zeitverlust neue Werte an. Simulationen solcher Systeme werden als *diskrete Ereignissimulationen* bezeichnet. Der Zustand des Modells verändert sich hier nur zu bestimmten Ereignissen, welche auf beliebigen Punkten entlang einer Zeitskala angeordnet sein können. Auf diese Weise wird eine Vereinfachung der simulierten Umwelt vorgenommen. Es müssen hierdurch ausschließlich diejenigen Prozesse betrachtet werden, die auch einen Einfluss auf das Endergebnis der Simulation haben. Alle anderen für das Endergebnis nicht direkt relevanten Vorgänge können zu Einzelereignissen aggregiert werden. Z. B. würde in einer Netzwerksimulation der Transport von einzelnen Datenbits über ein Übertragungsmedium nicht auf eine kontinuierliche Weise simuliert werden. Statt dessen würde man das Senden eines Datenwortes als Sendeereignis und das Empfangen des Wortes als Empfangsereignis definieren.

In diskreten Ereignissimulationen unterscheidet man *primäre* von *sekundären* Ereignissen. Erstere werden von außen durch den Benutzer vorgegeben. Sie definieren die vorgegebenen Eckpunkte, an denen sich eine Simulation orientieren soll. So werden als primäre Ereignisse in einer Netzwerksimulation bspw. vorgegeben, welche Netzwerknoten miteinander zu welchem Zeitpunkt in Kontakt treten sollen oder wann sich ein Knoten an welche Stelle bewegen soll.

Sekundäre Ereignisse ergeben sich während der Simulationsdurchführung als Folge der primären Ereignisse. Sie werden durch die Ablauflogik des Simulatorprogramms bestimmt. Bspw. kann das Anstoßen von Routenfindungsalgorithmen, die zu Beginn

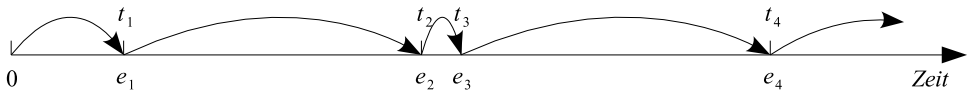


Abbildung 2.1: Voranschreiten der Simulationszeit zum jeweils nächsten Ereignis

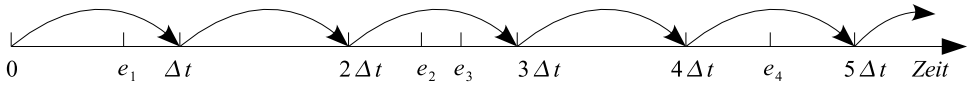


Abbildung 2.2: Voranschreiten der Simulationszeit um einen festen Betrag Δt mit den Ereignissen $e_1, \ldots, e_4$

eines Datentransfers zwischen zwei Netzwerkknoten benötigt werden, zur Erzeugung sekundärer Ereignisse führen.

Die vorrangige Aufgabe eines Simulators für diskrete Ereignissimulationen ist es nun, eine Liste von Ereignissen, die auch als *Events* bezeichnet werden, der Reihe nach abzuarbeiten. Das gesamte Simulationsmodell wird dabei von einem vorgegebenen Anfangszustand in einen bestimmten Endzustand transformiert. Dieser ergibt sich aus der Interaktion der Simulationsobjekte miteinander und mit der simulierten Umwelt. Die Ereignisse orientieren sich hierbei an einer internen Zeitachse. Durch diese wird die Simulationszeit festgelegt, welche nicht synchron zur realen Zeit läuft. Sie dient ausschließlich dazu, die temporale Beziehung zwischen einzelnen Events festzulegen und die Simulationsergebnisse zeitbezogen interpretierbar zu machen.

Für die Verwaltung der Eventliste ist ein so genannter *Scheduler* verantwortlich. Er verwaltet die interne Simulationszeit und bringt die anfallenden Ereignisse in der richtigen Reihenfolge zur Ausführung.

Für die Verwaltung der Simulationszeit bieten sich zwei Varianten an: das Vorwärtsspringen zum jeweils nächsten Ereignis und das Voranschreiten der Zeit um einen festen Betrag[1]. Die erste Variante ist die am häufigsten verwendete. Bei ihr wird die innere Uhr einer Simulation auf den Startzeitpunkt des jeweils als nächstes anstehenden Ereignisses gesetzt. Die Zeit zwischen zwei aufeinanderfolgenden Events wird also nicht weiter betrachtet. Abbildung 2.1 illustriert diesen Fall anhand eines Beispiels. Dort sind die Ereignisse e_1 bis e_4 auf einer Zeitachse angeordnet. Die Simulationszeit springt dabei in derselben Reihenfolge auf die Zeitpunkte t_1 bis t_4.

Im Gegensatz dazu wird beim Voranschreiten der Zeit um einen festen Betrag die Simulationszeit immer um ein festes Δt inkrementiert. Ereignisse, die innerhalb eines Intervalls

$$]k\Delta t, (k+1)\Delta t]$$

liegen, werden am jeweiligen Intervallende behandelt. Damit ist diese Variante ein Spezialfall der ersten Möglichkeit. Abbildung 2.2 stellt dies exemplarisch dar.

[1] engl.: *next-event time advance* und *fixed-increment time advance*

2.1.1 Bedeutung des Zufallsmoments für Simulationen

Das mit der Durchführung von Simulationen verfolgte Ziel ist die Gewinnung von Aussagen über die Eigenschaften realer Vorgänge und Systeme. In der Regel unterliegen diese einer Reihe von stochastischen Prozessen und sind damit von Zufallsvariablen abhängig. Ein stochastischer Prozess ist als eine Menge von Zufallsvariablen definiert, die auf einer Zeitachse angeordnet sind. D. h. jedem Zeitindex wird eine Zufallsvariable $X_i(a)$ zugeordnet [31]. Eine Zufallsvariable ist eine Abbildung, die jedem möglichen Ergebnis eines Zufallsexperiments einen bestimmten Wert zuweist [2].

Als Beispiel für einen stochastischen Prozess sei der Schalterbetrieb einer Bank genannt [24]. Hier stellen sich Kunden mit den Zeitabständen A_1, A_2, ... an der Schlange vor einem Schalter an und werden dort nach dem *First-Come, First-Served*-Prinzip bedient. Mit den Zufallsvariablen S_1, S_2, ... wird die Dauer der einzelnen Bedienvorgänge angegeben. Mit diesen Informationen lassen sich die Zufallsvariablen D_1, D_2, ... als stochastische Ergebnisprozesse definieren. Sie stehen für die Verzögerungen in der Warteschlange:

$$\begin{aligned} D_1 &= 0 \\ D_{i+1} &= \max\{D_i + S_i - A_{i+1}, 0\} \quad \text{für } i = 1, 2, \ldots \end{aligned}$$

Mit Hilfe von Simulationen möchte man nun ermitteln, wie die wahren Charakteristika des untersuchten Modells beschaffen sind. Da die Eingabewerte für eine Simulation wie oben beschrieben zumeist Zufallsvariablen sind, sind auch die Ausgabewerte selbst wieder Zufallsvariablen. Aus diesem Grund liegt das Interesse bei der Untersuchung eines Simulationsmodells auf der Bestimmung der Verteilung und Verteilungseigenschaften von verschiedenen stochastischen Ergebnisprozessen.

Mit der Durchführung von einzelnen Simulationen erhält man allerdings lediglich eine Stichprobe X_i aus den entsprechenden Verteilungen. Man muss daher auf Schätzungen zurückgreifen, um die gesuchten Aussagen zu erhalten. So ist bspw. das Stichprobenmittel

$$\bar{X} = \frac{1}{n} \sum_{i=0}^{n} X_i$$

eine Schätzfunktion für den Erwartungswert $EX = \mu$ einer Zufallsvariablen [2]. Um nun valide Aussagen treffen zu können, müssen Simulationen mit einer bestimmten Anzahl von Durchläufen wiederholt werden.

Dies lässt sich mit Anwendung des Gesetzes der großen Zahlen begründen. Sei $c > 0$ beliebig und die Wahrscheinlichkeit, dass das Stichprobenmittel $\bar{X}$ in das Intervall

$$[\mu - c; \mu + c]$$

fällt, gegeben durch

$$\mathfrak{P}(\mu - c \leq \bar{X} \leq \mu + c).$$

Sei weiter die Stichprobengröße gegeben durch $n \in \mathbb{N}$. Dann besagt das Gesetz der großen Zahlen, dass

$$\lim_{n\to\infty} \mathfrak{P}(\mu - c \leq \bar{X} \leq \mu + c) = 1.$$

Dies bedeutet also, dass mit genügend großem n die Schätzfunktion beliebig nahe an den gesuchten tatsächlichen Wert kommt. Für Simulationen, die stochastische Prozesse modellieren, hat dies zur Folge, dass sie mehrmals mit den gleichen Eingabeparametern durchgeführt werden müssen, um aussagekräftigere Ergebnisse zu erhalten.

Hat man für ein Simulationsmodell alle nötigen Eingabezufallsvariablen und deren Wahrscheinlichkeitsverteilungen festgelegt, muss der Simulator während des Voranschreitens der Simulationszeit zufällige Werte aus diesen Verteilungen erzeugen. Er bedient sich dafür eines Zufallsgenerators. Ein solcher Generator erzeugt anhand vorgegebener Rechenvorschriften so genannte *Pseudozufallszahlen* [23, 13]. Pseudozufallszahlen sind Folgen von Zahlen, die durch einen deterministischen Algorithmus generiert werden und damit nicht im eigentlichen Sinne zufällig sind. Sie zeigen aber bestimmte statistische Eigenschaften, die sie als Ersatz echt zufälliger Werte qualifizieren. So müssen diese Zahlen gleichverteilt auf dem Intervall $[0, 1]$ sein und dürfen keine Korrelationen untereinander aufweisen, um als gute Zufallszahlen in Frage zu kommen [24].

Eine Eigenschaft, die Pseudozufallszahlen von echten Zufallszahlen unterscheidet, ist deren Reproduzierbarkeit. Ein deterministischer Zufallszahlengenerator muss mit einem bestimmten Startwert initialisiert werden, bevor er eine Folge von Werten generieren kann. Dieser Startwert wird als *Random Seed* bezeichnet. Der Determinismus des Generators hat nun zur Folge, dass für den gleichen Startwert auch die gleiche Folge von Zahlen erzeugt wird [23].

Eine solche Eigenschaft ist für Simulationen besonders wünschenswert. So bekommt man mit Hilfe unterschiedlicher Random Seeds unterschiedliche Folgen von Zufallszahlen. Diese können für die Erhebung einer Stichprobe aus den Verteilungen der Zufallsvariablen eines Simulationsmodells eingesetzt werden [13]. Gleichzeitig bleiben die einzelnen Simulationsergebnisse reproduzierbar. D. h. man kann zu einem späteren Zeitpunkt genau die gleichen Simulationsergebnisse erzeugen, indem man dieselben Random Seeds früherer Simulationsläufe verwendet. Damit wird u. a. die Fehlersuche erleichtert und die exakte Wiederholung alter Simulationsläufe ermöglicht.

Diese Eigenschaft macht man sich auch bei Netzwerksimulationen zunutze. Um eine möglichst gute Stabilisierung der statistischen Ausgabewerte einer Simulation zu erhalten, wiederholt man diese einige Male mit gleichbleibenden Eingabewerten und variiert dabei den Startwert des Zufallszahlengenerators. Man erhält damit eine größere Stichprobe. Aufgrund der langen Rechendauer eines kompletten Simulationsprojekts kann man allerdings nur eine sehr kleine Anzahl von Wiederholungen wählen.

2.1.2 ns-2 und SimpleSim — Ein Vergleich

Zwei Vertreter von Simulatoren für diskrete Ereignissimulationen sollen an dieser Stelle vorgestellt und miteinander verglichen werden.

Der in der Forschung auf dem Gebiet der Rechnernetze am häufigsten eingesetzte

Netzwerksimulator ist der ns-2 [29]. Begonnen wurde die Entwicklung von ns-2 im Jahr 1989 als eine Variante des REAL Netzwerksimulators [36]. Später wurde die Arbeit an ns-2 durch das VINT-Projekt der DARPA unterstützt. Heute wird der Simulator von einer Reihe von Institutionen und Forschungseinrichtungen weiterentwickelt. So finden sich Firmen wie Sun Microsystems und Xerox unter den Trägern, sowie auch universitäre Einrichtungen, wie die University of Berkeley in Kalifornien (UCB) oder das Information Sciences Institute der University of Southern California (ISI/USC).

Ns-2 ermöglicht die Simulation einer Vielzahl unterschiedlicher Konfigurationen von kabelgebundenen und drahtlosen Netzwerken unter Verwendung verschiedenster Protokolle. Dem Benutzer steht hierbei eine offene Systemarchitektur zur Verfügung. Es ist ohne weiteres möglich, Implementierungen eigener Protokolle in die vorhandene Codebasis einzubringen, oder die bestehende Architektur nach eigenen Wünschen anzupassen. Auf diese Weise eignet sich ns-2 sehr gut für die Entwicklung und Untersuchung von Netzwerktechniken.

Implementiert wurde ns-2 mit einer Kombination der objektorientierten Sprachen C++ [41] und OTcl [30]. Der Kern des Simulators besteht aus C++-Klassen, während parallel dazu eine gespiegelte Implementierung in OTcl existiert [44]. Dadurch wird eine Einbindung von Steuerskripten ermöglicht. Die eigentliche Funktionalität des Programms wird durch den C++-Teil festgelegt. In diesem befinden sich die Klassen für den Ereignis-Scheduler, die Netzwerkknoten, die Kommunikationsprotokolle und weiterer Bestandteile. Klassen zur Bestimmung der Eigenschaften und Parameter für eine Simulation haben jeweils Gegenstücke in Form von OTcl-Klassen. C++- und OTcl-Teil sind über einen Spiegelungsmechanismus miteinander gekoppelt, d. h., für bestimmte C++-Klassen gibt es ein Pendant im OTcl-Teil. Mit Hilfe von OTcl-Skripten kann das Verhalten der Simulatorinterna von außen beeinflusst werden. Auf diese Weise kann der Benutzer beliebige Simulationsszenarien erstellen. Er übergibt dafür seinen Wünschen angepasste Skripte an den Simulator und lässt die damit definierten Szenarien von ihm berechnen. Durch die Skripte können bspw. Bewegungs- und Kommunikationsmuster von Netzwerkknoten oder die Parameter der Kommunikationsprotokolle festgelegt werden.

Der Simulator erhält damit eine sehr hohe Flexibilität. Seine Arbeitsweise kann ohne die Notwendigkeit einer Neukompilierung des Programmcodes dynamisch angepasst werden. Zudem lassen sich gewonnene Simulationsergebnisse zu späteren Zeitpunkten einfach nachvollziehen, indem man die zugehörigen Steuerskripte zusammen mit den entsprechenden Ergebnissen aufbewahrt.

Der Netzwerksimulator ns-2 zeigt allerdings auch eine Reihe von Nachteilen, die insbesondere im universitären Lehrbetrieb zu Tage treten. Durch die offene Architektur des Simulators bietet es sich an, ns-2 auch als Grundlage für wissenschaftliche Arbeiten von Studenten einzusetzen. Dies geschieht bspw. am Lehrstuhl für Praktische Informatik IV an der Universität Mannheim. Dort setzen Studenten schon lange für ihre Studien- oder Abschlussarbeiten ns-2 ein. Die Erfahrung hat dabei deutlich gemacht, dass der Neueinstieg in die Arbeit mit dem Simulator einen sehr hohen Einarbeitungsaufwand erfordert. Bevor man das Programm zielgerichtet einsetzen kann, muss man sich zunächst mit der

zweigeteilten Architektur von C++- und OTcl-Code vertraut machen und die Arbeitsweise des Simulators erlernen. Durch die hohe Komplexität des Programmcodes und eine stellenweise knapp ausfallende Dokumentation verbrauchen Studenten mit der Einarbeitung in das Programm regelmäßig einen relativ großen Anteil ihrer zur Verfügung stehenden Zeit.

Weiterhin problematisch an der Implementierung von ns-2 ist die Tatsache, dass dessen Codebasis von vielen unterschiedlichen und voneinander unabhängigen Entwicklern geschrieben worden ist. Es sind damit einige sich stark unterscheidende Programmierstile in den Code eingeflossen. Man stößt daher bei der Arbeit mit dem Simulator relativ häufig auf schwer zu findende Programmfehler. Es ist für einen Einsteiger sehr schwierig, in fremden Programmteilen solche Fehler ausfindig zu machen. Auch die Zweiteilung von OTcl und C++-Code erhöht die Fehleranfälligkeit des Simulators. Dazu kommt die hohe Komplexität der Sprache C++, mit der gerade Einsteiger Schwierigkeiten haben.

SimpleSim als Neuentwicklung eines Netzwerksimulators versucht die oben genannten Probleme zu vermeiden. Die Idee zu SimpleSim entsprang dem Wunsch, einen einfach zu bedienenden und unkompliziert wartbaren Netzwerksimulator zur Verfügung zu haben, der sich auch als Hilfsmittel bei der Erstellung wissenschaftlicher Arbeiten durch Studenten eignet.

SimpleSim versucht, dieses Ziel durch die folgenden Maßnahmen zu erreichen. In erster Linie soll die erhöhte Programmkomplexität, die durch die Verwendung zweier Programmiersprachen entsteht, vermieden werden. Daher wurde die Entscheidung getroffen, SimpleSim komplett mit der Sprache Java [18] zu entwickeln. Java eignet sich hierfür besonders aufgrund der Tatsache, dass die Sprache zum einen relativ einsteigerfreundlich ist und zum anderen nicht denselben Grad an Fehleranfälligkeit besitzt, wie C++ [34]. Man denke nur bspw. an die Zeiger- und Speicherleckproblematik, die es bei Java in derselben Form durch dessen *Garbage Collection*-Mechanismus nicht oder nur eingeschränkt gibt. Schließlich ist zu erwarten, dass Studenten tendenziell eher bessere Kenntnisse in Java als in C++ mitbringen.

Das Designziel der Einfachheit für SimpleSim drückt sich schon in dessen Namen aus. Um das Programm möglichst unkompliziert zu halten, soll auf unnötigen Ballast verzichtet werden. Die Verwendung einfacher Schnittstellen soll es erlauben, sich schnell in die Funktionsweise des Simulators einzuarbeiten und somit rasch Ergebnisse erhalten zu können.

Die Flexibilität bei der Konfiguration und Steuerung von Simulationsläufen wird ähnlich wie bei ns-2 durch die Verwendung von Skripten erreicht. Allerdings führt SimpleSim dazu keine weitere Skriptsprache in sein Konzept ein. Statt dessen bietet der Simulator u. a. eine Schnittstelle für die Verwendung von XML-Steuerdateien. XML [51] eignet sich als standardisiertes Format besonders gut für die Konfiguration von Simulationen.

Ein weiterer Ansatz, der in SimpleSim zur Fehlereindämmung verfolgt wird, besteht in der breiten Anwendung von so genannten Unit-Tests. Dies sind spezielle Klassen, mit denen sich Programmcode automatisiert auf semantische Korrektheit überprüfen lassen [3]. Dabei wird jede Methode einer zu testenden Klasse mit einer oder mehreren

Testfunktionen abgedeckt. Eine solche Funktion überprüft, ob die getestete Methode für bestimmte Eingabewerte die erwarteten Ausgabewerte liefert. Auf diese Weise kann man ein Sicherheitsnetz von Testfällen spannen, die bei später auftretenden Programmierfehlern sofort einen Hinweis auf diese Fehler geben können. Es wird dadurch möglich, auch solche Fehler relativ frühzeitig aufzuspüren, die sich sonst möglicherweise nur in seltenen Fällen oder an ganz anderen Stellen bemerkbar machen.

Das Konzept der Unit-Tests stammt ursprünglich aus dem Bereich des *Extreme Programming* [4]. Als Framework für Unit-Tests wird für SimpleSim JUnit [21] verwendet.

Die frühzeitige Aufspürung und Vermeidung von Programmierfehlern ist gerade für Simulationsprogramme von entscheidender Bedeutung. Anders als bei gewöhnlichen Anwendungen, bei denen vereinzelte unentdeckte Programmfehler nicht zwingend die Benutzung des Programms unmöglich machen, kann bei Simulatoren schon der kleinste Fehler signifikante Auswirkungen auf die Simulationsergebnisse haben. Die Verwendung von Unit-Tests ist hierbei ein wichtiges Hilfsmittel für die Sicherstellung der Korrektheit aller gewonnener Ergebnisse.

Mit den oben genannten Designzielen und Anforderungen an SimpleSim zielt dieser Simulator klar auf die gleichzeitige Verwendung in Forschung und Lehre ab.

2.1.3 SimpleSim als Simulator für diskrete Ereignissimulationen

Simulationsprogramme für diskrete Ereignissimulationen haben üblicherweise eine Reihe von Kernmodulen gemeinsam, aus denen sie zusammengesetzt sind [24]. Auch für SimpleSim als ein Vertreter dieser Klasse von Simulatoren sind diese Elemente implementiert. Die folgende Aufzählung bietet einen Überblick über die wichtigsten Komponenten eines Simulators für diskrete Ereignissimulationen und eine Beschreibung wie diese Komponenten in SimpleSim realisiert sind.

Zustand des Systems Der Zustand, in dem sich ein simuliertes System befindet und mit dem sich dieses zu jedem Zeitpunkt beschreiben lässt, wird über eine Menge von Variablen festgehalten und verwaltet. Diese Variablen befinden sich hauptsächlich in den Klassen der Simulationsobjekte. In SimpleSim sind dies die Klassen für Netzwerkknoten, Netzwerkprotokolle, verschickte Datenpakete und viele weitere mehr.

Ereignis-Scheduler Der Scheduler bestimmt die Reihenfolge der abzuarbeitenden Ereignisse. Zudem kümmert er sich um das Voranschreiten der Simulationszeit. Aktive Simulationsobjekte können dem Scheduler sekundäre Ereignisse übergeben. Alle Events werden zu den für sie festgelegten Zeitpunkten aktiviert, woraus sich entsprechende Änderungen des Systemzustands ergeben.

Simulationszeit Die Verwaltung der internen Simulationszeit geschieht mit Hilfe einer Variable innerhalb der Scheduler-Klasse.

Liste der abzuarbeitenden Ereignisse Alle noch nicht behandelten primären und sekundären Ereignisse werden vom Scheduler in einer Liste verwaltet.

Statistikvariablen Allgemeine Leistungsparameter und Statistiken des simulierten Systems werden während der Simulation ermittelt und in speziellen Variablen aggregiert. Am Ende der Simulation werden diese zur weiteren Verarbeitung in eine Datei ausgegeben.

Systeminitialisierung Zu Beginn jeder Simulation müssen alle Variablen des simulierten Modells mit ihren Startwerten initialisiert werden. Die interne Zeit wird auf Null gesetzt. Alle von außen sichtbaren Variablen der Simulationsobjekte werden mit Werten gefüllt, welche der Benutzer vorgeben kann. Dem Netzwerksimulator werden dafür über eine Konfigurationsdatei Steuerdaten übergeben, mit deren Hilfe der Startzustand des simulierten Modells festgelegt wird.

Ablaufverfolgung Um das Verhalten eines simulierten Systems nachvollziehen und auswerten zu können, braucht man genaue Informationen über alle während der Simulation in den Simulationsobjekten ablaufenden internen Prozesse. Dafür wird ein so genannter *Tracer* verwendet, mit dem das System Zustandsmeldungen in aggregierter Form ausgeben kann. Ändern sich die Zustandsvariablen eines Simulationsobjektes, so können über die Schnittstelle des Tracer-Moduls alle für diese Änderung relevanten Daten in eine Datei geschrieben werden.

Hauptmodul Das Hauptmodul sorgt für die Integration aller Bestandteile des Simulators. Es stößt die Arbeit des Schedulers zu Beginn der Simulation an und sorgt für die korrekte Behandlung der vom Benutzer übergebenen Konfigurationsparameter.

2.1.4 Beschaffenheit von Simulationsaufträgen

In diesem Abschnitt wird näher darauf eingegangen, wie die Simulation mit einem Netzwerksimulator grundsätzlich aufgebaut ist. Zudem soll beschrieben werden, wie eine solche charakterisiert werden kann. Ein Simulationslauf ist typischerweise durch ein *Szenario* für die Repräsentation des simulierten Systems und durch eine Menge von Eingabeparametern definiert. Mit Hilfe des Szenarios werden die Rahmenbedingungen für die simulierte Umgebung festgelegt. Dazu gehören in erster Linie die simulierte Netzwerkinfrastruktur und alle Gegebenheiten, die Auswirkungen auf die simulierte Umwelt haben. So muss z. B. für die Simulation eines mobilen Ad-Hoc Netzwerks[2] festgelegt werden, an welchen Positionen sich die einzelnen Netzwerkknoten befinden und wohin sich diese während der Simulation mit welcher Geschwindigkeit bewegen sollen. Zusätzlich können dabei auch für Funkwellen undurchlässige Hindernisse, wie etwa Gebäude in einer Stadt, definiert werden. Neben diesen infrastrukturellen Gegebenheiten werden für ein Szenario zusätzlich die Kommunikationsmuster festgelegt, nach denen die beteiligten Netzwerkknoten miteinander in Kontakt treten und Daten austauschen.

Der zweite Bestandteil zur Definition einer Simulation wird durch die Menge der Eingabeparameter gebildet. Mit ihnen lassen sich die internen Variablen des Simulators

[2] engl.: *mobile ad-hoc network* oder *MANET*. Darunter versteht man drahtlose Netzwerke, deren Teilnehmer sich frei im Raum bewegen können. Dadurch entsteht eine nicht-statische Netztopologie [32].

und der Simulationsobjekte von außen beeinflussen und mit gewünschten Werten belegen. Sie bestimmen im Wesentlichen das spätere Verhalten der simulierten Objekte während der Simulationsdurchführung. Üblicherweise bestehen die Eingabeparameter aus all denjenigen Variablen, deren Auswirkung auf das simulierte System untersucht werden soll. Als mögliche konfigurierbare Werte seien hier beispielhaft die verwendeten Routing-Protokolle der Netzwerkknoten, die maximale Sendereichweite, Senderaten und Datenpaketgrößen genannt.

Die Menge der Eingabeparameter ergibt zusammen mit einem Szenario das Grundgerüst für einen Simulationslauf. Üblicherweise werden diese Daten dem Simulator in Form von Skripten und Konfigurationsdateien übergeben. In ihnen werden alle relevanten Informationen in strukturierter Form und für das Simulationsprogramm auswertbar aufgeführt. Startet man nun den Simulator mit allen benötigten Eingabewerten, erhält man danach eine oder mehrere Ausgabedateien mit den Ergebnissen der Simulation. Eine solche Ausgabedatei kann ein ausführliches Protokoll über alle in der simulierten Umgebung stattgefundenen Ereignisse oder eine Zusammenfassung hiervon in Form von statistischen Auswertungen enthalten.

An dieser Stelle sei besonders der Unterschied zwischen einem Simulations*lauf* und einem Simulations*auftrag* hervorgehoben. Es ist für das Verständnis der folgenden Kapitel wichtig, diese beiden Konzepte zu trennen. Bei der Untersuchung eines Systems ist nicht nur die Frage von Interesse, wie sich das System mit einer bestimmten Konfiguration verhält. Gleichermaßen von Bedeutung ist auch die Frage, welche Unterschiede ein System in seinem Verhalten zeigt, wenn man bestimmte Parameter variiert. Um dieser Fragestellung nachzugehen, führt man den Simulator in der Regel mehrere Male hintereinander aus. Dabei hält man alle Parameter bis auf einen fest. Diese interessierende Variable setzt man nun bei jedem Aufruf des Simulators auf einen anderen Wert und zeichnet die sich dadurch ergebenden Änderungen im Verhalten des Systems auf. Die einmalige Ausführung des Simulators wird als ein Simulationslauf bezeichnet.

Auf diese Weise erhält man eine Menge von unterschiedlichen Konfigurationen, die jeweils einzeln in einem separaten Simulatoraufruf behandelt werden. Die Gesamtheit aller dieser Simulationsläufe wird im Folgenden als *Simulationsauftrag* bezeichnet. Z. B. kann man für eine Netzwerksimulation eine Reihe von unterschiedlichen Kommunikationsprotokollen mit einem über mehrere Stufen steigenden Datenaufkommen kombinieren. Damit lässt sich untersuchen, wie sich die einzelnen Protokolle unter verschiedenen Lastsituationen im Hinblick auf bestimmte Leistungskriterien, wie Durchsatz und Verzögerung, verhalten werden.

2.2 Maschinen-Scheduling

Zu Beginn dieser Arbeit wurde in Abschnitt 1.1 die Problemstellung umrissen, die mit Hilfe der automatisierten Simulationsumgebung SimpleGrid gelöst werden soll. Dort hieß es, die hier vorgestellte Software wurde mit dem Ziel entwickelt, umfangreiche Netzwerksimulationsprojekte dergestalt auf einen Verbund von verfügbaren Rechnern zu verteilen, dass das gesamte Projekt in möglichst kurzer Zeit abgearbeitet werden kann.

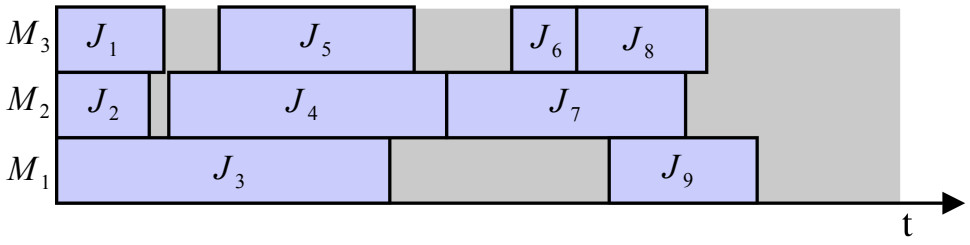


Abbildung 2.3: Beispiel für ein Gantt-Diagramm mit 9 Aufträgen $J_1, \ldots, J_9$ und 3 Maschinen $M_1, \ldots, M_3$

Mit der Lösung solcher Probleme befasst man sich auf dem Gebiet des Operations Research, einer Teildisziplin der Angewandten Mathematik. In der Unternehmensforschung spielen ähnliche Fragestellungen in Bereichen wie der Produktionsplanung, Logistik und Kapazitätsplanung eine wichtige Rolle. Welche Lösungsansätze für das hier behandelte Problem existieren und wie diese sinnvoll eingesetzt werden können, soll nun in den folgenden Abschnitten behandelt werden.

Die Verfahren für die optimale Verteilung eines Simulationsauftrags auf verfügbare Ressourcen lassen sich auf dem Gebiet des *Maschinen-Schedulings* (auch Maschinenbelegungsplanung) finden. Ein Scheduling-Problem ist wie folgt definiert [8]. Gegeben seien m Maschinen M_j $(j = 1, \ldots, m)$ und n Aufträge, oder auch *Jobs*, J_i $(i = 1, \ldots, n)$. Als *Schedule* oder *Maschinenbelegungsplan* wird eine Zuordnung bezeichnet, die jedem Job einen oder mehrere Zeitintervalle auf einer oder mehreren Maschinen zuweist. Während eines solchen Zeitintervalls wird der entsprechende Auftrag der ihm zugeteilten Maschine bearbeitet. Ein Scheduling-Algorithmus hat nun zur Aufgabe, eine Zuordnung zu finden, die ein vorgegebenes Optimalitätskriterium erfüllt.

Maschinenbelegungspläne können in Form von Gantt-Diagrammen dargestellt werden. Abbildung 2.3 zeigt beispielhaft ein solches Diagramm. Hier sind neun Aufträge auf drei Maschinen verteilt worden. Die Länge der einzelnen Auftragsbalken entspricht dabei der Zeit, die für die Bearbeitung des jeweiligen Auftrags benötigt wird.

Ein Job J_i kann sich aus einer Anzahl unterschiedlicher Operationen $O_{i1}, \ldots, O_{in_i}$ zusammensetzen. Jede Operation O_{ik} setzt eine bestimmte Zeit p_{ik} voraus, die für ihre Bearbeitung notwendig ist. Weiterhin können für jeden Auftrag J_i einige zusätzliche Bedingungen festgelegt werden. Dies sind im Einzelnen ein Freigabezeitpunkt r_i, eine Frist d_i und eine Gewichtung w_i. Der Freigabezeitpunkt r_i bestimmt den Termin, zu welchem die erste Operation O_{i1} eines Jobs zur Bearbeitung bereitsteht. Eine Frist d_i gibt den spätesten Zeitpunkt an, zu dem der Auftrag vollständig abgeschlossen sein muss. Die Gewichtung w_i kann als Steuerungsparameter in der Kostenfunktion $f_i(t)$ eingesetzt werden. Diese Funktion wird für jedes Scheduling-Problem definiert. Sie gibt die Kosten an, die entstehen, wenn Auftrag J_i zum Zeitpunkt t abgeschlossen wird.

Für die Bearbeitung der einzelnen Aufträge J_i steht eine bestimmte Anzahl von Ma-

schinen zur Verfügung. Für jede Operation O_{ik} wird dabei eine Menge

$$\mu_{ik} \subseteq \{M_1, \ldots, M_m\}$$

einzelner Maschinen festgelegt, auf denen diese ausgeführt werden können. Hat man mit den μ_{ik} einelementige Mengen, so bezeichnet man die M_i als *dedizierte Maschinen*. D. h., jede Operation kann dann nur auf einer bestimmten Maschine M_i bearbeitet werden. Enthält jedes μ_{ik} die Gesamtheit aller vorhandenen Maschinen, so spricht man von *parallelen Maschinen*.

2.2.1 Klassifikation von Scheduling-Problemen

Die Vielzahl der bekannten Scheduling-Probleme lässt sich mit einem System nach Graham et al. [15] klassifizieren. Man erhält damit die Möglichkeit, sich mit Hilfe einer kompakten Notation auf bestimmte Problemfälle zu beziehen. Dies soll im Folgenden kurz vorgestellt werden.

Die Klasse eines Scheduling-Problems kann durch die dreielementige Notation

$$\alpha \mid \beta \mid \gamma$$

beschrieben werden. Das α steht hierbei für die Maschinenumgebung, β für die Auftragscharakteristika und γ für ein Optimalitätskriterium. Diese einzelnen Variablen können mit bestimmten Werten belegt werden, wodurch eine genaue, kodifizierte Beschreibung eines Scheduling-Problems möglich wird.

2.2.1.1 Maschinenumgebung α

Die Maschinenumgebung wird durch eine Zeichenkette $\alpha = \alpha_1\alpha_2$ beschrieben. α_2 spezifiziert die Anzahl der Maschinen, die für die Auftragsbearbeitung zur Verfügung stehen. Gilt $\alpha_2 = k$, wobei k eine positive, natürliche Zahl größer als Null ist, dann steht dieses k für eine beliebige, aber feste Anzahl von Maschinen. Ist $\alpha_2 = \circ$ gesetzt, wobei $\circ$ das Leersymbol darstellt, ist die Maschinenzahl nicht näher bestimmt und kann beliebige Werte annehmen.

α_1 kann einen der Werte $\circ, P, Q, R, PMPM, QMPM, G, X, O, J, F$ annehmen. Gilt für $\alpha_1 \in \{\circ, P, Q, R, PMPM, QMPM\}$, dann bestehen die einzelnen Jobs J_i nur aus jeweils einer einzigen Operation. Ist $\alpha_1 = \circ$, so muss jedes J_i auf einer speziellen, dedizierten Maschine bearbeitet werden. Wenn hingegen jeder Auftrag J_i an eine beliebige Maschine $M_1, \ldots, M_m$ zugewiesen werden kann, hat man den Fall paralleler Maschinen und $\alpha_1 \in \{P, Q, R\}$. Dabei steht P für identische, parallele Maschinen, Q für einheitliche, parallele Maschinen und R für unterschiedliche, parallele Maschinen[3]. Im ersten Fall gilt, dass die Bearbeitungszeit p_{ij} für einen Auftrag J_i auf jeder Maschine M_j gleich groß ist, d. h., $p_{ij} = p_i$ für alle $j = 1, \ldots, m$ und $i = 1, \ldots, n$. Bei einheitlichen, parallelen Maschinen ist die Bearbeitungszeit abhängig von der Geschwindigkeit s_j einer Maschine M_j. Für diesen Fall $\alpha_1 = Q$ gilt daher $p_{ij} = p_i/s_j$. Ist $\alpha_1 = R$, so ist die Geschwindigkeit

[3] engl.: *identical, uniform* bzw. *unrelated parallel machines*

einer Maschine zusätzlich abhängig von dem Auftrag, der auf ihr ausgeführt werden soll, d. h. $p_{ij} = p_i/s_{ij}$. Die beiden Maschinenumgebungen $PMPM$ und $QMPM$ stehen für Mehrzweckmaschinen mit identischer bzw. einheitlicher Geschwindigkeit. Diese können mit unterschiedlichen Werkzeugen ausgerüstet werden, um verschiedenartige Aufträge bearbeiten zu können.

Nimmt α_1 einen Wert aus $\{G, X, O, J, F\}$ an, dann liegt ein Modell vor, bei dem jeder Job J_i aus einer Reihe von Einzeloperationen $O_{i1}, \ldots, O_{in_i}$ zusammengesetzt ist. Derartige Aufträge werden in einer so genannten Shop-Fertigung bearbeitet. Die einzelnen Belegungen für α_1 bedeuten dabei der Reihe nach *General Shop*, *Mixed Shop*, *Open Shop*, *Job Shop* und *Flow Shop*. Diese Maschinenumgebungen sollen an dieser Stelle nicht weiter behandelt werden.

2.2.1.2 Auftragscharakteristikum β

Das Auftragscharakteristikum β setzt sich aus den Variablen $\beta_1, \beta_2, \beta_3, \beta_4, \beta_5$ und β_6 zusammen. β_1 gibt dabei an, ob für die Jobs ein so genanntes *Preemption* möglich ist. Ist dies der Fall, so kann die Bearbeitung eines einzelnen Jobs auf einer Maschine unterbrochen werden und zu einem späteren Zeitpunkt oder auf einer anderen Maschine wieder fortgesetzt werden. Es wird dann $\beta_1 = pmtn$ gesetzt.

β_2 beschreibt die Reihenfolgebeziehung zwischen den einzelnen Jobs. Eine solche Beziehung kann durch einen azyklischen, gerichteten Graphen $G = (V, A)$ ausgedrückt werden [10]. Dabei entsprechen die Knoten $V = \{1, \ldots, n\}$ des Graphen den einzelnen Jobs. Jede Kante $(i, k) \in A$ drückt aus, dass der Auftrag J_i erledigt sein muss, bevor Auftrag J_k beginnt. Für den Fall gegebener Reihenfolgebeziehungen wird $\beta_2 = prec$ gesetzt. Graphen mit spezieller Struktur können durch eigene Bezeichnungen beschrieben werden. So existieren die Bezeichnungen *chains*, *intree*, *outtree* und *sp-graph* für β_2, auf die hier nicht vertiefend eingegangen werden soll [8].

Die Variable β_3 wird mit r_i belegt, falls für jeden Job ein Freigabetermin bestimmt werden kann, zu dem der Auftrag für die Bearbeitung bereit steht.

Die notwendige Bearbeitungsdauer für einen Job wird mit der Variable β_4 ausgedrückt. Ist β_4 mit $p_i = 1$ (bzw. $p_{ij} = 1$) gegeben, so heißt dies, dass jeder Auftrag eine einheitliche Bearbeitungsdauer hat. Des Weiteren können für β_4 genauere Angaben gemacht werden, wie z. B. $p_i \in \{1, 2\}$.

Sind für die einzelnen Jobs Fristen festgelegt, so kann dies mit dem Parameter β_5 spezifiziert werden. Darf ein Auftrag J_i nicht später als der Zeitpunkt d_i beendet werden, so drückt man dies durch $\beta_5 = d_i$ aus.

Schließlich bezeichnet die letzte Variable β_6 Scheduling-Varianten, bei denen einzelne disjunkte Teilmengen von Jobs zu Stapeln, oder *Batches*, zur gemeinsamen Bearbeitung auf einer Maschine gruppiert werden. Hier gibt β_6 die Art des Batches an [8]. Auch dieser Fall soll hier nicht weiter vertieft werden.

2.2.1.3 Zielfunktion γ

Die letzte der drei Klassifikationsvariablen, γ, stellt das Optimalitätskriterium dar, welches von der Lösung eines Scheduling-Problems erfüllt werden soll. Dieses Kriterium wird durch eine Zielfunktion ausgedrückt, die durch einen Maschinenbelegungsplan je nach Problemstellung maximiert oder minimiert werden soll.

Im Folgenden wird der Zeitpunkt, zu dem die Bearbeitung eines Auftrags J_i abgeschlossen wurde, mit C_i und die damit verbundenen Kosten mit $f_i(C_i)$ bezeichnet. Die Zielfunktion eines Scheduling-Problems kann durch eine Gesamtkostenfunktion ausgedrückt werden, die durch einen realisierbaren Maschinenbelegungsplan minimiert werden soll. Es lassen sich zwei Arten dieser Gesamtkostenfunktionen unterscheiden. Zum einen unterscheidet man ein so genanntes *Engpasskriterium*[4]

$$f_{max}(C) := \max\{f_i(C_i) \mid i = 1, \ldots, n\}$$

und zum anderen ein *Summenkriterium*[5]

$$\sum f_i(C) := \sum_{i=1}^{n} f_i(C_i).$$

Die am häufigsten vorzufindenden Zielfunktionen sind die *Produktionsspanne*[6]

$$\max\{C_i \mid i = 1, \ldots, n\}, \tag{2.1}$$

die *Gesamtdurchflusszeit*[7]

$$\sum_{i=1}^{n} C_i \tag{2.2}$$

und die *gewichtete Gesamtdurchflusszeit*

$$\sum_{i=1}^{n} w_i C_i. \tag{2.3}$$

Für die Kodifizierung dieser Fälle setzt man jeweils $\gamma = C_{max}$ für (2.1), $\gamma = \sum C_i$ für (2.2) und $\gamma = \sum w_i C_i$ für (2.3).

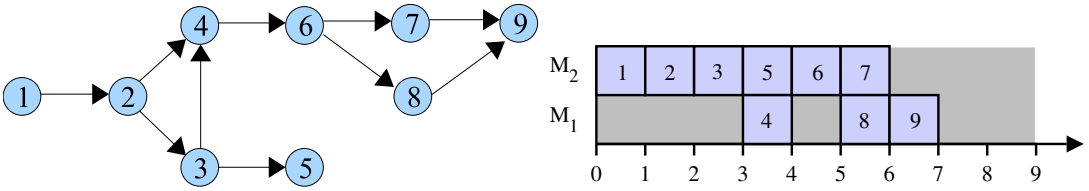
Neben diesen Optimalitätskriterien existieren noch eine ganze Reihe weiterer Zielfunktionen, die sich anhand gegebener Bearbeitungsfristen d_i der einzelnen Aufträge J_i angeben lassen. Diese werden im Einzelnen in Tabelle 2.1 aufgeführt.

Für all diese Funktionen G_i aus Tabelle 2.1 lassen sich nun vier verschiedene Zielkriterien festlegen: $\gamma = \max G_i$, $\max w_i G_i$, $\sum G_i$, $\sum w_i G_i$. Häufig verwendete Zielfunktionen sind bspw. die maximale Verspätung $L_{max} := \max_{i=1}^{n} L_i$, die Summe der Strafkosten $\sum U_i$, die Summen der absoluten und quadrierten Abweichungen $\sum D_i$ bzw. $\sum S_i$ und deren gewichteten Versionen $\sum w_i U_i$, $\sum w_i D_i$ bzw. $\sum w_i S_i$.

[4] engl.: *bottleneck objective*
[5] engl.: *sum objective*
[6] engl.: *makespan*
[7] engl.: *total flow time*

Zielfunktion	Bedeutung
$L_i := C_i - d_i$	Verspätung
$E_i := \max\{0, d_i - C_i\}$	Frühzeitigkeit
$T_i := \max\{0, C_i - d_i\}$	maximale Verspätung
$D_i :=\mid C_i - d_i \mid$	absolute Abweichung
$S_i := (C_i - d_i)^2$	quadrierte Abweichung
$U_i := \begin{cases} 0 & \text{falls } C_i \leq d_i \\ 1 & \text{sonst} \end{cases}$	einheitliche Strafe

Tabelle 2.1: Zielfunktionen für das Optimalitätskriterium eines Scheduling-Problems



(a) Reihenfolgeabhängigkeitsgraph

(b) Gantt-Diagramm für einen möglichen Maschinenbelegungsplan

Abbildung 2.4: Beispiel einer Probleminstanz für $P \mid prec; p_i = 1 \mid C_{max}$

2.2.1.4 Beispiel für die Klassifikation eines Scheduling-Problems

Ein Beispiel aus Brucker [8] soll an dieser Stelle die Verwendung der vorgestellten Klassifizierung verdeutlichen. Gegeben sei ein Scheduling-Problem, bei dem identische Maschinen verwendet werden. Als Optimalitätskriterium soll hier die Minimierung der Gesamtproduktionsdauer angestrebt werden. Weiterhin hat jeder zu verteilende Auftrag denselben Bearbeitungsaufwand. Die Jobs können zudem bestimmte Reihenfolgeabhängigkeiten aufweisen. Scheduling-Probleme dieser Art lassen sich nach Graham et al. [15] als

$$P \mid prec; p_i = 1 \mid C_{max}$$

klassifizieren.

Ein solches Problem ist gegeben durch einen gerichteten Graphen mit n Knoten und einer Menge von m Maschinen. Abbildung 2.4 zeigt ein Beispiel hierfür. Dort ist der Reihenfolgeabhängigkeitsgraph für 9 Jobs dargestellt. Ein möglicher Schedule mit $m = 2$ und $C_{max} = 7$ zeigt Abbildung 2.4(b).

2.2.2 Algorithmen für das Scheduling

Mit den bisher gemachten Erläuterungen kann untersucht werden, mit welchen Hilfsmitteln das in dieser Arbeit gestellte Problem gelöst werden kann. Wie schon weiter oben beschrieben, soll eine gegebene Aufgabe so unterteilt werden, dass sie von mehreren Rechnern parallel und möglichst schnell bearbeitet werden kann. Dabei stellt sich die

Frage, wie diese Teilaufgaben am besten auf die verfügbaren Rechner verteilt werden sollen. Weist man eine sehr große Aufgabe einem langsamen Rechner zu, so braucht dieser u. U. eine zu lange Zeit für deren Bearbeitung. Es muss also ein Weg gefunden werden, die Teilaufgaben möglichst ausgeglichen auf die verfügbaren Rechner zu verteilen, um nicht einzelne Rechner zu überlasten, während man gleichzeitig die Kapazität anderer Computer ungenutzt lässt.

Für die Beantwortung solcher Fragestellungen existieren eine Reihe von Algorithmen. Im Folgenden sollen vier Scheduling-Verfahren vorgestellt werden, die sich für die Lösung der hier vorgestellten Problemstellung anbieten.

2.2.2.1 Lokale Suche

Scheduling-Probleme, bei denen kein Preemption möglich ist, Jobs also nicht unterbrochen werden können, gehören zu den so genannten *diskreten Optimierungsproblemen* [8]. Ein diskretes Optimierungsproblem kann wie folgt definiert werden: Für eine endliche Menge S und eine Zielfunktion $f : S \to \mathbb{R}$ wird eine Lösung $s_{opt} \in S_{opt} \subseteq S$ gesucht, so dass die folgende Ungleichung erfüllt ist:

$$f(s_{opt}) \leq f(s) \quad \forall s \in S.$$

Hier steht S_{opt} für die Menge aller optimalen Lösungen. In ihr können ein oder mehr Elemente vorhanden sein.

Die lokale Suche ist eine Möglichkeit, Lösungen für ein diskretes Optimierungsproblem zu finden. Im vorliegenden Fall ist das Problem wie folgt gegeben. Die Menge S besteht aus all denjenigen gültigen Maschinenbelegungsplänen, die aus allen Jobs J_i und allen Maschinen M_j erzeugt werden können. Die Zielfunktion $f(s)$ ist eine Abbildung, die jedem Schedule $s \in S$ den Zeitpunkt zuordnet, zu dem der letzte Job beendet wurde. Der gesuchte Schedule s_{opt} ist nun der Maschinenbelegungsplan mit der kürzesten Ausführungsdauer.

Für die Vorstellung lokaler Suchmethoden ist zunächst die Definition einer *Nachbarschaftsstruktur*

$$\mathcal{N} : S \to 2^S$$

auf S nötig [8]. Hier steht 2^S für die Potenzmenge von S. Eine Nachbarschaftsstruktur ist eine Abbildung, die jeder Lösung $i \in S$ eine Teilmenge $S_i \subset S$ von Lösungen zuordnet, die „nahe an“ i liegen. Die Menge S_i heißt *Nachbarschaft* der Lösung i, und jedes $j \in S_i$ heißt *Nachbarlösung* oder *Nachbar* von i. Es gilt dabei $j \in S_i \Leftrightarrow i \in S_j$ [1].

Ausgehend von einer beliebigen Lösung i kann man ein $j \in S_i$ dadurch erhalten, indem man i mittels einer atomaren Operation leicht abändert. Dies kann bspw. durch die Permutation von Lösungselementen erfolgen.

Eine einfache lokale Suche sucht nun so lange nach besseren Lösungen in den Nachbarschaften bisher gefundener Lösungen, bis dies nicht mehr weiter möglich ist. Der Algorithmus lässt sich mit Pseudocode, wie in Algorithmus 1 dargestellt, beschreiben.

Mit diesem Verfahren lässt sich ein $\hat{s}$ finden, welches keine bessere Lösung in seiner Nachbarschaft $\mathcal{N}(\hat{s})$ hat. Allerdings erkennt man leicht, dass $\hat{s}$ nicht zwingend das globale Optimum s_{opt} sein muss. In der Regel erhält man mit einer lokalen Suche lediglich

Algorithmus 1 Lokale Suche

1: Wähle eine Startlösung $s \in S$;
2: **repeat**
3: Erzeuge die beste Lösung $s' \in \mathcal{N}(s)$;
4: **if** $f(s') < f(s)$ **then**
5: $s := s'$;
6: **end if**
7: **until** $f(s') \geq f(s)$;

ein lokales Optimum, für das gilt

$$f(\hat{s}) \geq f(s_{opt}).$$

Um nun mit einer lokalen Suche das globale Optimum finden zu können, muss der Algorithmus die Möglichkeit bekommen, aus einem lokalen Optimum wieder zu entkommen. Ein Algorithmus, der genau dies ermöglicht, ist das Simulated Annealing.

2.2.2.2 Simulated Annealing

Das *Simulated Annealing*, oder auch *simulierte Abkühlung*, stellt eine Verbesserung der lokalen Suche dar, indem dem Algorithmus eine probabilistische Komponente hinzufügt wird. Das Verfahren wird daher auch *Stochastic Relaxation* oder *Probabilistic Hill Climbing* genannt [1]. *Hill Climbing* oder *Bergsteigeralgorithmus* sind alternative Bezeichnungen für die lokale Suche. Diese Begriffe verbildlichen für die Maximierung einer Zielfunktion die Suche nach der besten Nachbarlösung und damit nach einer „Bergspitze“ in der sinnbildlichen Hügellandschaft, die durch die Funktion $f(s)$ beschrieben wird.

Annealing beschreibt ein Konzept aus der Werkstoffkunde. Der Begriff steht für den Vorgang, der beim Härten eines Materials abläuft. Bei diesem Prozess wird ein Stoff, z. B. ein Metall, bis zu seinem Schmelzpunkt erhitzt. In diesem flüssigen Zustand ordnen sich die Moleküle des Feststoffs nach einem zufälligen Muster an. Anschließend lässt man das Material sehr langsam abkühlen. Durch diese behutsame Abkühlung haben die Moleküle der Schmelze genügend Zeit, sich selbstständig in einem sehr stabilen, kristallinen Gitter anzuordnen und einen energiearmen Grundzustand zu erreichen. Bezogen auf die Gesamtenergie hat das Material damit ein globales Optimum erreicht [22].

Das Gegenstück zu diesem Härtungsvorgang ist ein sehr schnelles Abkühlen, oder Abschrecken des Materials. In diesem Fall ordnen sich die Moleküle in energiereichen und metastabilen Strukturen an, was einem Verharren in lokalen Optima entspricht. Das Material wird spröde.

In Analogie zu dem oben beschriebenen physikalischen Härtungsprozess wurde 1983 der Algorithmus Simulated Annealing von Kirkpatrick et al. [22] vorgeschlagen, um Optimierungsprobleme zu lösen. Ein solches Optimierungsproblem kann mit dem zu härtenden Material verglichen werden. Dabei entspricht die Zielfunktion des Problems dem Energiezustand des Feststoffs. Eine Lösung des Optimierungsproblems ist äquivalent zu einer bestimmten Molekülanordnung des Materials. Der Algorithmus ist

so konzipiert, dass er die physikalischen Vorgänge während eines Härtungsprozesses nachbildet und somit in der Lage ist, ein globales Optimum der Zielfunktion zu finden.

Das Verfahren des Simulated Annealing geht auf einen einfachen Algorithmus zurück, der schon 1953 von Metropolis et al. [27] beschrieben wurde. Dieser nach seinem Autor benannte *Metropolis-Algorithmus* beschreibt den in einem Feststoff stattfindenden Vorgang, wenn dieser bei einer bestimmten Temperatur in den Zustand des so genannten *thermischen Gleichgewichts*[8] übergeht. Ein solches Gleichgewicht ist erreicht, wenn sich der Zustand des Systems nicht mehr ändert [1].

Der Metropolis-Algorithmus erzeugt nun nach dem folgenden Prinzip eine Folge von Zuständen, die der Feststoff annehmen kann. Ausgehend von einem Zustand i und dem Energieniveau E_i des Stoffs wird ein nachfolgender, zufälliger Zustand j erzeugt, indem mit einem bestimmten Mechanismus eine minimale Störung an i vollzogen wird. Eine solche Störung kann bspw. durch die willkürliche Bewegung eines Moleküls an eine andere Stelle erfolgen. Dadurch erreicht das gesamte System ein neues Energieniveau E_j. Der nächste Zustand j wird nur unter einer bestimmten Voraussetzung angenommen. Ist die Energiedifferenz $E_j - E_i$ kleiner oder gleich Null, so wird der neue Zustand akzeptiert. In diesem Fall erreicht das System einen niedrigeren Energiezustand; es bewegt sich also auf ein – möglicherweise lokales – Minimum zu. Ist die Energiedifferenz größer als Null, so wird der neue Zustand nur mit einer bestimmten Wahrscheinlichkeit angenommen. Diese Wahrscheinlichkeit ist durch das so genannte *Metropolis-Kriterium*

$$\exp\left(\frac{E_i - E_j}{k_B T}\right)$$

gegeben. Hierbei steht T für die Temperatur des Feststoffs und k_B für die Boltzmann-Konstante. Man sieht also, dass mit abnehmender Temperatur die Wahrscheinlichkeit, mit der das System ein höheres Energieniveau annimmt, ebenfalls sinkt.

Das thermische Gleichgewicht, das ein Feststoff für eine gegebene Temperatur nach einer ausreichend großen Anzahl von Zustandsübergängen annimmt, wird durch die Boltzmann-Verteilung bestimmt. Diese gibt für eine gegebene Temperatur T die Wahrscheinlichkeit an, mit der sich ein Feststoff im Zustand i und der Energie E_i befindet. Sie ist gegeben durch

$$\mathfrak{P}_T\{\mathbf{X} = i\} = \frac{1}{Z(T)} \exp\left(\frac{-E_i}{k_B T}\right).$$

Hierbei ist $\mathbf{X}$ die Zufallsvariable, die den aktuellen Zustand des Systems angibt, und $Z(T)$ ist eine Funktion

$$Z(T) = \sum_j \exp\left(\frac{-E_i}{k_B T}\right),$$

bei der die Summe über alle möglichen Zustände iteriert. Die Wahrscheinlichkeit $\mathfrak{P}_T$ ist genau dann gültig, wenn sich das System im thermischen Gleichgewicht befindet [1].

Mit Hilfe des Metropolis-Algorithmus lässt sich ein Verfahren entwickeln, mit dem Optimierungsprobleme gelöst werden können [43]. Zunächst wird dafür ein Kontrollparameter $c \in \mathbb{R}^+$ definiert, der die Rolle der Temperatur im oben beschriebenen Mecha-

[8] engl.: *thermal equilibrium*

nismus übernimmt. Der Algorithmus des Simulated Annealing besteht darin, den Kontrollparameter c schrittweise zu verringern und dabei für jeden Schritt den Metropolis-Algorithmus auszuführen [1].

Weiter wird eine Nachbarschaftsstruktur $\mathcal{N}(i)$ und ein Mechanismus zur Generierung einer zufälligen Nachbarlösung $j \in \mathcal{N}(i)$ benötigt. Seien nun eine Lösung i und dessen Nachbarlösung $\mathcal{N}(i) = j$ mit den Zielfunktionswerten $f(i)$ und $f(j)$ gegeben. Dann wird j als die nächste Lösung mit der folgenden Wahrscheinlichkeit akzeptiert:

$$\mathfrak{P}_c\{j \text{ wird akzeptiert}\} = \begin{cases} 1 & \text{falls } f(j) \leq f(i) \\ \exp\left(\frac{f(i)-f(j)}{c}\right) & \text{falls } f(j) > f(i) \end{cases} \tag{2.4}$$

Eine *Transition* ist definiert als ein zweistufiger Vorgang, bei dem die aktuelle Lösung i in eine nachfolgende Lösung j überführt wird. Dafür wird zuerst mit dem Mechanismus zur Generierung einer Nachbarlösung $j = \mathcal{N}(i)$ erzeugt. Schließlich wird das Akzeptanzkriterium (2.4) angewandt, um zu bestimmen, ob j als neue Lösung akzeptiert wird.

Der Algorithmus Simulated Annealing kann nun folgendermaßen beschrieben werden. Man wählt zu Beginn eine Anfangstemperatur c_0, die möglichst so groß sein sollte, dass das Akzeptanzkriterium alle vorgeschlagenen Transitionen zulässt. Dies entspricht dem geschmolzenen Zustand eines Materials, bei dem sich die Moleküle frei anordnen können. Dann erzeugt man für den aktuellen Wert des Kontrollparameters eine Folge von Transitionen bis sich das thermische Gleichgewicht einstellt. Dieses ist dann erreicht, wenn keine weiteren Transitionen mehr für die aktuelle Temperaturstufe akzeptiert werden.

Das Verfahren arbeitet währenddessen mit einer einzigen Repräsentation der aktuellen Lösung. Zu Beginn des Annealing-Prozesses wird diese mit einer zufälligen Startlösung initialisiert. Wie dies im Einzelnen geschieht, spielt für die Funktionalität des Simulated Annealing keine Rolle. Möchte man mit dem Verfahren ein Scheduling-Problem lösen, könnte man bspw. alle vorhandenen Aufträge auf eine beliebige Maschine legen oder man verteilt alle Jobs auf zufällig gewählte Maschinen.

Für die Behandlung von Scheduling-Problemen bietet sich die folgende Methode zur Ermittlung eines Nachbarn für die Lösung i an. Die Lösung i besteht hier aus einem vollständigen Maschinenbelegungsplan. Die Nachbarschaft für i wird nun durch all diejenigen Schedules gebildet, die sich nur in der Maschinenzuordnung von einem einzigen Auftrag von i unterscheiden, ansonsten aber identisch zu i sind. Für ein Scheduling-Problem mit n Jobs und m Maschinen ist damit die Größe der Nachbarschaft von i durch $|\mathcal{N}(i)| = n(m-1)$ gegeben. Aus dieser Nachbarschaft lässt sich nun eine beliebige Lösung bestimmen, indem man einen zufälligen Auftrag J_i ermittelt, diesen von der ihm eigentlich zugewiesenen Maschine entfernt und ihn auf eine andere, zufällig gewählte Maschine legt.

Nach der Durchführung einer ausreichenden Zahl von Transitionen kann c um einen geringen Prozentsatz δ ($\delta \in]0,1[$) verringert und daraufhin eine neue Folge von Transitionen berechnet werden. Die Temperatur von Schritt $k+1$ ergibt sich dann als $c_{k+1} = \delta \cdot c_k$. Dies wird so lange wiederholt, bis der Kontrollparameter unter einen vorgegebenen Schwellenwert c_{stop} sinkt, was dem Erstarren des Materials entspricht. Der Vorgang kann nun abgebrochen werden. Bei der richtigen Wahl der nötigen Para-

meter für den Algorithmus hat man jetzt das globale Optimum gefunden. Algorithmus 2 fasst das Verfahren zusammen. Darin bezeichnen für die k-te Iteration die Variablen c_k den Wert des Kontrollparameters und L_k die Anzahl der erzeugten Transitionen des Metropolis-Algorithmus.

Algorithmus 2 Simulated Annealing

```
1:  Initialisiere(i_start, c_0, L_0);
2:  k := 0;
3:  i := i_start;
4:  repeat
5:    for l := 1 to L_k do
6:      Erzeuge(j aus S_i);
7:      if f(j) ≤ f(i) then
8:        i := j;
9:      else
10:       if exp((f(i)−f(j))/c_k) > random[0, 1) then
11:         i := j;
12:       end if
13:     end if
14:   end for
15:   k := k + 1;
16:   BerechneLänge(L_k);
17:   BerechneKontrollparameter(c_k);
18: until c_k < c_stop
```

Anhand des Akzeptanzkriteriums (2.4) lässt sich die generelle Arbeitsweise des Verfahrens erläutern. Man sieht, dass Transitionen, die den Zielfunktionswert verbessern, bedingungslos akzeptiert werden. Dies entspricht der gleichen Vorgehensweise, wie sie bei der lokalen Suche angewandt wird. Während dort allerdings Transitionen, die zu einer Verschlechterung des Zielfunktionswertes führen, verworfen werden, können diese bei der simulierten Abkühlung dennoch akzeptiert werden.

Betrachtet man Gleichung (2.4) genauer, kann man leicht zwei Auswirkungen erkennen, die das Akzeptanzkriterium auf den Algorithmus hat. Zum einen ist die Akzeptanzwahrscheinlichkeit umso kleiner, je größer die Differenz der Zielfunktionswerte $|f(i)-f(j)|$ zweier benachbarter Lösungen wird. Je stärker also die Verschlechterung der neuen Lösung ist, desto unwahrscheinlicher ist es, dass sie angenommen wird. Der Kontrollparameter c hat darauf einen verstärkenden Einfluss. Ist die Temperatur nämlich niedrig und c damit klein, fällt diese Wahrscheinlichkeit entsprechend geringer aus. Das heißt für den Algorithmus, dass bei hohen Temperaturen noch große Verschlechterungen des Zielfunktionswerts möglich sind, während bei niedriger werdenden Temperaturen nur noch geringe Verschlechterungen angenommen werden.

Aufgrund dieser Eigenschaft kann man beobachten, wie sich bei hohen Temperaturen die groben Strukturen des Optimierungsproblems herausbilden. Erst mit fortschreitender Abkühlung formieren sich die Feinheiten der Problemlösung. In Abbildung 2.5 wird

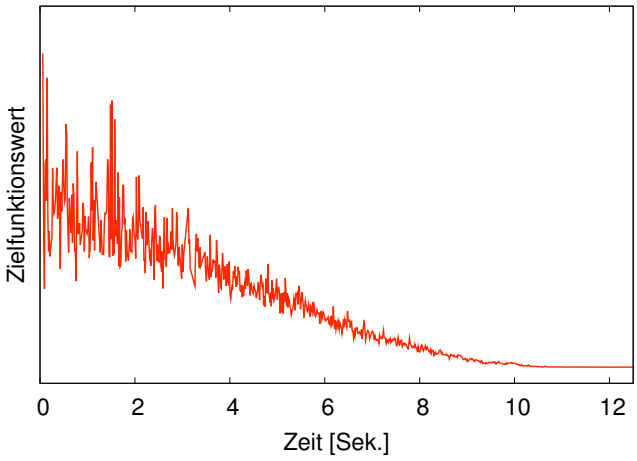


Abbildung 2.5: Beispiel für die Entwicklung der Zielfunktion beim Simulated Annealing

dies grafisch dargestellt. Hier wurde für ein beispielhaftes Scheduling-Problem auf einer Zeitachse abgetragen, wie sich der Zielfunktionswert während des Annealing-Prozesses entwickelt. Man sieht wie am Anfang bei hohen Temperaturen noch sehr große Verschlechterungen des Zielfunktionswertes möglich sind: die Kurve schlägt sehr stark aus. Mit kleiner werdendem Kontrollparameter c nähert sich das Verfahren dann allmählich einer optimalen Lösung an. Dabei werden nur noch geringe Verschlechterungen akzeptiert. Man kann erkennen, dass keine Änderungen an der Lösung mehr stattfinden, nachdem etwa zehn Sekunden vergangen sind. Dies deutet darauf hin, dass das Verfahren ein Optimum gefunden hat und aufgrund des niedrigen Kontrollparameters keine Verschlechterungen mehr zulässt.

2.2.2.3 Branch And Bound

Die *Branch And Bound-Methode* ist ein enumeratives Verfahren, mit dem sich der gesamte Lösungsraum S des behandelten Problems zielgerichtet und systematisch durchsuchen lässt [25, 37]. Um zu verhindern, dass durch eine vollständige Enumeration alle möglichen Lösungen eines Optimierungsproblems untersucht werden müssen, erlaubt der Algorithmus das Überspringen von Teilmengen von S. Wenn für eine solche Teilmenge festgestellt werden konnte, dass die gesuchte Lösung nicht in ihr enthalten sein kann, braucht diese nicht weiter untersucht zu werden. Dadurch wird es möglich, große Bereiche des Lösungsraums von der Durchsuchung auszuschließen und damit den Rechenaufwand des Verfahrens drastisch zu verringern. In der folgenden Beschreibung des Algorithmus wird davon ausgegangen, dass die Zielfunktion des zu lösenden Optimierungsproblems minimiert werden soll. Der Maximierungsfall verläuft analog.

Der Algorithmus Branch And Bound besteht aus zwei Komponenten: dem Aufteilen des Gesamtproblems in Unterprobleme[9] und der Berechnung von Ober- und Untergren-

[9] engl.: *branching*

zen[10].

Für den ersten Schritt des Algorithmus wird ein rekursives Verfahren benötigt, mit dem sich das ursprüngliche Problem in Teilprobleme unterteilen lässt. Die Teilprobleme müssen sich nach derselben Methode wiederum in Unterprobleme aufspalten lassen. Als Ergebnis dieses Prozesses erhält man eine Baumstruktur, an deren Blättern sich alle möglichen Lösungen des behandelten Problems befinden. Die Knoten des Baumes repräsentieren Teillösungen, die wiederum als Grundlage für die an den Knoten hängenden Unterbäume dienen.

Für ein Scheduling-Problem bietet sich die folgende Branching-Methode an. Auf der k-ten Ebene des Lösungsbaums wird der Auftrag J_k auf jede der verfügbaren Maschinen verteilt. Jeder Knoten auf dieser Ebene steht dabei für eine der m Maschinen. Auf diese Weise erhält man für jeden Knoten einen Teilschedule, bei dem die ersten k Jobs auf die Maschinen verteilt wurden. Der Pfad von der Wurzel zu einem der inneren Knoten des Baumes legt dabei fest, wie ein solcher Teilschedule aufgebaut ist.

Dieser Lösungsbaum wird bei der Branch And Bound-Methode nach einer optimalen Lösung durchsucht. Das Verfahren wandert dabei ausgehend von der Wurzel systematisch durch den Baum zu einem Blattelement, wo es das Optimum vorzufinden hofft.

Damit nicht jedes der vorhandenen Blätter besucht werden muss – was einer vollständigen Enumeration entspräche – wird ein Teil der Unterbäume von der Durchsuchung ausgeschlossen. Dies geschieht anhand der Berechnung von Unter- und Obergrenzen für die Zielfunktion.

Untergrenzen werden für jeden Knoten ermittelt, an dem sich der Algorithmus gerade befindet. Diese Untergrenze sagt für die Teillösung an einem Knoten aus, wie niedrig der Zielfunktionswert für alle vollständigen Lösungen in dem Unterbaum dieses Knotens höchstens werden kann. Die genaueste Untergrenze für ein Teilproblem bekäme man, wenn man den kleinsten Wert berechnen könnte, den die Zielfunktion für dieses Teilproblem annehmen kann. Dies würde allerdings wieder dem Ursprungsproblem entsprechen, zu dessen Lösung man die Branch And Bound-Methode überhaupt anwendet.

Um dennoch eine Untergrenze zu erhalten, kann man eine *Relaxation* der Randbedingungen des zu lösenden Optimierungsproblems durchführen und anschließend eine schnell zu berechnende Heuristik für dieses weniger strikte Problem anwenden. Relaxation bedeutet in diesem Fall, dass man eine der Nebenbedingungen des Optimierungsproblems streicht und man somit eine einfachere Variante des ursprünglichen Problems erhält [9].

Eine mögliche Relaxation für ein Scheduling-Problem ist das Zulassen von Preemption. Dadurch, dass Jobs während ihrer Bearbeitung unterbrochen und auf einer anderen Maschine weiterbearbeitet werden können, lassen sich Scheduling-Probleme durch einfache Heuristiken berechnen.

Hat man eine Methode zur Berechnung der Untergrenzen, kann man nun entscheiden, ob eine Teillösung weiter durchsucht werden muss, oder ob man den Unterbaum von einer weiteren Durchsuchung ausschließen kann. Dies geschieht mit Hilfe einer Obergrenze. Das Verfahren merkt sich dafür zu jedem Zeitpunkt seiner Ausführung die be-

[10] engl.: *upper* und *lower bounds*

ste bis dahin gefundene Lösung. Der Zielfunktionswert dieser aktuellen Lösung, welche auch als *Incumbent* bezeichnet wird, bildet die benötigte Obergrenze. Befindet sich der Algorithmus an einem beliebigen Knoten des Suchbaums, so berechnet er zuerst die Untergrenze für den Unterbaum an diesem Knoten. Ist diese Untergrenze größer als der Zielfunktionswert der aktuellen Lösung, so kann der Unterbaum aus der weiteren Suche ausgeschlossen werden. Es ist nicht möglich, darunter noch die gesuchte optimale Lösung zu finden.

Algorithmus 3 Branch And Bound

```
1:  LIST := {S};
2:  Incumbent := durch Heuristik ermittelte Lösung;
3:  UB := f(Incumbent);
4:  while LIST ≠ ∅ do
5:    Wähle einen Knoten k aus LIST;
6:    Entferne k aus LIST;
7:    Erzeuge die Kindknoten child(i) für alle i = 1, ..., n_k und berechne die jeweiligen
      Untergrenzen LB_i;
8:    for i := 1 to n_k do
9:      if LB_i < UB then
10:       if child(i) besteht aus einer einzigen, vollständigen Lösung then
11:         UB := LB_i;
12:         Incumbent := die Lösung unter child(i);
13:       else
14:         Füge child(i) zu LIST hinzu;
15:       end if
16:     end if
17:   end for
18: end while
```

Die Arbeitsweise der Branch And Bound-Methode lässt sich wie in Algorithmus 3 dargestellt umreißen. Der oben beschriebene Suchbaum wird während der Laufzeit des Verfahrens dynamisch erzeugt. Dazu werden in der Variable $LIST$ diejenigen Knoten des Baums vermerkt, die noch weiter durchsucht werden müssen. Diese Liste wird zu Beginn mit dem Wurzelknoten des Suchbaums initialisiert.

Im zweiten Schritt wird die aktuelle Lösung „$Incumbent$“ mit einer Startlösung belegt. Hier kann bspw. eine Heuristik verwendet werden, mit der man eine gute Näherungslösung für das behandelte Problem erhalten kann. Die Variable für die Obergrenze UB enthält den Zielfunktionswert $f(Incumbent)$ der aktuellen Lösung.

Als nächstes wird der Suchbaum durchlaufen. Dazu wird ein Knoten aus $LIST$ gewählt und die Untergrenzen für dessen Kindknoten berechnet. Ist nun die Untergrenze LB_i eines Kindknotens i größer als die aktuelle Obergrenze UB, so wird dieser Knoten verworfen und von der weiteren Durchsuchung ausgeschlossen. Es steht für ihn fest, dass sich darunter keine bessere Lösung als „$Incumbent$“ finden lässt. Ist die Untergrenze jedoch kleiner als die aktuelle Obergrenze, so wird eine Fallunterscheidung gemacht. Hat

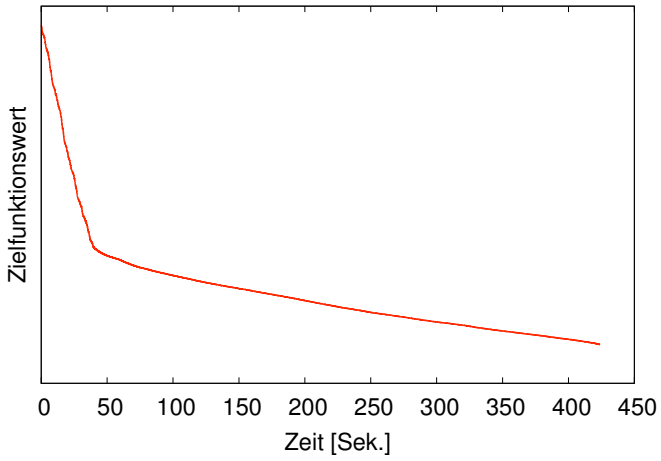


Abbildung 2.6: Beispiel für die Entwicklung der Zielfunktion bei der Branch And Bound-Methode

der Algorithmus mit dem aktuellen Knoten ein Blattelement des Suchbaums erreicht, dann heißt dies, dass er eine vollständige Lösung gefunden hat. Deren Zielfunktionswert (gegeben durch dessen Untergrenze) ist besser als der Zielfunktionswert der aktuellen Lösung. Man hat also eine neue, bessere Lösung gefunden. Die Variable „*Incumbent*“ wird entsprechend auf die Lösung gesetzt, die von diesem Blattelement repräsentiert wird. Befindet sich der Algorithmus hingegen noch innerhalb des Suchbaums, so besagt die niedrigere Untergrenze, dass sich in dem Teilbaum unter dem Knoten theoretisch noch eine bessere Lösung als „*Incumbent*“ befinden kann. Der Unterbaum muss daher weiter untersucht werden. Er wird dafür in der Suchliste vermerkt.

Hat man für einen Knoten die Untergrenzen für alle Kindknoten berechnet und die weiter zu untersuchenden Kindelemente in der Suchliste vermerkt, stellt sich nun die Frage, nach welchem Schema man in Schritt 5 von Algorithmus 3 den nächsten zu untersuchenden Knoten k auswählen soll. Es bieten sich dafür drei Methoden an: eine *Breadth-First-*, eine *Depth-First-* oder eine *Best-First*-Suche. Bei der ersten Variante werden zuerst alle Knoten einer bestimmten Ebene des Suchbaums abgearbeitet bevor in die nächsttiefere Ebene abgestiegen wird. Bei der Depth-First-Suche steigt das Verfahren dagegen sofort in den Suchbaum hinab und versucht, zunächst ein Blattelement zu erreichen. Die Best-First-Suche schließlich sortiert die Kindelemente eines Knotens nach aufsteigendem Zielfunktionswert und wählt für den nächsten Schritt denjenigen Knoten, der den kleinsten Zielfunktionswert besitzt und damit am vielversprechendsten ist.

Ein kleines Beispielproblem, das mit Hilfe des Branch And Bound-Algorithmus gelöst wird, ist in Abbildung 2.7 dargestellt. Hier sollen insgesamt fünf Aufträge auf drei Maschinen verteilt werden. Die Kapazitäten der einzelnen Maschinen sind mit jeweils $1,0$, $2,0$ und $3,0$ angegeben. Die einzelnen Aufträge haben einen Bearbeitungsaufwand von jeweils $1,0$, $2,0$, $3,0$, $4,0$ und $6,0$. Für die Berechnung der Bearbeitungsdauer eines Auftrags auf einer Maschine wird im Folgenden der Kapazitätswert der Maschine mit dem

Bearbeitungsaufwand des Auftrags multipliziert. In Abbildung 2.7(a) ist die Startlösung dargestellt, mit der der Branch And Bound-Algorithmus initialisiert wird. Dort wurde z. B. der Auftrag mit Aufwand $4,0$ auf die Maschine 1 gelegt. Damit ergibt sich für diesen Job eine Bearbeitungsdauer von $12,0$ Zeiteinheiten. Die gesamte Startlösung hat einen Zielfunktionswert von $15,0$. Dieser dient nun dem Algorithmus als Obergrenze UB.

Abbildung 2.7(b) zeigt den Weg des Verfahrens durch den Suchbaum. Auf der linken Seite sind die jeweiligen Teillösungen abgebildet, die sich nach jedem Schritt ergeben. Im ersten Schritt des Verfahrens wird versucht, den Auftrag mit Aufwand $6,0$ auf eine Maschine zu legen. Dabei errechnet sich für die erste Maschine eine Untergrenze von $LB_1 = 18,0$, für die zweite die Untergrenze $LB_2 = 12,0$ und für die dritte Maschine ein Untergrenze von $LB_3 = 8,7$ Zeiteinheiten[11]. Die Werte für die einzelnen LB_i sind in den jeweiligen Knoten des Suchbaums vermerkt.

Die erste Untergrenze mit $LB_1 = 18,0$ ist größer als die aktuelle Obergrenze. Der Unterbaum unter diesem Knoten kann daher von der weiteren Untersuchung ausgeschlossen werden. Die beiden anderen Untergrenzen LB_2 und LB_3 sind hingegen kleiner als UB. Die entsprechenden Unterbäume müssen also weiter durchsucht werden. Da der Algorithmus in diesem Beispiel nach der Best-First-Suche vorgeht, wählt er den Knoten mit der Untergrenze $8,7$ als die nächste zu durchsuchende Teillösung.

Auf diese Weise fährt der Algorithmus fort, bis er schließlich an einem Blattknoten ankommt. Der zuerst erreichte Blattknoten hat eine Untergrenze – und als ein solcher Knoten damit einen Zielfunktionswert – von $12,0$. Dieser Wert ergibt sich, indem der letzte Auftrag mit einem Aufwand von $1,0$ auf Maschine 1 gelegt wird. Dies ist in Abbildung 2.7(b) im untersten Gantt-Diagramm durch den gestrichelten Auftragsbalken auf Maschine 1 angedeutet. Da sich der Algorithmus nun an einem Blattknoten befindet, dessen Zielfunktionswert kleiner ist als die aktuelle Obergrenze, wird die an diesem Knoten gefundene Lösung als die neue aktuelle Lösung „$Incumbent$“ gesetzt. Die neue Obergrenze wird damit zu $UB = 12,0$. Derselbe Vorgang geschieht auch bei den folgenden zwei Blattknoten. Der letzte Knoten mit der Untergrenze $LB_3 = 9,0$ stellt eine optimale Lösung dar. Um diese Tatsache zu verifizieren, wird der Algorithmus nun durch den Suchbaum zurückwandern und alle zuvor für die weitere Durchsuchung markierten Knoten aus der Suchliste entfernen. Da diese durchweg keine kleinere Untergrenze als die nun aktuelle Obergrenze $UB = 9,0$ vorweisen können, brauchen die entsprechenden Unterbäume nicht mehr weiter untersucht werden.

Anhand der bisher erläuterten Arbeitsweise der Branch And Bound-Methode lässt sich eine wichtige Eigenschaft des Verfahrens hervorheben. Der Algorithmus hält zu jedem Zeitpunkt die beste bisher gefundene Lösung in der Variable „$Incumbent$“ bereit. Diese wird erst dann verworfen, sobald eine bessere Lösung gefunden wurde. Damit bewegt sich das Verfahren monoton auf eine optimale Lösung zu, ohne dass sich die bisher gefundene Lösung zwischendurch verschlechtert. Wenn der Algorithmus schließlich alle Elemente in $LIST$ abgearbeitet hat, befindet sich in der Variable „$Incumbent$“ eine optimale Lösung [25].

[11] Das Verfahren für die Berechnung dieser Werte wird in Anhang A genauer beschrieben.

Dieses Verhalten wird in Abbildung 2.6 anhand eines Beispiels dargestellt. Der Graph zeigt ebenso wie Abbildung 2.5 die Entwicklung des Zielfunktionswertes über die Zeit. Man sieht hier, wie der Algorithmus zu Beginn seiner Berechnung sehr schnell die aktuelle Lösung verbessert. Nach etwa 40 Sekunden ist der Zielfunktionswert soweit herabgesetzt, dass von diesem Zeitpunkt an nur noch selten eine bessere Lösung gefunden wird. Aus diesem Grund verläuft die Kurve dort flacher.

2.2.2.4 Greedy Scheduling

Der Algorithmus *Greedy Scheduling* geht nach einer sehr einfachen Heuristik vor, um das Scheduling-Problem zu lösen. Typisch für „gierige“ (*greedy*) Algorithmen ist, dass sie je nach Anwendungsfall versuchen, entweder eine vorhandene Ressource so gut es geht zu schonen oder verfügbare Mittel so weit wie möglich auszunutzen oder zu vereinnahmen. Sie hoffen dabei, das globale Optimum zu finden, indem sie lokal die momentan beste zu findende Lösung wählen [10, 48]. Der in diesem Abschnitt vorgestellte Algorithmus gehört zu der ersten, „geizigen“ Variante. Das Verfahren versucht hier, mit der abstrakten Ressource „Produktionsdauerzuwachs“ optimal zu haushalten.

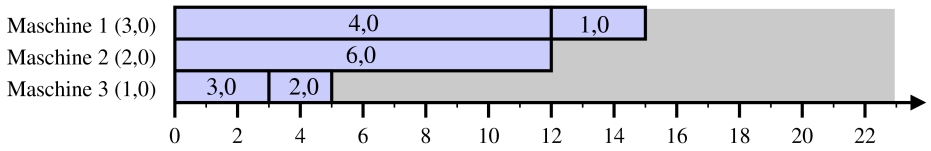
Der Algorithmus sortiert zu Beginn seiner Arbeit alle zu verteilenden Aufträge nach absteigendem Bearbeitungsaufwand. Dadurch werden zuerst die Aufträge mit dem größten Aufwand auf die Maschinen verteilt. In der k-ten Iteration hat der Algorithmus einen Teilschedule berechnet, bei dem alle Jobs $J_l,\ l = 1, \ldots, k-1$ auf die verfügbaren Maschinen verteilt worden sind. Für diesen Teilschedule ist der Zeitpunkt C_k bekannt, zu dem dieser Plan beendet sein wird. Es wird nun für jede der Maschinen M_j $(j = 1, \ldots, m)$ berechnet, wann diese mit der Auftragsbearbeitung fertig werden würden, wenn man den Job J_k auf sie legt. Dieser Zeitpunkt sei mit P_{kj} bezeichnet. Anhand dieser Daten lässt sich anschließend ermitteln, wie groß der Zuwachs $\Delta_{C_{kj}}$ der Produktionsdauer für die einzelnen P_{kj} ausfallen würde:

$$\Delta_{C_{kj}} = P_{kj} - C_k, \quad j = 1, \ldots, m \tag{2.5}$$

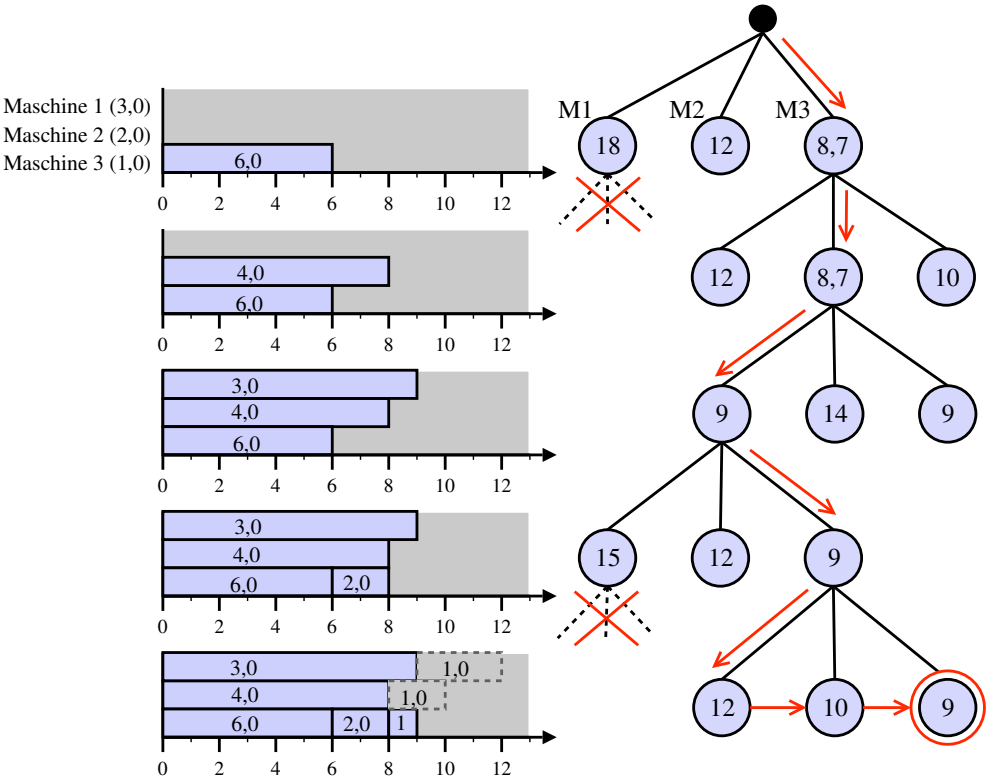
Der Scheduling-Algorithmus wählt nun diejenige Maschine für den aktuellen Auftrag J_k aus, bei der $\Delta_{C_{kj}}$ minimal ist. Algorithmus 4 auf Seite 34 fasst das Verfahren noch einmal zusammen.

Abschließend muss für diese Methode hervorgehoben werden, dass sie nicht in der Lage ist, für jedes gestellte Problem eine optimale Lösung zu finden. Die Heuristik führt vielmehr zu einer Lösung, die in der Nähe eines Optimums liegt.

Abbildung 2.8 zeigt ein Beispiel für den Greedy Scheduler. Hier sollen die Aufträge mit einem Bearbeitungsaufwand von jeweils $2,0$, $3,0$, $4,0$ und $6,0$ auf die drei Maschinen aus dem Beispiel für den Branch And Bound-Algorithmus verteilt werden. In Abbildung 2.8 wurden schon die ersten drei Aufträge mit dem Greedy Scheduler auf die Maschinen verteilt. Das Verfahren befindet sich also im vierten Schritt mit $i = 3$. Auch hier berechnet sich wieder die Bearbeitungsdauer eines Auftrags auf einer Maschine durch die Multiplikation der Maschinenkapazität mit dem Bearbeitungsaufwand des Auftrags. Der aktuelle Teilschedule hat damit eine maximale Bearbeitungsdauer von $C_3 = 9,0$ Zeiteinheiten.



(a) Zufällige Startlösung (*Incumbent*) für den Branch And Bound-Scheduler mit dem Zielfunktionswert $f(Incumbent) = 15$



(b) Gantt-Diagramme der Teillösungen jeder Ebene und Suchpfad des Algorithmus im Teillösungsbaum

Abbildung 2.7: Beispielproblem für den Branch And Bound-Algorithmus. Die Knoten des Suchbaums sind mit den jeweiligen Untergrenzen LB_i beschriftet. Die roten Pfeile kennzeichnen den Weg durch den Suchbaum.

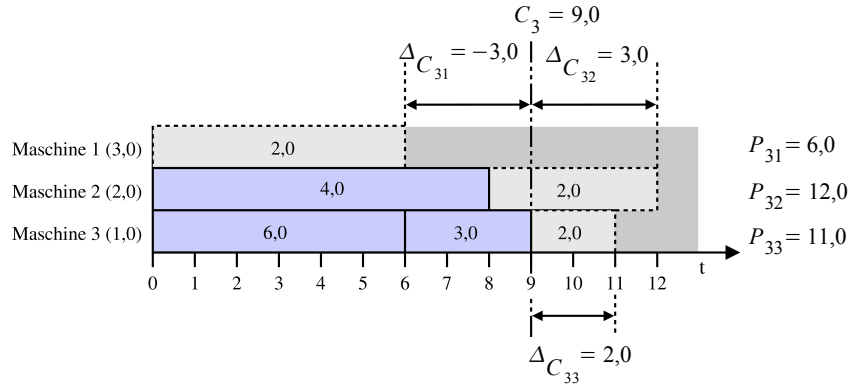


Abbildung 2.8: Beispiel für den Greedy Scheduling-Algorithmus

Es soll nun der nächste Auftrag mit dem Aufwand $2,0$ auf eine Maschine gelegt werden. Die zusätzliche Bearbeitungsdauer, die sich für jede Maschine ergibt, wenn ihr dieser Auftrag zugewiesen wird, wird in Abbildung 2.8 durch die hellgrau hinterlegten Balken mit gestrichelter Umrandung dargestellt. Für die einzelnen P_{kj} ergeben sich damit die Werte $P_{31} = 6,0$, $P_{32} = 12,0$ und $P_{33} = 11,0$. Setzt man diese Beträge in Gleichung (2.5) ein, so ergeben sich für die einzelnen $\Delta_{C_{kj}}$ die Werte $\Delta_{C_{31}} = -3,0$, $\Delta_{C_{32}} = 3,0$ und $\Delta_{C_{33}} = 2,0$. Hier hat man also mit $\Delta_{C_{31}}$ den kleinsten Produktionsdauerzuwachs. Der Auftrag mit dem Aufwand $2,0$ wird daher auf Maschine 1 gelegt.

Algorithmus 4 Greedy Scheduling

```
1: Sortiere alle Aufträge nach absteigendem Bearbeitungsaufwand;
2: Setze C_0 = 0;
3: for i := 0 to n do
4:    for j := 0 to m do
5:       Berechne Δ_{C_ij} = P_ij − C_i;
6:    end for
7:    Weise Auftrag J_i derjenigen Maschine M_μ zu, für die gilt Δ_{C_iμ} = min_{j=0}^{m}{Δ_{C_ij}};
8:    Berechne die neue Produktionsdauer C_{i+1};
9: end for
```

2.2.2.5 List Scheduling

Auch das *List Scheduling* ist eine einfache Heuristik zur Einplanung von Aufträgen auf Maschinen. Im Gegensatz zu den bisher erläuterten Algorithmen ist beim List Scheduling die Kenntnis der Bearbeitungsdauer p_i eines Auftrags nicht notwendig. Das Verfahren eignet sich daher insbesondere auch für Scheduling-Probleme, bei denen $p_i = 1$ für alle $i = 1, \ldots, n$ gilt. Vorgeschlagen wurde das List Scheduling ursprünglich 1966 von Graham in [14].

Bei diesem Verfahren wird zuerst die Liste der einzuplanenden Aufträge mit absteigender Priorität sortiert, falls dies möglich ist. Im nächsten Schritt sortiert man die Liste der verfügbaren Maschinen absteigend nach ihrer Geschwindigkeit. Falls nun ein Auftrag für die Bearbeitung freigegeben wird, wird dieser an die nächste freie Maschine aus der sortierten Liste zugewiesen. Es werden also immer möglichst die schnellsten Maschinen für die Bearbeitung der Jobs verwendet. Das Verfahren wird in Algorithmus 5 als Pseudocode zusammengefasst.

Algorithmus 5 List Scheduling

```
1: Sortiere alle Aufträge nach absteigendem Bearbeitungsaufwand;
2: Sortiere alle Maschinen nach absteigender Geschwindigkeit;
3: for i := 0 to n do
4:     Weise Auftrag J_i der nächstschnellsten Maschine zu;
5: end for
```

Kapitel 3

Konzeption und Analyse

3.1 Leistungsparameter der Scheduling-Algorithmen

Die im vorangegangenen Kapitel vorgestellten Scheduling-Verfahren wurden im Rahmen der vorliegenden Arbeit implementiert und ausgewertet. An dieser Stelle soll nun vorgestellt werden, wie sich die Algorithmen bei der Behandlung konkreter Optimierungsprobleme verhalten und welche Besonderheiten sie dabei aufweisen. Zunächst sollen einige Vorbemerkungen zu den durchgeführten Algorithmenauswertungen gemacht werden.

3.1.1 Vorbemerkungen

Der Hauptfokus bei der Beurteilung der Scheduling-Verfahren liegt im Folgenden auf zwei Leistungsparametern. Dies ist zum einen die Zeit, die die jeweiligen Methoden zur Lösung eines Optimierungsproblems benötigen. Üblicherweise wünscht man eine möglichst kurze Zeit für die Ermittlung einer optimalen Lösung. Die Algorithmen zeigen allerdings recht unterschiedliche Geschwindigkeiten, wie in den nächsten Abschnitten noch gezeigt werden soll. Es muss daher weiter untersucht werden, für welche Anwendungsfälle die jeweiligen Verfahren geeignet sind.

Der zweite Leistungsparameter, der an dieser Stelle vorgestellt und analysiert werden soll, gibt Auskunft über die Güte der gefundenen Lösung. Während manche Methoden, wie z. B. der Branch And Bound-Algorithmus, grundsätzlich dazu in der Lage sind, eine optimale Lösung für ein Optimierungsproblem zu finden, reichen andere Verfahren nur in die Nähe eines Optimums. Andere Methoden wiederum können zwar ein Optimum finden, brauchen dafür aber u. U. eine längere Zeit. Da die Zeit für das Finden einer Lösung für realistische Anwendungen einen kritischen Faktor darstellt, geht man üblicherweise bei solchen Algorithmen einen Kompromiss zwischen der Rechendauer des Verfahrens und der Güte der gefundenen Lösung ein. Damit man mit einem vertretbaren Zeitaufwand ein ausreichend gutes Ergebnis erhält, wird man sich daher auch mit einer Lösung zufrieden geben, die genügend nahe an das Optimum heranreicht.

Dieser zweite Leistungsparameter wird in Form eines Fehlermaßes ϵ ausgedrückt und nach der folgenden Vorschrift berechnet [1]:

$$\epsilon := \frac{f^* - f_{opt}}{f_{opt}} \tag{3.1}$$

Hierbei steht f^* für den Zielfunktionswert der Lösung, die von einem Scheduling-Algorithmus ermittelt wurde. Der Wert f_{opt} ist der Zielfunktionswert eines der gesuchten

Optima. Das Fehlermaß ϵ kann also als die prozentuale Abweichung der gefundenen Lösung vom eigentlichen Optimum gesehen werden.

Bei der Berechnung von ϵ ergibt sich eine Schwierigkeit. Für die Ermittlung von f_{opt} müsste man eigentlich einen der Algorithmen, die wie Branch And Bound immer ein Optimum finden können, solange suchen lassen, bis sie f_{opt} ausgeben. Dies würde in den meisten Fällen allerdings eine viel zu lange Rechenzeit in Anspruch nehmen, so dass die Berechnung eines genauen Optimums f_{opt} unmöglich wird. Aus diesem Grund wird – wie auch schon bei der Berechnung der Untergrenze für den Branch And Bound-Algorithmus – auf eine Relaxation der Ausgangsbedingungen für das Optimierungsproblem zurückgegriffen. Lässt man nämlich die Bedingung fallen, nach der Preemption nicht erlaubt ist, kann man für diesen einfacheren Fall ein alternatives $\hat{f}_{opt}$ sehr leicht nach der Formel

$$\hat{f}_{opt} = \frac{\sum_{i=0}^{n} p_i}{\sum_{j=0}^{m} \frac{1}{s_j}}$$

berechnen. Dieser Wert entspricht dem Zielfunktionswert des *theoretischen Optimums*. Hier stehen p_i für den Bearbeitungsaufwand von Auftrag J_i und s_j für die Geschwindigkeit von Maschine M_j. Es soll an dieser Stelle angemerkt werden, dass die Bearbeitungsdauer π_{ij} eines Auftrags J_i auf einer Maschine M_j in der vorliegenden Arbeit mit $\pi_{ij} = p_i \cdot s_j$ berechnet wird. Auf diese Tatsache wird an späterer Stelle noch genauer eingegangen.

Ein Nachteil dieser Methode ist die Tatsache, dass für ein Optimierungsproblem immer

$$\hat{f}_{opt} \leq f_{opt} \tag{3.2}$$

gilt; eine Lösung mit erlaubtem Preemption kann also einen kleineren Zielfunktionswert haben als eine Lösung des ursprünglichen Problems. D. h., es gibt Fälle mit $\hat{f}_{opt} < f_{opt}$, bei denen das Fehlermaß ϵ für eine eigentlich optimale Lösung nicht den erwarteten Wert von 0 hat, wenn man für die Berechnung von ϵ das theoretische Optimum heranzieht. Die Gleichheit in Ungleichung (3.2) gilt nur dann, wenn bei einer gefundenen Lösung alle Maschinen ihre Aufträge zur gleichen Zeit beendet haben, ohne dass dabei auf Preemption zurückgegriffen werden musste. In diesem Fall entspricht das tatsächlich erreichbare Optimum dem theoretischen Optimum.

Für die Untersuchung der in Abschnitt 2.2 vorgestellten Algorithmen wurden die einzelnen Verfahren für unterschiedliche Problemfälle ausgeführt. Dabei wurden die beiden Leistungsparameter ermittelt und in Graphen dargestellt. Eine Probleminstanz für die Scheduling-Methoden besteht aus einer bestimmten Anzahl von unterschiedlich leistungsfähigen Maschinen, auf die mit Hilfe der Algorithmen eine Menge von verschieden arbeitsintensiven Aufträgen verteilt werden soll.

Die verwendeten Probleminstanzen wurden in drei Gruppen unterteilt. In die erste Gruppe fallen alle Aufgabenstellungen, bei denen nur wenige Jobs auf eine kleine Anzahl von Maschinen verteilt werden sollen. Die zweite Gruppe enthält alle Probleminstanzen, bei denen die Auftrags- und Maschinenzahlen noch moderate Größen haben. Besonders umfangreiche Aufgabenstellungen mit Auftragsgrößen von bis zu 30.000 Jobs bei bis zu 500 Maschinen fallen in die dritte Gruppe.

Die Geschwindigkeiten der mit Aufträgen zu belegenden Maschinen wurden für jeden Problemfall nach dem Zufallsprinzip gewählt. Im Gegensatz dazu orientiert sich der Bearbeitungsaufwand der zu verteilenden Aufträge nach real gemessenen Rechenzeiten. Es wurden dafür etwa 1.800 Simulationen mit dem Netzwerksimulator ns-2 durchgeführt. Für jeden dieser Aufrufe von ns-2 wurde ein eigener Satz von Parametern verwendet, so dass eine bestimmte Simulation nicht mehr als einmal ausgeführt wurde. Der für jeden ns-2-Lauf gemessene Zeitbedarf dient im Folgenden als Aufwand p_i der zu verteilenden Aufträge. Dadurch konnten die Scheduling-Algorithmen mit realitätsnahen Daten für den Bearbeitungsaufwand der einzelnen Jobs getestet werden.

Die Graphen, mit denen in den nachfolgenden Abschnitten die Leistungsparameter der verschiedenen Scheduling-Algorithmen illustriert werden sollen, sind wie folgt aufgebaut. Auf der y-Achse wurden die jeweiligen Leistungsparameter abgetragen. Die Ausführungsdauer der Algorithmen wird dabei in Sekunden und das Fehlermaß ϵ in Prozentpunkten angegeben. Auf der x-Achse werden die Anzahl der eingeplanten Aufträge für die einzelnen Probleminstanzen abgetragen. So bedeuten bspw. die Datenpunkte, die in Abbildung 3.1(b) für eine Anzahl von 20 Maschinen eingezeichnet sind, wie lange das Berechnen eines Maschinenbelegungsplans gedauert hat, wenn unterschiedliche Auftragsgrößen im Bereich von 20 bis 100 Jobs auf 20 Maschinen verteilt wurden.

Die Messpunkte für die einzelnen Probleminstanzen wurden ausschließlich zur besseren Erkennbarkeit mit Linien verbunden. Es darf bei den Graphen nicht angenommen werden, dass man die Werte zwischen den Messpunkten linear interpolieren kann.

Weiterhin soll bemerkt werden, dass bei diesen Leistungstests den Algorithmen immer mindestens genausoviele Maschinen zur Verfügung gestellt wurden, wie Aufträge verteilt werden sollten. Aus diesem Grund finden sich z. B. für den oben erwähnten Testfall von 20 Maschinen keine Auftragsmengen von weniger als 20 Jobs.

3.1.2 Branch And Bound

Die Implementierung der Branch And Bound-Methode wurde nach den folgenden Gesichtspunkten vorgenommen. Um zu erreichen, dass der Algorithmus möglichst schnell gegen die optimale Lösung konvergiert, wurde das Verzweigen (*Branching*) mit Hilfe der Best-First-Suche umgesetzt. Es wird also, nachdem für einen Knoten im Suchbaum die Untergrenze eines jeden Kindknotens berechnet wurde, in denjenigen Zweig abgestiegen, der am vielversprechendsten für das Finden der optimalen Lösung ist. Die Untergrenzen werden berechnet, indem durch eine Relaxation der Nebenbedingungen das Verwenden von Preemption erlaubt wird. Das genaue Verfahren wird in Anhang A näher erläutert.

Wie in Algorithmus 3 dargestellt, benötigt die Branch And Bound-Methode eine Startlösung für die Berechnung der Obergrenze. Für die Ermittlung dieser Startlösung wurden abwechselnd die folgenden Verfahren verwendet: Greedy Scheduling, Simulated Annealing und ein Zufallsscheduler, der die einzelnen Aufträge auf zufällig gewählte Maschinen verteilt.

Bei der Durchführung der Leistungstests für die Branch And Bound-Methode ist aufgefallen, dass der Algorithmus schon bei relativ kleinen Probleminstanzen extrem lange Laufzeiten aufweist. Die Ursache dafür wurde in der Schätzfunktion der Untergrenze

ausgemacht. Diese hat die Eigenschaft, die Untergrenze systematisch zu unterschätzen. Da, wie schon anhand von Ungleichung (3.2) gezeigt wurde, Optimierungsprobleme mit erlaubter Preemption niedrigere Zielfunktionswerte für ihre optimalen Lösungen haben können als äquivalente diskrete Probleme ohne Preemption, können auch die Untergrenzen für die Teilprobleme zu niedrig eingeschätzt werden. Dies hat nun zur Folge, dass der Algorithmus Unterbäume erst dann von der weiteren Untersuchung ausschließen kann, wenn er schon sehr tief in den Suchbaum herabgestiegen ist. Im schlimmsten Fall muss der Algorithmus sogar erst bis zu einem Blattelement herabsteigen, bevor er feststellen kann, dass sich dort keine bessere Lösung befindet. Gleichzeitig verhindert die Unterschätzung der Untergrenze, dass das Verfahren erkennen kann, ob es schon eine optimale Lösung gefunden hat [25]. Tritt nämlich dieser Fall ein und der Algorithmus ist auf ein Optimum gestoßen, kann es für ihn wegen (3.2) dennoch so erscheinen, als wäre es möglich, im Rest des Suchbaums eine noch bessere Lösung zu finden. Anstatt also abzubrechen und die aktuelle Lösung als ein gefundenes Optimum auszugeben, läuft das Verfahren trotzdem unnötig weiter.

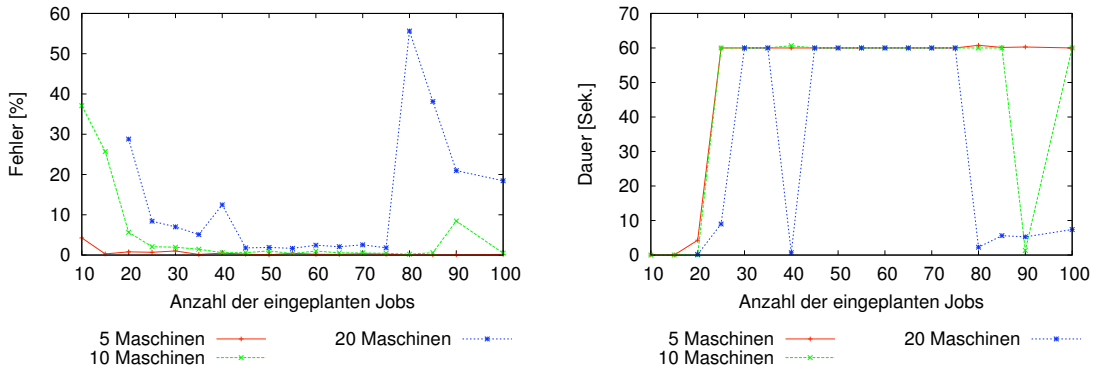
Um die Folgen dieser Eigenschaft abzuschwächen, wurden zwei Gegenmaßnahmen zur Beschränkung der Ausführungsdauer des Verfahrens ergriffen [25]. Zum einen ist es möglich, den Algorithmus nach einer bestimmten maximalen Rechendauer abbrechen zu lassen. Man erhält dann die bis zum Zeitpunkt des Abbruchs beste gefundene Lösung. Die Tatsache, dass sich das Verfahren monoton auf eine optimale Lösung zubewegt, ermöglicht diese Vorgehensweise.

Die zweite Variante zur Einschränkung der Rechendauer liegt darin, dem Algorithmus einen Toleranzparameter α zu übergeben. Mit diesem Parameter lässt sich festlegen, wie stark die gefundene Lösung vom theoretischen Optimum des Problems abweichen darf. Sinkt das Fehlermaß der bisher gefundenen Lösung unter den mit α vorgegebenen Wert, so wird die weitere Durchsuchung des Lösungsbaums ebenfalls abgebrochen.

Für die folgenden Untersuchungen der Branch And Bound-Methode wurde die maximal erlaubte Rechendauer auf 60 Sekunden gesetzt. Der Toleranzparameter α wurde zunächst nicht verwendet. Am Ende dieses Abschnitts wird kurz auf die Effekte eingegangen, die man mit einem α von 2% beobachten kann.

Abbildung 3.1 zeigt die Leistungsparameter der Branch And Bound-Methode für kleine Probleminstanzen. Hierbei wurde der Zufallsscheduler als Algorithmus für die Ermittlung der Startlösung verwendet. Anhand von Abbildung 3.1(b) kann man sehr gut erkennen, dass der Algorithmus für manche Problemfälle eine optimale Lösung findet. Das ist überall dort der Fall, wo die Suche nach weniger als 60 Sekunden abgeschlossen wurde. In dem dazu gehörenden Fehlergraphen 3.1(a) zeigt sich das in Abschnitt 3.1.1 beschriebene Verhalten: Man sieht z. B. für die Probleminstanz, bei der 40 Jobs auf 20 Maschinen verteilt wurden, dass ϵ bei etwa 12% liegt, obwohl der Algorithmus nach weniger als einer Sekunde abgeschlossen wurde. Hier fand das Verfahren also eine optimale Lösung, da es nicht vorzeitig abgebrochen werden musste. Der Zielfunktionswert der gefundenen optimalen Lösung ist allerdings größer als der Zielfunktionswert $\hat{f}_{opt}$ des theoretischen Optimums, bei dem Preemption erlaubt ist, woraus $\epsilon > 0$ resultiert.

Dasselbe Phänomen lässt sich auch bei 20 Maschinen und 80 Aufträgen beobachten.



(a) Fehlermaß ϵ

(b) Ausführungsdauer

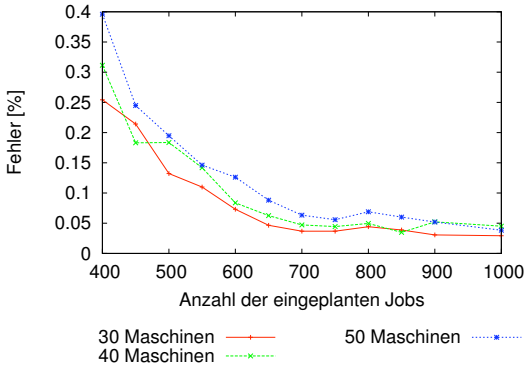
Abbildung 3.1: Ausführungsdauer und Fehlermaß für den Branch And Bound-Algorithmus bei kleinen Probleminstanzen. Die Startlösung wurde mit dem Zufallsscheduler berechnet.

Hier fällt der Fehlerwert ϵ mit etwa 55% noch extremer aus. Dies liegt darin begründet, dass bei Auftragsgrößen von mehr als 80 Jobs einige Aufträge mit einem sehr großen Bearbeitungsaufwand dazugekommen sind. Selbst wenn diese Aufträge auf den schnellsten Maschinen durchgeführt werden, haben die restlichen Jobs nicht genügend Bearbeitungsvolumen, um alle anderen Maschinen so lange auszulasten, dass sie zeitgleich mit den schnellsten Maschinen ihre Arbeit beenden können. Diese anderen, langsameren Maschinen müssen dann lange unbeschäftigt bleiben. Kann Preemption zum Einsatz kommen, so wäre dies nicht nötig.

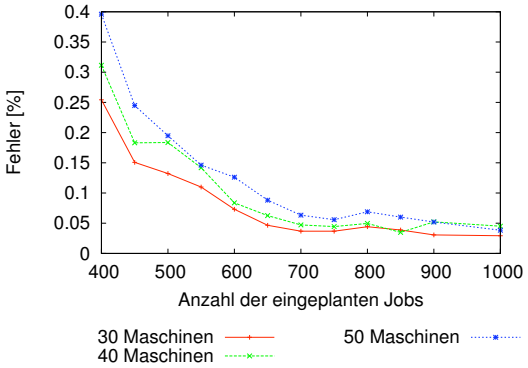
Verwendet man statt des Zufallsschedulers für die Startlösung der Branch And Bound-Methode das Simulated Annealing-Verfahren oder den Greedy Scheduler, erhält man die gleichen Graphen, wie in Abbildung 3.1 dargestellt. Bei solch kleinen Probleminstanzen spielt also die Wahl der Startlösung für den Branch And Bound-Scheduler keine Rolle, da der Algorithmus in der gegebenen Zeit immer auf dieselben Lösungen kommt. Die entsprechenden Graphen brauchen an dieser Stelle daher nicht weiter aufgeführt werden.

Ein ähnliches Bild ergibt sich bei Probleminstanzen mittlerer Größe. Die Abbildungen 3.2 und 3.3 zeigen für diese Fälle das Fehlermaß bei Verwendung der verschiedenen Initialisierungsalgorithmen. Man sieht, dass hier die Fehlerwerte etwa gleich groß sind. Hier kommt der Algorithmus also unabhängig von der Wahl des Initialisierers zu ähnlichen Ergebnissen. Für die Berechnung der Lösungen wurde bei der Initialisierung durch Zufallsscheduler und Greedy Scheduler durchweg die gesamte erlaubte Zeit von 60 Sekunden benötigt.

Bei Verwendung des Simulated Annealing-Verfahrens wird hingegen etwas mehr Zeit für den Scheduling-Vorgang verbraucht. Wie in Abbildung 3.4 zu erkennen, ist der Zeitbedarf umso größer, je höher die Anzahl der Maschinen und Aufträge wird. Dieser Anstieg der benötigten Zeit ist auf den zusätzlichen Aufwand zurückzuführen, den das Simulated Annealing-Verfahren erzeugt.



(a) Startlösung von Zufallsscheduler berechnet



(b) Startlösung durch Simulated Annealing ermittelt

Abbildung 3.2: Fehlermaß für den Branch And Bound-Algorithmus bei mittleren Probleminstanzen und unterschiedlichen Startlösungsalgorithmen

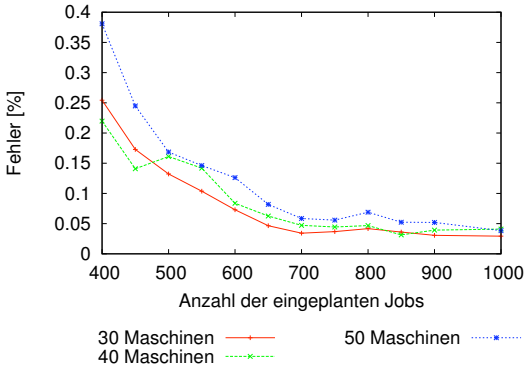


Abbildung 3.3: Fehlermaß für mittlere Probleminstanzen und dem Greedy Scheduler als Initialisierer für die Branch And Bound-Methode

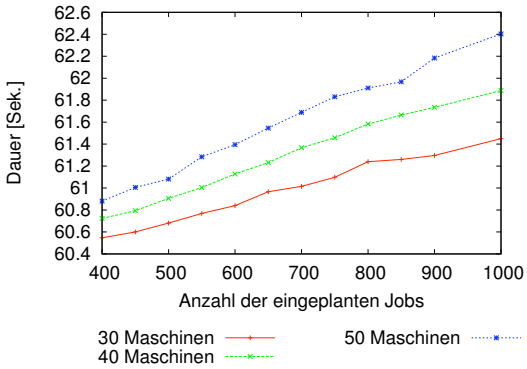
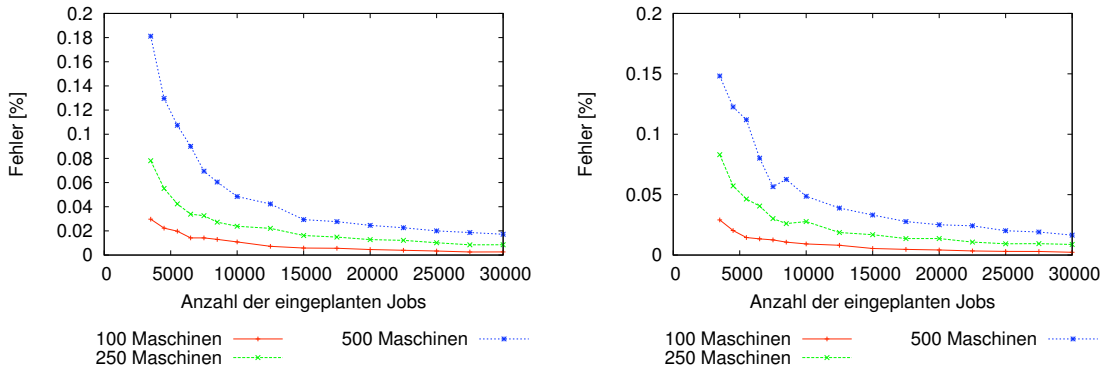


Abbildung 3.4: Rechendauer der Branch And Bound-Methode mit dem Simulated Annealing-Verfahren als Initialisierer bei mittelgroßen Probleminstanzen



(a) Startlösung von Zufallsscheduler berechnet

(b) Startlösung durch Simulated Annealing ermittelt

Abbildung 3.5: Fehlermaß für den Branch And Bound-Algorithmus bei großen Probleminstanzen und unterschiedlichen Startlösungsalgorithmen

Wendet man sich schließlich den großen Probleminstanzen zu, so lässt sich auch hier feststellen, dass die Güte der ermittelten Ergebnisse nur geringfügig von der Wahl des Initialisierungsalgorithmus abhängt. Abbildungen 3.5 und 3.6 zeigen das entsprechende Fehlermaß. Nur vereinzelte Lösungen bei Verwendung des Zufallsschedulers in Abbildung 3.5(a) sind etwas schlechter als bei Einsatz der anderen Startlösungsalgorithmen. Dies liegt darin begründet, dass die Startlösungen des Zufallsschedulers wesentlich schlechter sind, als die der anderen Verfahren. Der Branch And Bound-Algorithmus beginnt dadurch mit einer sehr hohen Obergrenze. Er benötigt daher mehr Zeit, um auf die gleichen Lösungen zu kommen, als in den Fällen, bei denen eine bessere Startlösung vorgegeben wurde. Bei der Rechendauer ergibt sich das gleiche Bild, wie bei den mittelgroßen Problemfällen: Es wird die gesamte verfügbare Rechenzeit von 60 Sekunden ausgenutzt. Einzig bei Verwendung des Simulated Annealing-Verfahrens für die Startlösung wird bedeutend mehr Zeit benötigt. Die Rechendauer für diesen Fall ist in Abbildung 3.7 dargestellt.

Abschließend sind in Abbildung 3.8 die Leistungsparameter der Branch And Bound-Methode für mittelgroße Probleminstanzen dargestellt, bei denen der Toleranzparameter α auf 2% gesetzt wurde. Für diesen Fall wurde die Startlösung mit Hilfe des Simulated Annealing-Verfahrens ermittelt. Der Einfluss von α lässt sich hier sehr gut an Fehlermaß und Rechendauer erkennen. Man sieht in Abbildung 3.8(a), dass der Fehler für die verschiedenen Probleminstanzen stark zwischen 0 und 2% schwankt, obwohl theoretisch noch bessere Werte möglich wären. Dafür ist die benötigte Rechenzeit in Abbildung 3.8(b) nur sehr gering und wird im Wesentlichen von der Rechendauer des Simulated Annealing-Schedulers beeinflusst.

Ein vergleichbares Bild ergibt sich für die anderen Problemgrößen und Initialisierungsalgorithmen. Sobald der Branch And Bound-Scheduler eine neue aktuelle Lösung mit $\epsilon < \alpha$ findet oder die gegebene Startlösung schon diese Bedingung erfüllt, bricht er sofort ab. Die entsprechenden Graphen zeigen eine ähnliche Charakteristik wie die

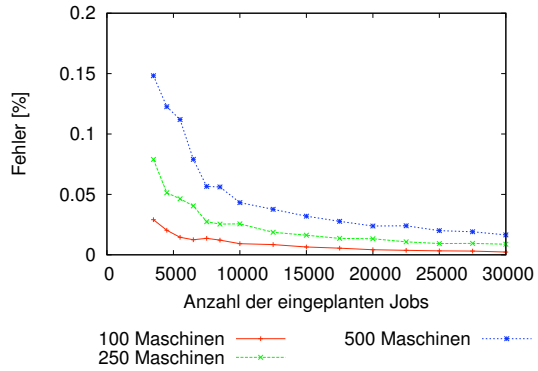


Abbildung 3.6: Fehlermaß für den Branch And Bound-Algorithmus bei großen Probleminstanzen und dem Greedy Scheduler als Initialisierungsalgorithmus

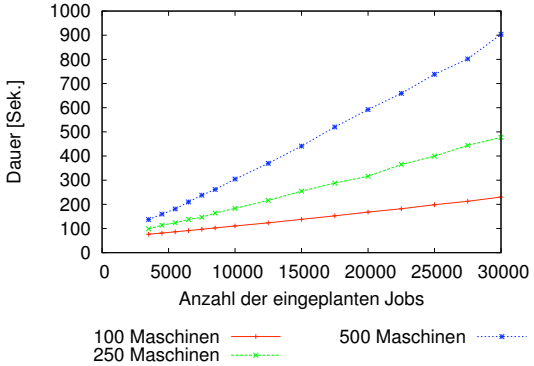
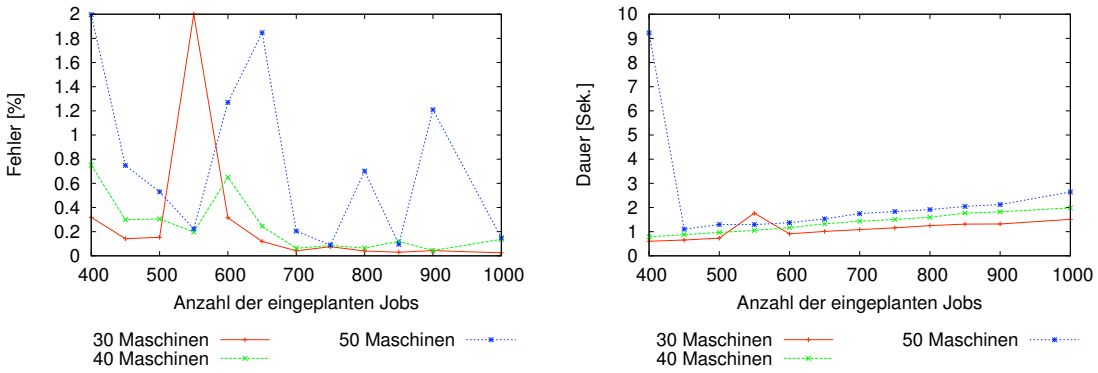


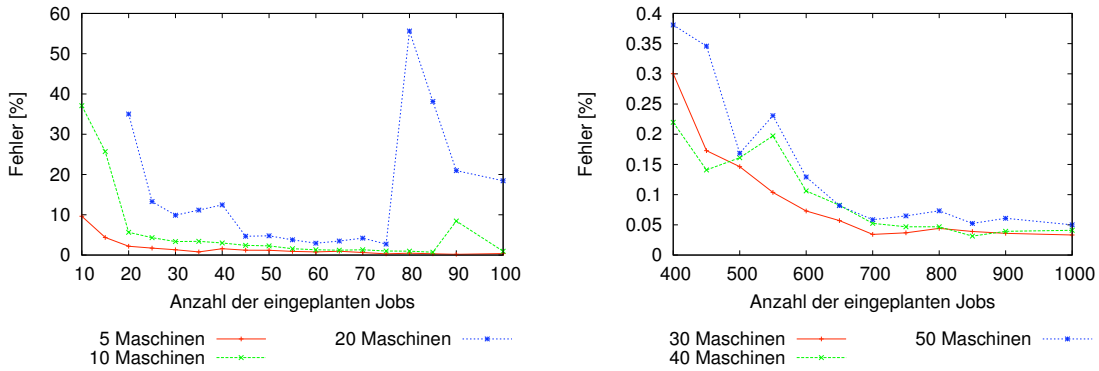
Abbildung 3.7: Rechendauer der Branch And Bound-Methode mit dem Simulated Annealing-Verfahren als Initialisierer bei großen Probleminstanzen



(a) Fehlermaß ϵ

(b) Rechendauer

Abbildung 3.8: Leistungsparameter der Branch And Bound-Methode bei mittelgroßen Probleminstanzen und mit einem Toleranzwert von 2%. Initialisierungsalgorithmus war hier das Simulated Annealing-Verfahren.



(a) Fehler für kleine Problemgrößen

(b) Fehler für mittlere Problemgrößen

Abbildung 3.9: Fehlermaß für den Greedy Scheduling-Algorithmus bei kleinen und mittelgroßen Probleminstanzen

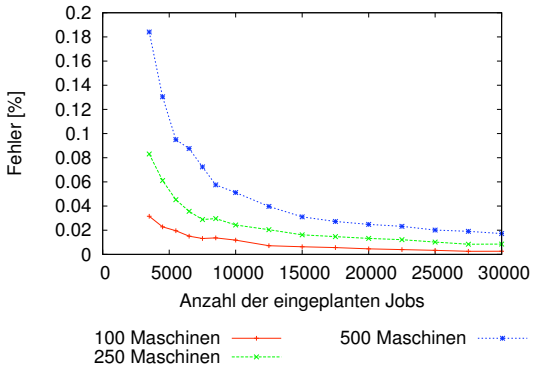


Abbildung 3.10: Fehler des Greedy Schedulers für große Problemgrößen

Graphen in Abbildung 3.8 und werden daher nicht weiter betrachtet.

3.1.3 Greedy Scheduling

Die einfache Struktur des Greedy Schedulers erlaubt eine sehr schnelle Berechnung von Maschinenbelegungsplänen. Für jeden der durchgeführten Tests wurde mit diesem Verfahren das Scheduling-Problem in weniger als einer Sekunde gelöst. Auf die entsprechenden Graphen für die Rechendauer wird daher an dieser Stelle verzichtet.

In Abbildung 3.9(a) ist das Fehlermaß für kleine Probleminstanzen abgetragen. Vergleicht man diesen Graph mit den Ergebnissen der Branch And Bound-Methode in Abbildung 3.1(a), so sieht man, dass der Greedy Scheduler bei kleinen Probleminstanzen nur selten eine optimale Lösung findet. So hat das Verfahren dies bspw. bei 20 Maschinen und 80, 85, 90 und 100 Jobs geschafft. Dass die dort gefundenen Lösungen optimal sind, lässt sich durch den Vergleich mit Abbildung 3.1(a) feststellen. Bei den meisten anderen Fällen findet der Branch And Bound-Algorithmus jedoch noch bessere Lösungen.

Die gleiche Beobachtung kann bei den mittelgroßen Probleminstanzen gemacht werden. Vergleicht man Abbildung 3.9(b) mit Abbildung 3.3 so sieht man, dass auch hier noch bessere Lösungen gefunden werden können, als durch den Greedy Scheduler geliefert werden.

Ein gänzlich anderes Ergebnis erhält man bei den großen Probleminstanzen. Man sieht, dass sich das Fehlermaß in Abbildung 3.10 nicht sonderlich von den Ergebnissen der Branch And Bound-Methode in Abbildung 3.6 unterscheidet. Der Greedy Scheduler schafft es für solche Fälle also, derart gute Lösungen zu ermitteln, dass der Branch And Bound-Scheduler diese in der gegebenen Zeit nicht mehr weiter verbessern kann.

3.1.4 Simulated Annealing

Mit dem Simulated Annealing-Verfahren hat man die Möglichkeit, durch eine entsprechende Wahl der Kernparameter für den Algorithmus einen Kompromiss zwischen der Güte einer gefundenen Lösung und der verwendeten Rechenzeit einzugehen. Kann man sich mit einer weniger guten Lösung zufrieden geben, so wählt man die Parameter dergestalt, dass das Verfahren schneller terminiert. Möchte man hingegen eine möglichst gute oder eine optimale Lösung erhalten, so setzt man die Parameter entsprechend, muss dann aber eine längere Laufzeit in Kauf nehmen.

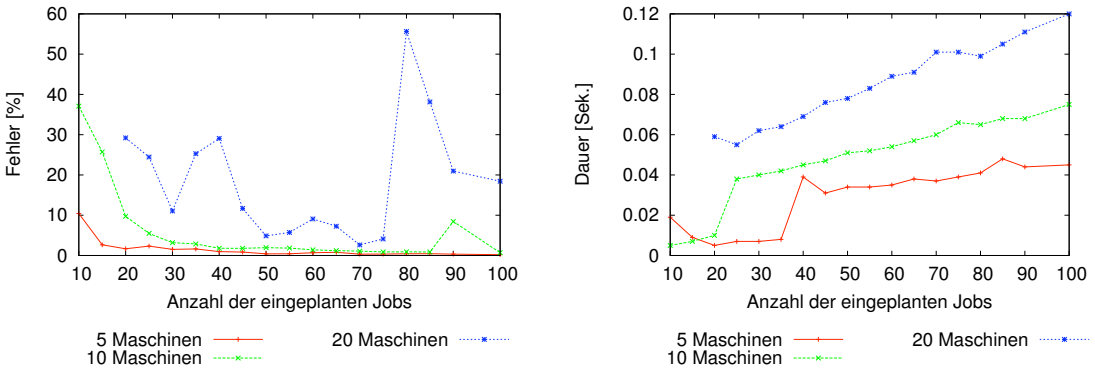
Die zwei wichtigsten Parameter zur Beeinflussung der Güte und Dauer der Simulated Annealing-Methode sind die Werte δ und L_k. Mit δ wird der Kontrollparameter c während des Abkühlens herabgesetzt. Mit L_k kann man die Anzahl der Transitionen, die zwischen jedem Herabsetzen der Temperatur c ausgeführt werden, festlegen. Wählt man für δ einen kleineren Wert, so wird c schneller verringert. Dies entspricht in der Analogie eines physikalischen Härtungsvorgangs dem rascheren Abkühlen des Feststoffs. Je kleiner δ gewählt wird, desto schneller kühlt sich das System ab. Gleichzeitig erhöht sich aber auch die Wahrscheinlichkeit, dass der Algorithmus in einem lokalen Optimum stecken bleibt.

Setzt man die Anzahl der Transitionen L_k jeder Temperaturstufe herab, so wird der Algorithmus ebenfalls schneller. Allerdings erreicht das System dann nicht mehr das notwendige thermische Gleichgewicht in jeder Temperaturstufe. Dies führt auch hier zu schlechteren Lösungen.

Bei den folgenden Graphen wurde der Temperaturmodifikator $\delta = 0,8$ gesetzt. Die Anzahl der Transitionen auf jeder Temperaturstufe berechnet sich als $L_k = mn$, wobei mit m wieder die Anzahl der Maschinen und mit n die Anzahl der zu verteilenden Aufträge gegeben ist.

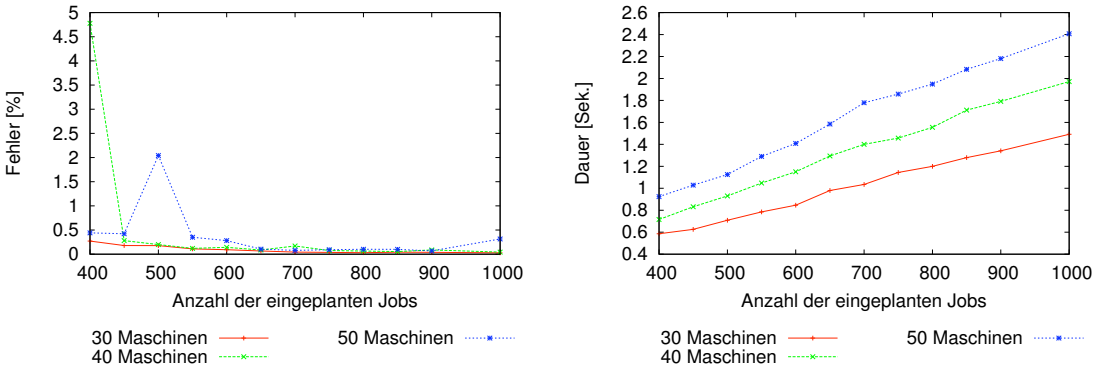
Abbildung 3.11 zeigt die Leistungsparameter des Simulated Annealing-Verfahrens für kleine Probleminstanzen. Vergleicht man das Fehlermaß in Abbildung 3.11(a) mit den Ergebnissen der Branch And Bound-Methode in Abbildung 3.1(a), so kann man erkennen, dass Simulated Annealing hier zwar grundsätzlich schneller ist als Branch And Bound, das Verfahren allerdings etwas schlechtere Ergebnisse liefert. Um hier optimale Lösungen zu erhalten, müsste man entsprechend δ, L_k oder beide Werte zusammen größer wählen.

Bei den mittelgroßen Probleminstanzen zeigt sich das gleiche Bild. Mit Rechenzeiten



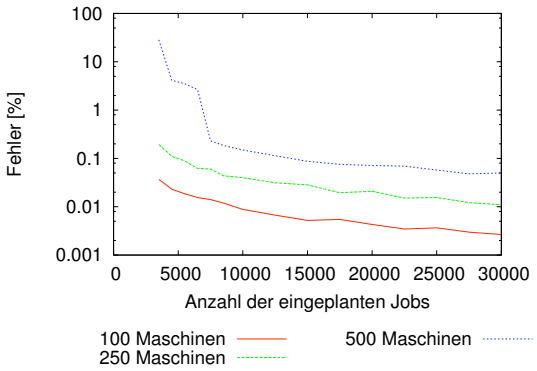
(a) Fehler ϵ (b) Laufzeit

Abbildung 3.11: Leistungsparameter des Simulated Annealing-Verfahrens bei kleinen Problemgrößen

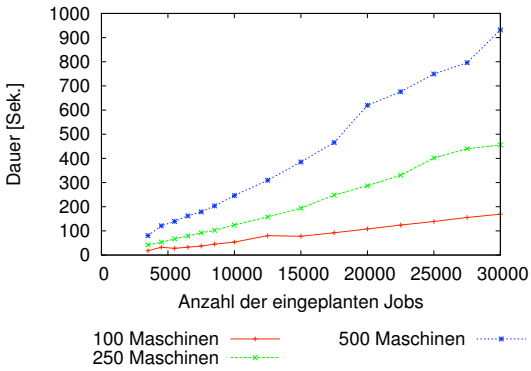


(a) Fehler ϵ (b) Laufzeit

Abbildung 3.12: Leistungsparameter des Simulated Annealing-Verfahrens bei mittleren Problemgrößen



(a) Fehler ϵ. Hier ist die y-Achse zur besseren Erkennbarkeit der weit gespreizten Werte logarithmisch skaliert.



(b) Laufzeit

Abbildung 3.13: Leistungsparameter des Simulated Annealing-Verfahrens bei großen Problemgrößen

von weniger als 3 Sekunden ist Simulated Annealing auch hier schneller als die Branch And Bound-Methode, liefert dabei aber etwas schlechtere Ergebnisse.

Möchte man große Optimierungsprobleme lösen, so muss man hier sehr lange Laufzeiten in Kauf nehmen. Abbildung 3.13(b) zeigt, dass für umfangreiche Problemstellungen Berechnungszeiten von mehreren Minuten anfallen. Die dabei gefundenen Lösungen sind sogar schlechter als die Lösungen des Greedy Schedulers. Um bessere Ergebnisse zu erzielen, müsste man auch hier wieder die entsprechenden Parameter des Verfahrens anpassen, was jedoch zu noch längeren Rechenzeiten führt. Simulated Annealing eignet sich für solch große Problemfälle also nur dann, wenn die benötigte Rechenzeit bloß eine untergeordnete Rolle spielt.

3.1.5 List Scheduling

Der List Scheduler wurde den oben besprochenen Leistungstests nicht unterzogen. Dies liegt darin begründet, dass dieser Algorithmus unter Unkenntnis des Bearbeitungsaufwandes der einzelnen Aufträge arbeitet und er somit nicht direkt mit den anderen Scheduling-Algorithmen vergleichbar ist. Gleichzeitig ist diese Eigenschaft jedoch eine Stärke des Verfahrens, da es eben auch ohne Kenntnis des Bearbeitungsaufwands anwendbar ist [16]. Da der List Scheduler nicht im Voraus einen Maschinenbelegungsplan erstellen muss, eignet sich das Verfahren auch als ein so genannter *Online*-Scheduler. Im Gegensatz zu *Offline*-Verfahren kann der List Scheduler einen Job J_i sofort zu dessen Freigabezeitpunkt r_i einer Maschine zuweisen, falls dies nicht durch weitere Nebenbedingungen verhindert wird [39]. Es muss also nicht vorher ein Schedule erstellt werden, wie dies bei den anderen Algorithmen der Fall ist.

3.1.6 Schlussfolgerungen

Betrachtet man nun die vier vorgestellten Algorithmen gemeinsam, so lassen sich aus den gezeigten Leistungsdaten die folgenden Schlussfolgerungen ziehen.

Der Branch And Bound-Algorithmus hat zwar grundsätzlich die Fähigkeit, optimale Lösungen zu finden. Er braucht dafür aber bei größeren Problemstellungen eine sehr lange Zeit. Mit einer guten, nicht optimalen Startlösung gelingt es ihm dennoch, die vorgegebene Lösung umso weiter zu verbessern, je mehr Zeit ihm zur Verfügung gestellt wird.

Wie man an Algorithmus 3 erkennen kann, benötigt die Branch And Bound-Methode eine variable Menge an Hauptspeicher. Die durchschnittliche Anzahl der in der Variable $LIST$ vermerkten Suchbaumknoten wird umso größer, je umfangreicher das zu lösende Problem hinsichtlich der Anzahl der zu verplanenden Aufträge und Maschinen ist. Ist der für die Implementierung eines Optimierungsverfahrens verfügbare Speicher stark begrenzt, kommt die Branch And Bound-Methode also nur bedingt als einzusetzender Algorithmus in Frage.

Der Greedy Scheduler ist durch die lineare Abhängigkeit der Laufzeit von seinen Eingabewerten der schnellste der hier vorgestellten Algorithmen. Obwohl er nicht die Ermittlung optimaler Lösungen garantieren kann, konnte man sehen, dass die von ihm verwendete Heuristik besonders bei größeren Problemfällen nahezu optimale Lösungen hervorbringt. Einzig bei kleinen Probleminstanzen weichen die Lösungen relativ stark vom Optimum ab.

Der Algorithmus Simulated Annealing schafft es lediglich für kleine und mittlere Problemfälle gute Lösungen in einer annehmbaren Zeit zu ermitteln. Verglichen mit den anderen Verfahren benötigt diese Methode für große Probleminstanzen eine sehr lange Rechenzeit. Gleichzeitig sind die gelieferten Lösungen schlechter als die Lösungen des Greedy Schedulers. Die Vorteile des Simulated Annealings sind allerdings der sehr geringe und konstante Speicherbedarf und die Möglichkeit zur direkten Beeinflussung von Ergebnisgüte und Rechenzeit. Der Speicherbedarf des Simulated Annealing-Schedulers wird lediglich durch ein Array festgelegt, in dem der aktuelle Maschinenbelegungsplan vorgehalten wird. Alle Transitionen, die während des Verfahrens stattfinden, werden direkt auf diesem Array durchgeführt. Der Speicherbedarf bleibt dadurch konstant. Simulated Annealing eignet sich also besonders für Anwendungsfälle, bei denen der verfügbare Speicher eine knappe Ressource darstellt.

3.2 Grundlegende Arbeitsweise der Simulationsumgebung

Die vorrangige Aufgabe, die von SimpleGrid als Simulationsumgebung erfüllt werden muss, besteht darin, unter möglichst effizienter Ausnutzung aller vorhandener Ressourcen die Gesamtdauer eines Simulationsprojektes zu minimieren. Die in den vorangegangenen Abschnitten vorgestellten Scheduling-Algorithmen sind dabei ein wichtiges Hilfsmittel für die Erreichung dieses Ziels. Bis jetzt ist die Frage offen geblieben, wie diese Algorithmen am sinnvollsten eingesetzt werden können. Es muss daher zunächst genauer untersucht werden, welche Vorgänge bei der Verteilung eines Simulationsauf-

trags innerhalb der Simulationsumgebung stattfinden und wie diese von den Scheduling-Algorithmen am besten unterstützt werden können.

In den folgenden beiden Abschnitten wird die herkömmliche Methode zur Durchführung eines Simulationsprojekts mit den Vorgängen verglichen, die während einer Simulation mit Hilfe von SimpleGrid stattfinden.

3.2.1 Herkömmliche Methode der Simulationsdurchführung

Ein Simulationsprojekt setzt sich üblicherweise aus mehreren Bestandteilen zusammen. Zum einen gehört dazu das Simulatorprogramm, das die Simulationen ausführt. Zum anderen hat man die Zusammenstellungen aller Eingabeparameter, mit welchen die einzelnen Simulationsläufe definiert werden. Solche Zusammenstellungen werden in Konfigurationsdateien vorgehalten, mit denen sich die Werte der Simulationsvariablen festlegen lassen. Weiterhin zählen dazu sämtliche Steuerdateien, mit denen die Struktur der Simulationsläufe bestimmt wird. Dies können z. B. Knotenbewegungs- und Kommunikationsmusterdateien sein.

Möchte man ohne Zuhilfenahme einer automatisierten Simulationsverteilung die Gesamtheit der durch diese Daten bestimmten Simulationsläufe ausführen, so geht man wie folgt vor. Mit Hilfe einer Skriptsteuerung fasst man die unterschiedlichen Konfigurationen jedes einzelnen Simulationslaufs zusammen. Diese werden dem Simulatorprogramm anschließend nacheinander übergeben. Der Simulator wird somit auf einem bestimmten Rechner für jede Konfiguration genau einmal gestartet und die Simulationsläufe dadurch sequenziell abgearbeitet. Alle Ausgaben, die währenddessen entstehen, werden an einer vorkonfigurierten Stelle zentral abgespeichert, an der sie dem Benutzer zur weiteren Auswertung zur Verfügung stehen.

3.2.2 Simulation mit Hilfe von SimpleGrid

Kann der Benutzer auf eine automatisierte Simulationsverteilung zurückgreifen, so ist aus seiner Sicht der Ablauf ähnlich. Auch hier fasst er wieder alle für einen Simulationsauftrag relevanten Daten zusammen und übergibt sie an eine zentrale Instanz, die sich selbstständig um die Durchführung der Simulation kümmert. Hier ist dies allerdings kein Skript, das einfach den Simulator nacheinander mit den unterschiedlichen Eingabewerten aufruft. Statt dessen besteht jetzt die Steuerungsinstanz aus der internen Logik der Simulationsumgebung. Diese versucht, eigenständig den gesamten Simulationsauftrag dergestalt zu parallelisieren, dass zum einen möglichst alle verfügbaren Ressourcen ausgenutzt werden und zum anderen der Auftrag so schnell wie möglich abgeschlossen wird. Auch in diesem Fall findet der Benutzer nach Beendigung des Simulationsprojekts alle angefallenen Ausgabedaten an einer zentralen Stelle vor, an der sie von der Simulationsumgebung zusammengetragen wurde. Von den jeweiligen internen Arbeitsabläufen bekommt der Benutzer unterdessen nichts mit.

Für die Parallelisierung eines Simulationsauftrags stehen alle Rechner zur Verfügung, die als Teilnehmer für SimpleGrid konfiguriert wurden und während der Dauer eines Simulationsauftrags bei der Simulationsumgebung angemeldet sind. Alle diese Rechner

können zur gleichen Zeit einen Teil des Rechenaufkommens übernehmen und bearbeiten. Es stellt sich nun die Frage, auf welche Art und Weise die anfallende Rechenlast auf die verfügbaren Computer verteilt werden soll.

Für diese Aufgabe muss eine Partitionierung des gesamten Rechenaufkommens gefunden werden, die es ermöglicht, jedem verfügbaren Rechner eine genau so große Teilaufgabe zuzuweisen, dass jeder beteiligte Computer etwa zum gleichen Zeitpunkt seine Bearbeitung abschließt und die Gesamtsimulationsdauer damit minimal wird. Gleichzeitig muss sich der Verwaltungsaufwand, der durch eine solche Aufteilung entsteht, in Grenzen halten.

Dazu lassen sich zwei mögliche Stellen unterscheiden, an denen man mit einer Partitionierung ansetzen könnte: zum einen auf der Ebene der Simulation selbst und zum anderen auf der Ebene der einzelnen Simulationsläufe. Setzt man für die Parallelisierung direkt in der Simulation selbst an, hieße das, dass man alle verfügbaren Rechner gleichzeitig an der Berechnung einer einzelnen Simulation arbeiten ließe. Diese Methode würde allerdings eine Reihe äußerst schwierig zu lösender Probleme hervorrufen. Zuerst müsste überlegt werden, wie man eine einzelne Simulation so in disjunkte Teilaufgaben zerlegen könnte, dass diese auf die freien Rechner verteilbar sind. Eine Möglichkeit wäre bspw., die Menge der simulierten Netzwerkknoten auf genau so viele Teilmengen zu verteilen, wie freie Rechner zur Verfügung stehen. Dann könnte man jedem Rechner eine dieser Knotenmengen zur Simulation zuweisen. Es entstünde dabei allerdings ein enormer Kommunikations- und Synchronisationsaufwand zwischen den beteiligten Rechnern. Der Ausfall einer dieser Computer würde dazu führen, dass die Simulation solange unterbrochen wird, bis die auf diesem Computer verwaltete Knotenmenge auf die restlichen Rechner verteilt wurde. Zudem müsste man an der Programmstruktur des Netzwerksimulators selbst signifikante Änderungen durchführen, damit dieser mit den neuartigen Bedingungen, die eine solche Simulationsaufteilung mit sich brächte, zurecht käme.

Diese Herangehensweise ist nicht praktikabel. Betrachtet man die Struktur eines Simulationsauftrags genauer, so lässt sich hier ein viel besserer Ansatzpunkt finden. Üblicherweise variiert und kombiniert man in einem Simulationsprojekt eine Vielzahl von unterschiedlichen Parametern. Um all diese Variationen abzuarbeiten, erhält man erfahrungsgemäß eine Anzahl von Simulationsläufen, die im drei- bis vierstelligen Bereich liegt. Für die in dieser Arbeit behandelten Netzwerksimulationen ist es charakteristisch, dass die Gesamtdauer aller Simulationsläufe zusammen sehr groß werden kann. Schon in der Einleitung wurde auf einen typischen Zeitbedarf von mehreren Tagen oder Wochen hingewiesen. Verglichen damit ist die Dauer eines einzelnen Simulationslaufs – also die Lebenszeit des damit verbundenen Simulatorprozesses – relativ gering. Hier können Simulationszeiten in der Größenordnung von einigen Sekunden bis zu mehreren Minuten beobachtet werden. Aus diesem Grund liegt es nahe, die einzelnen Simulationsläufe als Grundelement für eine Partitionierung der Gesamtrechenlast heranzuziehen.

SimpleGrid macht sich diese Eigenschaft für die Lastverteilung zunutze. Anstatt einen Simulationslauf selbst auf mehrere Rechner zu verteilen, werden ausschließlich die kompletten Simulationsläufe an einzelne Rechner zugewiesen. Für diese Aufgabe ist

ein *Scheduler* zuständig. Der Scheduler versucht mit Hilfe der in Abschnitt 2.2 vorgestellten Algorithmen, die zu bearbeitenden Simulationsläufe so auf die verfügbaren Rechner zu verteilen, dass zum einen möglichst wenig Rechner untätig bleiben, was der Verschwendung von Rechenkapazität gleichkäme. Zum anderen sollen möglichst alle beteiligten Rechner gleichzeitig ihren jeweils letzten Auftrag beenden, womit die Minimalitätsbedingung für die Gesamtrechendauer erfüllt wäre.

Wie schon zu Beginn dieser Arbeit ausgeführt ist der SimpleSim Netzwerksimulator ein Programm, das in Forschung und Lehre eingesetzt werden soll. Die fortdauernde Weiterentwicklung und Aktualisierung des Programms gehört zu den Kernaufgaben, welche bei der Arbeit mit dem Simulator erledigt werden sollen. Das Hinzufügen neuartiger Protokolle und Konzepte ist ein wesentlicher Bestandteil der Arbeit mit SimpleSim. Auf diese Tatsache muss auch die Simulationsumgebung Rücksicht nehmen. Es ist daher nicht sinnvoll, auf jedem Simulationsrechner eine eigene Kopie von SimpleSim vorzuhalten, die zur Bearbeitung der zugewiesenen Simulationsläufe verwendet wird. Dadurch wäre es für den Benutzer unmöglich, eigene Anpassung in das Programm einzubringen. Er hätte keine Möglichkeit, diese lokalen Kopien des Simulators mit seinen Änderungen zu aktualisieren. Aus diesem Grund erlaubt es SimpleGrid, eigene Versionen des Simulators für Simulationsaufträge zu verwenden. Diese Version wird dann zusammen mit den jeweiligen Konfigurationsdaten an die verfügbaren Simulationsrechner geschickt und dort für die Durchführung des Simulationslaufs verwendet. Dadurch wird es möglich, dass auch mehrere Benutzer unabhängig voneinander SimpleGrid mit den jeweils eigenen Versionen des Netzwerksimulators verwenden können, ohne sich dabei gegenseitig zu stören.

3.3 Scheduling von Simulationsaufträgen

Nach der Vorstellung der Überlegungen zur grundsätzlichen Arbeitsweise einer automatisierten Simulationsumgebung soll nun anschließend erläutert werden, wie die in der vorliegenden Arbeit vorgestellten Scheduling-Algorithmen optimal eingesetzt werden können. Auf die im nächsten Abschnitt dargelegte Klassifikation des hier zu lösenden Scheduling-Problems folgt eine Diskussion der Schwierigkeiten, die für den Einsatz der Algorithmen zu bewältigen sind. Schließlich wird beschrieben, wie die Scheduling-Algorithmen sinnbringend zum Einsatz kommen können.

3.3.1 Klassifikation des vorliegenden Problems

Mit den bisher gemachten Beobachtungen kann man das Scheduling-Problem, welches in dieser Arbeit gelöst werden soll, nach dem in Abschnitt 2.2.1 vorgestellten System von Graham et al. klassifizieren.

Für die Maschinenumgebung lässt sich im vorliegenden Fall feststellen, dass der Simulationsumgebung eine beliebige Anzahl von Rechnern zur Verfügung stehen können. Damit hat man für $\alpha_2 = \circ$. Welche spezielle Architektur die verfügbaren Computer haben, ist für die Simulationsumgebung nicht relevant. Es können prinzipiell alle Arten von Rechnersystemen an SimpleGrid teilnehmen. Für den Scheduler ist dabei alleine

die Tatsache entscheidend, dass die einzelnen verfügbaren Maschinen unterschiedlich schnell sind. Deren Geschwindigkeit hängt allein von der eingesetzten Hardware ab; sie ist also unabhängig von den Programmen, die darauf ausgeführt werden. Man hat hier also den Fall einheitlicher, paralleler Maschinen, wofür $\alpha_1 = Q$ gesetzt wird.

Wie weiter oben beschrieben wurde, besteht die Aufgabe des Schedulers darin, die einzelnen Läufe eines kompletten Simulationsauftrags auf die vorhandenen Rechner zu verteilen. Das heißt also, alle einzuplanenden Jobs sind schon zu Beginn des Schedulingvorgangs bekannt. Damit kann für jeden Auftrag J_i der Freigabezeitpunkt $r_i = 0$ und somit $\beta_3 = \circ$ gesetzt werden. Weiterhin ist es für den Auftraggeber lediglich von Bedeutung, dass der gesamte Simulationsauftrag möglichst früh abgeschlossen wird. Es ist daher nicht nötig, Fristen für die Beendigung der einzelnen Läufe festzulegen. Somit hat man für jeden Job J_i eine Frist von $d_i = \infty$ und $\beta_5 = \circ$. Gleichzeitig spielt die Reihenfolge, in der die einzelnen Simulationsläufe abgearbeitet werden, keine Rolle, da die Läufe unabhängig voneinander sind. Es lässt sich also $\beta_2 = \circ$ setzen. Eine Stapelverarbeitung von Simulationsläufen ist ebenfalls nicht notwendig und damit auch $\beta_6 = \circ$. Weiterhin soll festgelegt werden, dass ein Unterbrechen von Simulationsläufen oder die Migration eines Simulationslaufs auf einen anderen Simulationsrechner nicht möglich ist. Eine solches *Preemption* wird ausgeschlossen, womit man auch $\beta_1 = \circ$ erhält. Die Bearbeitungsdauer der einzelnen Simulationsläufe ist an keine besonderen Bedingungen geknüpft. Jeder der Läufe erfordert jeweils eine unterschiedliche Berechnungszeit. Dadurch ist auch $\beta_4 = \circ$ und somit insgesamt $\beta = \circ$.

Die im vorliegenden Fall verfolgte Zielsetzung für den Scheduler ist die Minimierung der Gesamtdauer eines Simulationsauftrags. Diese Gesamtdauer wird durch denjenigen Simulationslauf bestimmt, der als letztes beendet wird. Für die „Produktionsspanne“ M eines Simulationsauftrages gilt also $M = \max_{i=0}^{n}\{C_i\}$, wobei C_i für den Zeitpunkt steht, zu dem ein Simulationslauf J_i vollständig bearbeitet wurde. Die Produktionsspanne soll minimiert werden, weswegen man $\gamma = C_{max}$ setzt.

Mit diesen Erkenntnissen lässt sich das in dieser Arbeit behandelte Scheduling-Problem als ein Problem der Klasse

$$Q \mid\mid C_{max}$$

bestimmen.

3.3.2 Begriffsklärung

Wie in Kapitel 2 dargelegt, werden die Scheduling-Algorithmen dafür eingesetzt, Jobs mit unterschiedlichem Bearbeitungsaufwand auf Maschinen mit verschiedenen Geschwindigkeiten zu verteilen. Dafür muss für jeden Auftrag J_i der Aufwand p_i und für jede Maschine M_j deren Geschwindigkeit s_j bekannt sein. Da die einzelnen Jobs bei dem hier zu lösenden Problem aus den jeweiligen Simulationsläufen bestehen, ist es nun notwendig, herauszufinden, welchen Rechenaufwand diese auf einem Computer erzeugen. Dazu sollen an dieser Stelle zunächst einige Vorbemerkungen gemacht werden.

Ein Simulationslauf – und damit auch dessen Rechenaufwand – ist, wie an früherer Stelle schon gesehen, eindeutig durch die Menge seiner Eingabeparameter definiert. Diese werden dem Netzwerksimulator mit Hilfe einer Konfigurationsdatei übergeben, die

```
MaximumTime=300.0
MovementFileName=move-140nodes.ssm
NodeClassName=org.pi4.simplesim.test.SimpleNode
NumberOfNodes=140
StatisticsFilename=statistics.sss
Terrain.xLength=5000.000000
Terrain.yLength=5000.000000
Terrain.zLength=10.000000
TraceFileFlushAlways=false
TraceFileName=trace.sst
EventInitiatorClassName=org.pi4.simplesim.test.SimpleEventInitiator
EventInitiatorResourceFile=comm-140nodes-30links-30packets.ssa
RandomSeed=3
```

Tabelle 3.1: Beispiel einer Konfigurationsdatei für SimpleSim

sich in ihrem Aufbau nach den Vorgaben der Java-Klasse `Properties` [19] orientiert. In Tabelle 3.1 wird eine solche Datei beispielhaft dargestellt. Alle Parameter, die für die Definition eines Simulationslaufs nötig sind, finden sich in dieser Datei wieder und werden dort mit den gewünschten Werten belegt. Jeder einzelne Parameter übt dabei einen gewissen Einfluss auf den Rechenbedarf der Simulation aus. Dieser Rechenbedarf wird im Folgenden als die *Komplexität* eines Simulationslaufs bezeichnet.

Die Konfigurationsdatei für einen Simulationslauf kann auch als ein *Parametertupel* betrachtet werden. Ein solches Tupel ist dabei ein Element des kartesischen Produktes, welches aus den verschiedenen Parametertypen gebildet werden kann. Der Begriff des Parametertupels wird im Folgenden synonym für den Inhalt einer Konfigurationsdatei verwendet.

Es lässt sich nun für einige Konfigurationswerte leicht feststellen, dass sie keinerlei Einfluss auf die Komplexität einer Simulation haben. So gibt es Variablen, die sich nicht auf das Simulationsgeschehen selbst auswirken. Dazu gehören bspw. Parameter, mit denen die Namen der Ausgabedateien festgelegt werden, wie `TraceFileName`. Andere Parameter beeinflussen zwar die Simulation, wirken sich aber nur in vernachlässigbarem Maße auf die Komplexität aus. So legt bspw. der Parameter `RandomSeed` lediglich den Startwert des Zufallsgenerators fest und bestimmt darüber die Sequenz der erzeugten Zufallszahlen. Derartige Konfigurationswerte werden im Folgenden als *transparente Parameter* bezeichnet, da sie für die Komplexitätsbestimmung von Simulationsläufen nicht berücksichtigt werden müssen.

Für die weitere Untersuchung des behandelten Scheduling-Problems ist nun die Definition der *Distanz* zweier Parametertupel vonnöten. Die Distanz $\delta_P(P_i, P_j)$ zwischen zwei verschiedenen Parametertupeln P_i und P_j ist gegeben durch die Anzahl aller nichttransparenter Parameter, die sich in ihren Werten unterscheiden. Zwei Parametertupel mit einer Distanz von Null werden als *komplexitätsäquivalent* bezeichnet. Für zwei Simulationsläufe, deren Konfigurationsdaten komplexitätsäquivalent sind, wird erwartet,

Parametertupel P_1	Parametertupel P_2
`MaximumTime=300.0`	`MaximumTime=300.0`
`MovementFileName=move-40nodes.ssm`	`MovementFileName=move-250nodes.ssm`
`NumberOfNodes=40`	`NumberOfNodes=250`
`StatisticsFilename=first.sss`	`StatisticsFilename=second.sss`
`TraceFileName=first.sst`	`TraceFileName=second.sst`
`RandomSeed=1`	`RandomSeed=2`

Tabelle 3.2: Beispiel für die Distanz zweier Parametertupel mit $\delta_P(P_1, P_2) = 2$

dass sie bei ihrer Ausführung ungefähr denselben Rechenaufwand erzeugen. Gilt für zwei beliebige Parametertupel P_a und P_b dagegen $\delta_P(P_a, P_b) > 0$, so bedeutet dies, dass die beiden Tupel eine unterschiedliche Komplexität aufweisen. Wie groß der Unterschied dann zwischen den beiden Komplexitäten ist, wird nicht näher spezifiziert.

Tabelle 3.2 zeigt beispielhaft zwei Parametertupel P_1 und P_2 mit einer Distanz von zwei. Hier sind die Parameter `StatisticsFilename`, `TraceFileName` und `RandomSeed` transparent, während dies auf die restlichen Parameter nicht zutrifft. Da sich die beiden Tupel bei den nichttransparenten Parametern `MovementFileName` und `NumberOfNodes` in ihrer Konfiguration unterscheiden, gilt $\delta_P(P_1, P_2) = 2$.

Die Unterscheidung von komplexitätsäquivalenten und unterschiedlich komplexen Simulationsläufen wird an späterer Stelle für die Durchführung der Auftragsverteilung benötigt.

3.3.3 Schwierigkeiten beim Einplanen von Simulationsjobs auf freie Rechner

Bevor der Scheduler für SimpleGrid seine Aufgaben zur Verteilung aller abzuarbeitenden Simulationsläufe wahrnehmen kann, müssen zunächst eine Reihe von Problemen gelöst werden. Welche Schwierigkeiten bei der Simulationsverteilung auftreten und wie diese umgangen werden können, soll in den nächsten Abschnitten diskutiert werden.

Für die Anwendung der Scheduling-Algorithmen müssen neben der Komplexität der zu verteilenden Jobs auch die Rechengeschwindigkeiten s_j der zur Verfügung stehenden Maschinen bekannt sein. Diese Information ist für die Algorithmen wichtig, da sie damit die voraussichtliche Laufzeit eines Simulationslaufs auf einem bestimmten Rechner abschätzen können. Mit Kenntnis der Maschinengeschwindigkeiten ist dem Scheduler eine solche Schätzung möglich. Es stellt sich hier allerdings die Frage, wie die Geschwindigkeit einer Maschine ermittelt und als Kennzahl ausgedrückt werden kann.

Als Teilnehmer für das Simulationsgrid kommen prinzipiell alle Rechner in Frage, die in der Lage sind, Programme auszuführen, welche für die Version 1.5 des Java-Interpreters geschrieben worden sind. Einschränkungen hinsichtlich der Architektur oder des verwendeten Betriebssystems – sofern es sich um eines handelt, für das eine entsprechende Java-Version existiert – gibt es außer der genannten keine. Es ist daher möglich, dass eine Vielzahl von Computern mit den unterschiedlichsten Systemkonfigurationen als Teilnehmer an den Aufgaben der Simulationsumgebung mitwirkt.

Für die unterschiedlichen Rechnerkonfigurationen muss nun eine Kennzahl gefunden

werden, mit der eine möglichst akkurate Abschätzung der Simulationslaufzeiten möglich ist. Ein denkbarer Ansatz wäre es bspw., die verschiedenen Computerarchitekturen nach bestimmten Gesichtspunkten, wie z. B. Prozessortyp und -geschwindigkeit, Anzahl der verfügbaren Prozessoren und Prozessorkerne oder Größe des vorhandenen Hauptspeichers, zu klassifizieren und nach einem bestimmten Schema zu bewerten. Diese Herangehensweise ist aber aufgrund einer Reihe von Gründen nicht praktikabel. So hängt die tatsächliche Ausführungsgeschwindigkeit einer Simulation auf einem Computer von einer Vielzahl von Faktoren ab. Diese umfassen bspw. die Art und Version des verwendeten Betriebssystems, die Version der verwendeten Java-Umgebung, die Spezifikationen des Prozessors, wie Größe der Prozessor-Caches oder Breite der Rechenregister, oder die aktuellen Systemeinstellungen, wie Speicher- und Bustaktraten. Das Heranziehen einiger weniger Einflussfaktoren, wie die oben beispielhaft genannten, wird nicht genügen, um eine verlässliche Aussage über die Leistungsfähigkeit eines Systems machen zu können. Eine vollständige Berücksichtigung aller möglicher Einflussfaktoren ist aufgrund ihrer großen Anzahl nicht möglich.

Ein solches System wäre zudem nicht in der Lage, zukünftige Entwicklungen zu berücksichtigen. Bewertet man die Leistungsparameter der unterschiedlichen Systeme zu einem bestimmten Zeitpunkt, müsste man für später neu erscheinende Hardwarekonfigurationen und Softwareversionen diese neu klassifizieren. Der Aufwand würde in keinem Verhältnis zum erreichten Nutzen stehen.

Weiterhin problematisch ist bei diesem System, dass es nicht die jeweilige aktuelle Lastsituation eines verfügbaren Rechners berücksichtigen kann. Wird ein Computer nicht nur für die Berechnung von Netzwerksimulationen eingesetzt, sondern gleichzeitig auch für andere Aufgaben, so muss eine Geschwindigkeitsbewertung auf diesen Umstand Rücksicht nehmen. Die Geschwindigkeit eines Rechners, der häufig unter hoher Rechenlast steht, muss niedriger bewertet werden, als die eines baugleichen Rechners, der ausschließlich für Netzwerksimulationen bereit steht und keine anderen Aufgaben nebenher bearbeiten muss.

Wie dieses Problem von SimpleGrid behandelt wird, ist Gegenstand des nächsten Abschnitts.

Eine weitere Schwierigkeit, die für die Arbeit des Schedulers gelöst werden muss, ist die Komplexitätsbewertung der einzelnen Simulationsläufe. Wie schon im vorangegangenen Abschnitt erwähnt, beeinflussen die nichttransparenten Elemente eines Parametertupels die Rechenkomplexität einer Simulation. Hier stellt sich allerdings genau wie bei der Bewertung der Rechnerkapazitäten das Problem, dass sich keine feste und einmalige Klassifikation und Bewertung für die einzelnen Parametertypen festlegen lässt. Es ist nicht von vornherein klar, wie sich die Belegungen der einzelnen Parameter auf die Simulationsdauer auswirken werden. Zudem können die verschiedenen Parameter in eine nicht vorhersagbare Wechselwirkung treten und ein Abschätzen der Komplexität unmöglich machen.

Auch bei den Parametertypen, die für die Konfiguration eines Simulationslaufs zur Verfügung stehen, können sich zukünftige Änderungen ergeben. Zum einen kann sich die Auswirkung bestimmter Werte auf die Komplexität durch spätere Erweiterungen

oder Fehlerkorrekturen des Simulatorprogrammcodes ändern. Zum anderen können aufgrund der Erweiterbarkeit des Netzwerksimulators später noch neuartige Parametertypen eingeführt werden. Dies kann bspw. bei der Implementierung eines neuentwickelten Netzwerkprotokolls geschehen.

Welche Maßnahmen in der Simulationsumgebung ergriffen werden, um dennoch Aussagen über die Komplexität eines Simulationslaufs treffen zu können, wird in Abschnitt 3.3.5 erläutert.

Bei der Verteilung von Simulationsläufen auf verfügbare Rechner spielt nicht nur die Geschwindigkeit der verwendeten Rechner eine Rolle. Es muss zusätzlich auch auf vorhandene Speicherrestriktionen und die Eigenarten der Speicherverwaltungsmechanismen auf den Simulationsrechnern geachtet werden. Jede Simulation mit dem Netzwerksimulator geht mit einem bestimmten Hauptspeicherverbrauch einher, der ebenso wie der entstehende Rechenaufwand direkt von den Konfigurationsparametern einer Simulation abhängig ist. Die für die Ausführung der Simulationen zur Verfügung stehenden Rechner können über gänzlich unterschiedliche Hauptspeichermengen verfügen. Dabei ist es möglich, dass manche Simulationsläufe einen derartig großen Hauptspeicherbedarf haben, dass sie auf bestimmten Rechnern, die diesen Bedarf nicht decken können, nicht ausgeführt werden können. Es muss also ein Mechanismus vorhanden sein, der es erlaubt, den Speicherverbrauch eines Simulationslaufs zu ermitteln und während des Schedulings entsprechend darauf Rücksicht zu nehmen. Der Scheduler sollte es vermeiden, dass ein Simulationslauf einem Rechner zugewiesen wird, der für diese Simulation nicht genügend Speicher bereithält.

Eine weitere Besonderheit bei der Hauptspeicherverwaltung der Simulationsrechner muss im Rahmen der Simulationsverteilung berücksichtigt werden. Die heute eingesetzten, modernen Betriebssysteme arbeiten mit so genanntem *virtuellen* Speicher [26, 40]. Dabei ist der von Prozessen benutzbare Speicherbereich größer als der physikalisch vorhandene Hauptspeicher. Prozesse und Daten, auf die z. B. länger nicht zugegriffen wurde, können aus dem Hauptspeicher verdrängt werden. Das Betriebssystem lagert sie dann in einen speziellen *Auslagerungsspeicher*, der sich auf dem Hintergrundspeicher des Systems befindet, aus. Damit lässt sich für andere Prozesse Platz im Hauptspeicher schaffen. Diese Auslagerungsvorgänge werden mit dem Begriff *Memory Paging* bezeichnet. Im Gegensatz zum eigentlichen Hauptspeicher eines Systems ist der Zugriff auf den Hintergrundspeicher allerdings um einige Größenordnungen langsamer. Aus diesem Grund werden Programme massiv in ihrer Ausführungsgeschwindigkeit gebremst, wenn gleichzeitig solche Memory Paging-Ereignisse stattfinden. Werden Speicherseiten häufig auf den langsamen Hintergrundspeicher ausgelagert und wieder von dort zurückgeholt, kann der Prozessor währenddessen nicht für den entsprechenden Prozess weiterrechnen. Er muss dann erst auf die Beendigung der jeweiligen Ein- und Ausgabeoperationen warten [42]. Auf diesen Umstand muss in der Simulationsumgebung gesondert eingegangen werden. Es ist wichtig, Situationen, in denen dieses so genannte *Swapping* auftritt, zu erkennen und zu behandeln. Dies ist Gegenstand von Abschnitt 3.3.6.

3.3.4 Kapazitätsmessung der verfügbaren Rechner

Für die Klassifikation der zum Einsatz kommenden Computersysteme soll nun ein Verfahren vorgestellt werden, das völlig ohne die vorherige Bewertung existierender Computerarchitekturen auskommt. Das Verfahren ermöglicht die Ermittlung einer Kennzahl, mit der sich die Rechenkapazität eines Computers repräsentieren lässt und die den Vergleich von Rechnern untereinander erlaubt. Gleichzeitig berücksichtigt die Methode in einem begrenzten Maße die jeweilige Lastsituation eines Computers.

Für die Ermittlung der Leistungsfähigkeit eines verfügbaren Simulationsrechners wird ein so genannter *Benchmark* auf dem zu bewertenden Rechner durchgeführt. Ein Benchmark wird durch einen einmalig festgelegten Simulationslauf dargestellt. Dieser wird durch eine beliebig wählbare Konfiguration des Netzwerksimulators bestimmt. Welche Werte die einzelnen Parameter des Simulators dabei erhalten, ist für das Benchmarking irrelevant. Wichtig ist lediglich, dass eine einmalig gewählte Benchmarksimulation nicht mehr geändert wird.

Für die Durchführung eines solchen Benchmarks wird auf dem zu klassifizierenden Rechner die für diesen Zweck festgelegte Simulation durchgeführt. Gleichzeitig wird die währenddessen in Anspruch genommene Zeit gemessen. Die benötigte Zeit dient nun als Kennzahl κ_j für die Leistungsfähigkeit eines Simulationsrechners M_j. Da diese Kennzahl nur in Zusammenhang mit der entsprechenden Benchmarksimulation interpretierbar ist, wird hier auch deutlich, warum eine einmal gewählte Simulationskonfiguration für die Kapazitätsmessung nicht mehr geändert werden sollte. Der willkürliche Wechsel der Benchmarkkonfiguration würde alle bisher gemessenen Kapazitätskennzahlen ungültig machen.

An dieser Vorgehensweise erkennt man auch, dass eine auf Benchmarks aufbauende Kapazitätsmessung keine statischen Kennzahlen für die Leistungsfähigkeit eines Rechnersystems liefert. So kann man feststellen, dass man zwei sich um einen kleinen Betrag unterscheidende Werte für κ_j erhält, wenn man einen Benchmarksimulationslauf zweimal hintereinander auf demselben Rechner ausführt. Diese Tatsache liegt darin begründet, dass die Laufzeit eines Benchmarks nicht nur von der verwendeten Computerhardware abhängt, sondern zusätzlich auch von weiteren Prozessen, die auf dem jeweiligen Rechner gleichzeitig ausgeführt werden. Da diese Prozesse selbst einen Teil der Rechenleistung beanspruchen, steht für den Benchmarklauf entsprechend weniger Kapazität zur Verfügung.

Eine Berechnung der Leistungskennzahlen mit dieser Methode ermöglicht also lediglich eine gute Tendenzaussage. Gleichzeitig hat sie aber den Vorteil auch bei unbekannten Rechnerarchitekturen und Computersystemen flexibel anwendbar zu sein, ohne auf genauere Kenntnisse über die klassifizierten Systeme angewiesen sein zu müssen.

Wie schon weiter oben beschrieben, möchte man mit Hilfe der Leistungskennzahlen κ_j eine Vorhersage darüber treffen können, wie lange ein bestimmter Simulationslauf auf einem Rechner dauern wird. Neben der Rechenkapazität des betreffenden Rechners hat zusätzlich auch die aktuelle Lastsituation auf diesem System einen Einfluss auf die Rechendauer. Sind auf dem Computer nebenher noch weitere Prozesse aktiv, die ihren Teil der Rechenkapazität beanspruchen, so dauert eine gleichzeitig durchgeführte

Simulation naturgemäß länger, als wenn die gesamte Kapazität für die Simulation zur Verfügung stünde.

Auf diesen Umstand muss auch bei der Kapazitätsermittlung Rücksicht genommen werden. Hierbei ergibt sich allerdings das Problem, dass eine Vorhersage der zukünftigen Rechenlast eines Computers nicht möglich ist, da im Allgemeinen die Auslastung eines Rechners nach einem zufälligen Muster schwankt. Die Kenntnis der jeweiligen zu erwartenden Lastsituationen wäre jedoch für den Scheduler von Vorteil.

Eine Möglichkeit, dieser Problematik zu begegnen, ist das periodische Durchführen von Benchmarkläufen. Anstatt einen Simulationsrechner nur einmalig zu vermessen, wird der Benchmark mit einem bestimmten Zeitabstand regelmäßig wiederholt. Die Kapazitätskennzahl κ_j ergibt sich dann aus dem Mittelwert aller dabei gemessenen Rechenzeiten. Auf diese Weise erhebt man eine Stichprobe aus dem Lastprofil eines Computers, deren Mittelwert sich der durchschnittlichen Rechenlast des Systems annähert. So erhält man einen Schätzer für die Rechenkapazität eines Computersystems, der nicht nur die Leistungsfähigkeit der zum Einsatz kommenden Hardware einbezieht, sondern auch die mittlere Auslastung des Systems berücksichtigt.

3.3.5 Komplexitätsmessung der Simulationsläufe

Die zweite für einen Scheduling-Algorithmus wichtige Information ist der für einen Job J_i anfallende Rechenaufwand p_i. Dieser wird direkt von der Komplexität eines Simulationslaufs und damit durch den entsprechenden Parametertupel beeinflusst. Bevor ein optimaler Maschinenbelegungsplan erstellt werden kann, muss dieser Komplexitätswert zunächst bekannt sein. Wie zuvor schon erläutert, kann der Wert nicht a priori durch Berechnung anhand der Simulationsparameter ermittelt werden.

Auch hier ist wieder die direkte Messung einer vorherigen Berechnung vorzuziehen. Führt man einen Simulationslauf, dessen Komplexität noch unbekannt ist, auf einem Simulationsrechner aus und misst dabei die verstrichene Zeit, so kennt man danach den Rechenaufwand, den diese Simulation dort erzeugt hat. Natürlich ist die Kenntnis dieses Aufwands für die Einplanung des Simulationslaufs nicht mehr von Nutzen, nachdem man die Simulation schon durchgeführt hat. Mit Hilfe des in Abschnitt 3.3.2 eingeführten Konzeptes komplexitätsäquivalenter Parametertupel lässt sich jedoch dennoch Nutzen aus einer solchen Information ziehen.

Hat man eine Menge komplexitätsäquivalenter Simulationsläufe, so bedeutet diese Äquivalenzbeziehung, dass jeder dieser Simulationen etwa den gleichen Rechenaufwand erzeugt. Somit genügt es, den Rechenbedarf eines beliebigen Elementes dieser Menge zu kennen, um damit auf die Komplexität der anderen äquivalenten Simulationsläufe zu schließen.

Dadurch hat man eine Möglichkeit, die Komplexitätswerte der einzelnen zu verteilenden Jobs zu ermitteln und diese Information für das Scheduling einzusetzen. Man geht dafür wir folgt vor. Zuerst führt man eine Simulation J_i mit unbekannter Komplexität auf einem freien Rechner M_j aus und misst dabei die verstrichene Zeit T_{ij}. Der Komplexitätswert ξ_i ergibt sich nun als Vielfaches der Kapazitätskennzahl κ_j der verwendeten

Maschine:

$$\xi_i = \frac{T_{ij}}{\kappa_j}$$

Unterstellt man nun eine lineare Beziehung zwischen den verschiedenen Kapazitätskennzahlen κ_j und den Komplexitätswerten ξ_i, so lässt sich dieses Konzept für die Schätzung der voraussichtlich benötigten Rechenzeit eines Simulationslaufs auf einem beliebigen Rechner anwenden. Die lineare Beziehung besagt hierbei, dass z. B. eine Simulation auf einem Computer M_a doppelt so viel Zeit benötigt wie auf einem anderen Computer M_b, wenn auch der Benchmark auf M_a die doppelte Zeit benötigt hat, wie auf M_b. Für diesen Fall gilt also $\kappa_a = 2 \cdot \kappa_b$.

Eine Schätzung $\hat{T}_{kl}$ für die Zeit, die ein Simulationslauf J_k auf einem beliebigen Rechner M_l voraussichtlich benötigen wird, lässt sich also nach der folgenden Vorschrift berechnen:

$$\hat{T}_{kl} = \xi_k \cdot \kappa_l \tag{3.3}$$

3.3.6 Behandlung des Speicherbedarfs und Erkennung von exzessivem Memory Paging

Die Menge des vorhandenen Hauptspeichers eines Simulationsrechners hat nicht nur einen entscheidenden Einfluss auf die Durchführbarkeit von Netzwerksimulationen. Auch der Scheduling-Prozess wird davon beeinflusst. Dies soll im Folgenden etwas genauer beleuchtet werden.

Führt man einen Simulationslauf mit einem hohen Speicherbedarf auf einem Computer durch, der diese Anforderung nicht erfüllen kann, so wird der Netzwerksimulator zu einem nicht näher bestimmbaren Zeitpunkt während der Simulation mit einer Fehlermeldung abbrechen[1]. Die Simulation muss dann auf einer anderen Maschine, die den entsprechenden Speicherbedarf decken kann, wiederholt werden. Hierbei geht jedoch wertvolle Zeit verloren. Es muss daher dafür gesorgt werden, dass solche Fälle möglichst vermieden werden.

Dies kann man erreichen, indem man das Konzept der Komplexität einer Simulation um ein Element erweitert. So kann man davon ausgehen, dass zwei komplexitätsäquivalente Simulationsläufe nicht nur ein ähnliches Verhalten hinsichtlich ihres Rechenaufwands vorweisen, sondern zusätzlich auch tendenziell einen ähnlich hohen Speicherbedarf haben. Führt man nun ein Element aus einer Menge komplexitätsäquivalenter Simulationsläufe auf einem Simulationsrechner aus und misst den dabei verbrauchten Hauptspeicher, so lässt sich dieser Wert als repräsentativ für den Speicherbedarf der restlichen äquivalenten Simulationsläufe interpretieren.

Zusammen mit der Information darüber, welche Mengen an Hauptspeicher die einzelnen Simulationsrechner zur Verfügung stellen, kann der Scheduler anhand des gemessenen Speicherbedarfs der komplexitätsäquivalenten Simulationsläufe entscheiden, welche Rechner für die einzelnen Simulationen in Frage kommen, und welche bei der Einplanung außen vor gelassen werden müssen. Damit kann verhindert werden, dass

[1] Im Kontext einer Java-Anwendung wird von der Java Virtual Machine in einem solchen Fall eine Ausnahme vom Typ `OutOfMemoryError` geworfen [19, 38].

Simulationsläufe mit einem hohen Speicherbedarf an Rechner zugewiesen werden, die diesen Bedarf nicht decken können. Grundsätzlich ist es dabei möglich, dass eine Simulation dennoch mehr Hauptspeicher benötigt als durch diese Methode vorhergesagt wird. Es kann also geschehen, dass eine Simulation wider Erwarten aufgrund einer zu hohen Speicheranforderung abgebrochen werden muss. Die beschriebene Methode hilft jedoch dabei, diese Situationen in ihrer Häufigkeit zu beschränken.

Ein weiterer wichtiger Faktor, der die Funktionsweise der Simulationsumgebung beeinflussen kann, ist das Auftreten des weiter oben beschriebenen Ein- und Auslagerns (Swapping) von Speicherseiten zwischen physikalischem und virtuellem Speicher. Tritt eine solche Situation ein und während der Durchführung einer Simulation wird exzessives Memory Paging betrieben, so wird die Ausführungsgeschwindigkeit der Simulation extrem herabgesetzt. Die Ursache für dieses Verhalten muss dabei nicht zwingend beim Speicherverbrauch des Netzwerksimulators selbst liegen. Es ist auch sehr gut möglich, dass andere Prozesse, die auf dem Simulationsrechner zeitgleich ausgeführt werden, einen derart hohen Speicherbedarf haben, dass das System übermäßig häufig auf Auslagerungsspeicher zurückgreifen muss. Es lässt sich daher nicht antizipieren, wann dieser Effekt auftreten wird.

Auch wenn sich nicht vorhersagen lässt, zu welchem Zeitpunkt auf einem Simulationsrechner exzessiv auf Auslagerungsspeicher zugegriffen werden muss, so ist es doch von entscheidender Wichtigkeit, diese Situation sobald sie auftritt zu erkennen. Da die Ausführungsgeschwindigkeit eines Programms, dessen Daten auf den Hintergrundspeicher ausgelagert werden müssen, massiv herabgesetzt wird, würde dadurch auch die Komplexitätsmessung eines Simulationslaufs stark verzerrt werden. Würde man eine Simulation J_i auf einem Rechner durchführen, auf dem gerade starke Swapping-Aktivitäten stattfinden, so erhielte die dabei gemessene Komplexität ξ_i einen zu großen Wert. Dies würde gleichzeitig bedeuten, dass die Rechendauer aller zu J_i äquivalenter Simulationen stark überschätzt werden würde.

Aus diesem Grund muss während jeder durchgeführten Simulation der Zugriff auf den Auslagerungsspeicher überwacht werden. Wird häufiges Memory Paging festgestellt, muss die Simulation sofort abgebrochen und auf einem anderen Simulationsrechner wiederholt werden. Auf diese Weise lässt sich die Erhebung verzerrter Komplexitätswerte vermeiden.

3.4 Einsatz der Scheduling-Algorithmen für die Simulationsverteilung

In Abschnitt 3.1 wurde untersucht, welche Fähigkeiten und Eigenschaften die einzelnen Scheduling-Algorithmen zu Eigen haben. Daraus soll nun eine Strategie abgeleitet werden, wie diese im Rahmen der Simulationsverteilung sinnvoll eingesetzt werden können.

Ein Simulationsprojekt beginnt üblicherweise damit, dass ein Benutzer alle für seinen Simulationsauftrag benötigten Daten der Simulationsumgebung überreicht. Der Scheduler hat dann zur Aufgabe, diese mit Hilfe der oben vorgestellten Scheduling-Algorithmen auf freie Rechner zu verteilen. Dabei stellt sich ihm das Problem, dass er zunächst die

Komplexitätswerte der einzelnen Simulationsläufe ermitteln muss. Erst mit dieser Information kann er einen der Algorithmen Branch And Bound, Simulated Annealing oder Greedy Scheduling anwenden.

Zieht man das in Abschnitt 3.3.2 vorgestellte Konzept der Komplexitätsäquivalenz von Simulationsläufen heran, so bietet sich die folgende Strategie für die Simulationsverteilung an. Zu Beginn seiner Arbeit berechnet der Scheduler die Mengen der komplexitätsäquivalenten Simulationsläufe. Formal bedeutet dies, dass die Menge aller gegebener Parametertupel $\mathcal{P} = \{P_i \mid i = 0, \ldots, n\}$ in eine Menge disjunkter Teilmengen

$$\bar{\mathcal{P}} = \{\bar{\mathcal{P}}_k \mid k = 0, \ldots, n\}$$

mit

$$\bar{\mathcal{P}}_k = \{P_l \mid l \in \{0, \ldots, n\} \ \wedge \ \delta_P(P_k, P_l) = 0 \ \wedge \ P_k, P_l \in \mathcal{P}\}$$

aufgeteilt wird. Hierbei steht δ_P für die in Abschnitt 3.3.2 definierte Distanzfunktion. Es gilt weiterhin $\bigcup_{k=0}^{n} \bar{\mathcal{P}}_k = \mathcal{P}$.

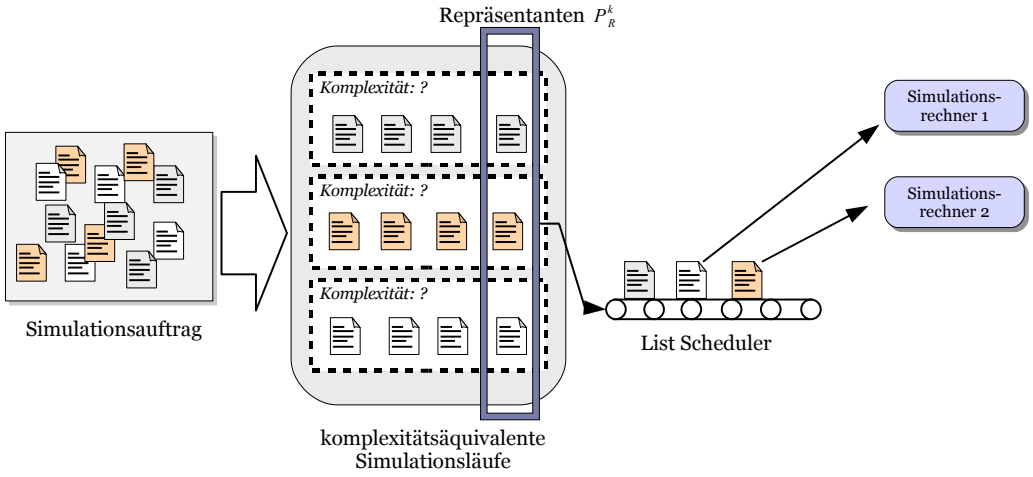
Der Scheduler wählt als nächstes aus jeder dieser Teilmengen ein beliebiges $P_R^k \in \bar{\mathcal{P}}_k$ als Stellvertreter für $\bar{\mathcal{P}}_k$ aus und verteilt diese mit Hilfe des List Schedulers auf die jeweils schnellsten der freien Rechner. Der List Scheduler muss hier deshalb zum Einsatz kommen, weil für die P_R^k noch keine Komplexitätswerte bekannt sind.

Sobald ein Simulationsrechner die Bearbeitung von einem P_R^k abgeschlossen hat, ist mit der gemessenen Simulationsdauer von P_R^k dessen Komplexitätswert – und damit die Komplexitätswerte aller in $\bar{\mathcal{P}}_k$ enthaltenen Elemente – bekannt. Damit kann nun einer der anderen Scheduling-Verfahren zum Einsatz kommen. D. h., es ist jetzt möglich, für die restlichen der in den $\bar{\mathcal{P}}_k$ verbliebenen Simulationsläufe einen optimalen Maschinenbelegungsplan zu erstellen. Abbildung 3.14 stellt diesen Vorgang symbolisch dar.

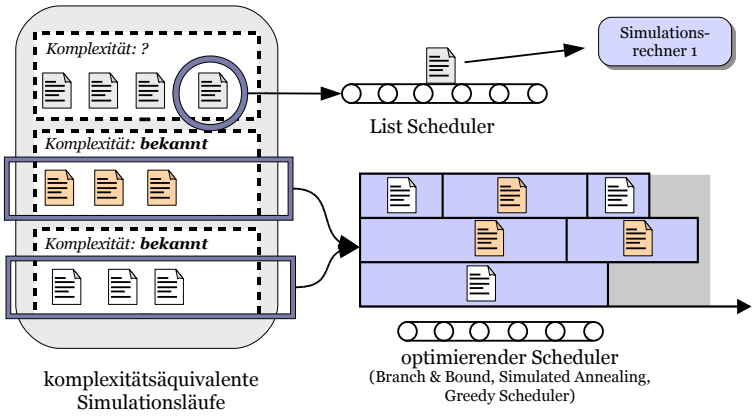
Hierfür stellt sich die Frage, welche der Scheduling-Algorithmen zur Anwendung kommen sollen. Mit Hilfe der in Abschnitt 3.1.6 gemachten Schlussfolgerungen lässt sich eine Strategie ableiten, bei der die Scheduling-Verfahren in genau den Fällen zum Einsatz kommen, für die sie die jeweils günstigsten Eigenschaften aufweisen.

Wie man in Abschnitt 3.1.3 sehen konnte, liefert der Greedy Scheduler für sehr große Probleminstanzen in sehr kurzer Zeit Lösungen, die nahe am theoretischen Optimum liegen. Das Simulated Annealing-Verfahren kommt bei diesen Fällen aufgrund seiner sehr langen Laufzeit nicht in Frage. Die Branch And Bound-Methode kann die vom Greedy Scheduler vorgegebene Lösung in der gegebenen Zeit nicht signifikant verbessern (vgl. hierfür Abbildung 3.6 und 3.10). Es liegt daher nahe, für eine sehr große Anzahl von zu verteilenden Aufträgen ausschließlich auf den Greedy Scheduler zurückzugreifen.

Hat man hingegen nur eine moderate Menge an einzuplanenden Jobs, treten die Nachteile von Simulated Annealing und Branch And Bound in den Hintergrund, während ihre Vorteile überwiegen. So lassen sich hier die beiden Verfahren sinnbringend einsetzen, indem die Branch And Bound-Methode die vom Simulated Annealing-Scheduler erhaltene Startlösung noch weiter verbessert.



(a) Zuerst werden für einen Simulationsauftrag die Mengen komplexitätsäquivalenter Simulationsläufe gebildet. Der List Scheduler verteilt die Repräsentanten dieser Mengen dann auf freie Rechner.



(b) Sind die ersten Komplexitätswerte ermittelt worden, so kann ein optimierendes Scheduling-Verfahren für die entsprechenden Jobs Maschinenbelegungspläne berechnen.

Abbildung 3.14: Vorgang bei der Verteilung eines Simulationsauftrags. Die einzelnen Parametertupel werden durch die Textdateisymbole repräsentiert.

Kapitel 4

Implementierung

In den vorangegangenen Kapiteln wurden die Grundlagen für die Konzepte gelegt, die für die Entwicklung einer automatisierten Simulationsverteilung von Bedeutung sind. In diesem Kapitel soll nun dargestellt werden, wie diese Konzepte angewandt wurden, um die gegebene Problemstellung zu lösen.

4.1 Anforderungen an die Simulationsumgebung

Zunächst soll einleitend eine Erörterung der Anforderungen erarbeitet werden, die für diese Arbeit an eine automatisierte Simulationsverteilung für SimpleSim gestellt werden. Wie schon im Abschnitt 2.1.3 dargelegt, wurde die Entwicklung von SimpleSim durch eine Reihe von Verbesserungswünschen gegenüber herkömmlichen Netzwerksimulatoren angeregt. Darunter findet sich als der wichtigste Maßstab, der an das Programm gelegt wird, der Grad der Einfachheit bezüglich der Bedienung und Programmierung des Simulators. Eine Simulationsumgebung für SimpleSim muss sich nun natürlich eng an diesen Vorgaben orientieren, um nicht die Konzepte zu überdecken, die zur Erreichung dieser Ziele für das Simulatorprogramm implementiert worden sind.

Als die wichtigsten Anforderungen für SimpleGrid seien hier Einfachheit, Transparenz, Fehlertoleranz, Ausfallsicherheit und Skalierbarkeit genannt.

4.1.1 Transparenz

Unter dem Begriff der Transparenz wird die Tatsache verstanden, dass die gesamte Komplexität der Simulationsumgebung vor dem Anwender verborgen bleiben soll. Die möglicherweise sehr große Anzahl an Simulationsrechnern muss weitgehend autonom und ohne die Notwendigkeit von Benutzereingriffen innerhalb der Simulationsumgebung agieren können. Für den Betrieb von SimpleGrid soll einzig ein Systemadministrator verantwortlich sein, der gelegentliche Wartungsarbeiten und Konfigurationsanpassungen durchführt.

Idealerweise sollte dem Benutzer die Simulationsumgebung als eine Art *Black Box* erscheinen. D. h., für die Durchführung eines Simulationsauftrages muss es genügen, alle dafür benötigten Daten an ein Programm zu übergeben, welches dann alle weiteren notwendigen Schritte automatisch übernimmt. Somit dürfen die Schritte für die Durchführung von Netzwerksimulationen nicht davon abhängig sein, ob man mit oder ohne Hilfe der Simulationsumgebung arbeiten möchte.

4.1.2 Einfachheit

Die Benutzung und Verwaltung der Simulationsumgebung muss sowohl für den Benutzer als auch für den Administrator möglichst einfach bleiben. Wie auch schon bei der Transparenzanforderung erwähnt, sollte es für den Endbenutzer keine Rolle spielen, ob seine Simulationsaufträge mit Hilfe einer komplexen Simulationsumgebung durchgeführt werden oder auf einem einzelnen Rechner. Die Konfiguration der Simulationen darf davon nicht betroffen sein.

4.1.3 Fehlertoleranz und Ausfallsicherheit

Das unerwartete Auftreten von Fehlersituationen muss erkannt und automatisch behandelt werden. Fehlersituationen können bspw. die in Abschnitt 3.3.6 aufgeführten Speicherverwaltungsprobleme sein. Es muss auf Fälle reagiert werden können, bei denen entweder zu wenig Hauptspeicher für einen Simulationslauf verfügbar ist, oder ein Simulationsrechner exzessiv auf virtuellen Speicher zugreift.

Die Ausfallsicherheit betrifft die adäquate Reaktion auf das Versagen einzelner Bestandteile der Simulationsumgebung. Dazu gehört z. B. der Ausfall von Simulationsrechnern, die gerade einen Simulationslauf bearbeiten. In solchen Fällen müssen die abgebrochenen Simulationen auf anderen freien Rechnern wiederholt werden. Die Stabilität der Simulationsumgebung darf ebenfalls nicht von Software-Fehlern beeinträchtigt werden. Falls bisher noch unentdeckte Fehler in der Implementierung der SimpleGrid-Module vorhanden sind oder später neue Fehler eingeführt werden, dürfen diese nicht zu einem Totalausfall der gesamten Simulationsumgebung führen. Die Auswirkungen dieser Fehler müssen daher möglichst abgefangen und ausgeglichen werden.

4.1.4 Skalierbarkeit

Das Hinzufügen von weiteren Rechnern zur Simulationsumgebung muss transparent und auch während der Durchführung von Simulationsaufträgen geschehen können. Gleichzeitig muss es den teilnehmenden Rechnern möglich sein, sich zu beliebigen Zeitpunkten aus der Simulationsumgebung abzumelden. Dies alles darf den Betrieb von SimpleGrid nicht beeinträchtigen. Zudem soll das An- und Abmelden von Simulationsrechnern automatisch, also ohne das Eingreifen eines Administrators geschehen können.

4.2 Architektur von SimpleGrid

An dieser Stelle soll nun die grundlegende Architektur beschrieben werden, die für SimpleGrid zur Bewältigung der gestellten Aufgaben zum Einsatz kommt.

Die Simulationsumgebung setzt sich aus der Installation dreier Softwareinstanzen zusammen. Diese interagieren im Verbund einer Client-Server-Architektur miteinander. Die Hauptkomponenten sind dabei der *Gridserver*, die *Gridnodes* und ein *Client-Programm*. In den nachfolgenden Abschnitten werden die Aufgaben dieser einzelnen Instanzen näher umrissen.

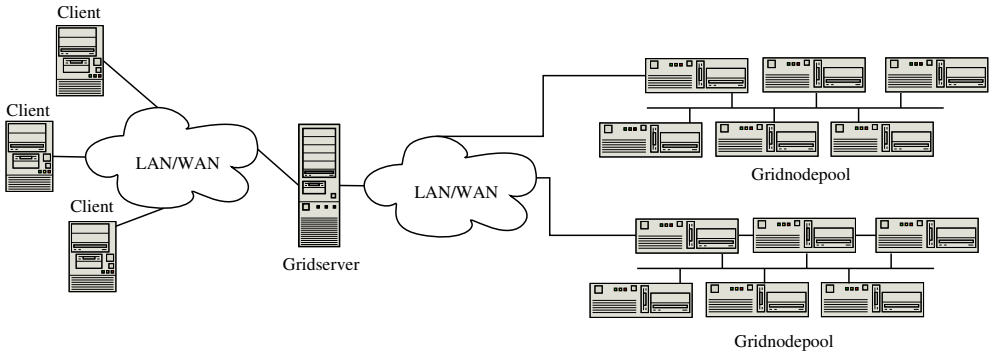


Abbildung 4.1: Beispiel für die Organisation einer SimpleGrid-Installation

4.2.1 Gridserver

Der Gridserver ist die zentrale Schaltstelle der Simulationsumgebung. Er verwaltet die Struktur des gesamten Rechenverbunds und ermöglicht die Interaktion aller beteiligter Komponenten. Seine Hauptaufgabe ist es, die Simulationsaufträge der Benutzer entgegenzunehmen und ihre Ausführung zu organisieren. Wie in Abschnitt 3.2.2 beschrieben unterteilt er dafür einen Simulationsauftrag in entsprechende Teilmengen von komplexitätsäquivalenten Simulationsläufen und weist diese an die zur Verfügung stehenden freien Rechner zu. Der für diese Aufgabe benötigte Scheduler bildet dabei den Hauptbestandteil des Servers.

Abbildung 4.2 illustriert, wie die Simulationsverteilung gehandhabt wird. Für die Zuweisung eines Simulationslaufs schickt der Server einem freien Rechner die ausführbare JAR-Datei des Netzwerksimulators zusammen mit allen notwendigen Konfigurationsdaten zu. Wurde dem Simulationsrechner schon vorher für den aktuell durchgeführten Simulationsauftrag die Simulatordatei geschickt, so muss dem Rechner lediglich die Konfiguration für den nächsten Simulationslauf übertragen werden. Dies ist in Abbildung 4.2 für den mit „Gridnode 2“ bezeichneten Rechner angedeutet. Der Rechner führt nun die Simulation aus und schickt dem Server anschließend alle Ergebnisdateien zurück.

Die zweite Hauptaufgabe des Gridservers ist die Verwaltung aller verfügbaren Rechenressourcen. Diese werden von den Computern dargestellt, die als Arbeitsmaschinen für die Durchführung einzelner Simulationsläufe verfügbar sind. Der Server koordiniert die für die Kapazitätsmessungen notwendigen Benchmarkläufe. Weiterhin nimmt er die während der Durchführung eines Simulationsauftrags anfallenden Simulationsergebnisse in Form von Trace- und Statistikdateien entgegen, um sie zentral abzuspeichern. Wie in Abbildung 4.2 dargestellt, kann der Benutzer von SimpleGrid nach Durchführung eines Simulationsauftrags die gesammelten Ergebnisdateien von dieser zentralen Stelle abholen.

Abbildung 4.1 stellt beispielhaft dar, wie eine Simulationsumgebung mit Hilfe von SimpleGrid aufgebaut werden kann. Man sieht, dass der Gridserver das zentrale Element einer solchen Installation darstellt. Alle anderen SimpleGrid-Module sind mit ihm über das Netzwerk verbunden.

4.2.2 Gridnode

Als Gridnodes (oder auch *Knoten*) werden diejenigen Maschinen bezeichnet, die im Rechenverbund der Simulationsumgebung für die Durchführung von Simulationsläufen zur Verfügung stehen. Es können beliebige Rechner unabhängig von deren eigentlichem Einsatzzweck als Gridnode konfiguriert werden. Es kann sich bei ihnen bspw. um einfache Bürorechner oder speziell für Netzwerksimulationen vorgesehene Computer handeln. Um einen Computer als Gridnode zu konfigurieren, startet man auf diesem einfach das Gridnode-Programm. Dieses Programm verbindet sich daraufhin mit dem Gridserver und zeigt ihm seine Bereitschaft an, Simulationsläufe entgegenzunehmen. Liegt beim Server ein Simulationsauftrag vor, so werden die einzelnen Läufe an alle verfügbaren Gridnodes verteilt. Die Knoten starten daraufhin den Simulator lokal mit diesen Eingabedaten und schicken schließlich die Simulationsergebnisse dem Gridserver zurück.

Gridnodes können sich beliebig am Server an- und wieder abmelden. Die Bedingungen, nach denen diese An- und Abmeldevorgänge geschehen, werden ausschließlich von den Knoten selbst festgelegt. Die Arbeit des Servers wird davon nicht beeinträchtigt. Er ist darauf vorbereitet, dass Gridnodes auch unerwartet ausfallen können. Auf diese Weise ist es z. B. denkbar, dass man das Gridnode-Programm an einen Bildschirmschoner koppelt. Wird der aktiv, so kann gleichzeitig das Gridnode-Programm gestartet werden und der entsprechende Rechner damit an der Simulationsumgebung teilnehmen. Bei Deaktivierung des Bildschirmschoners meldet sich auch der Gridnode vom Server ab. Damit erreicht man ein ähnliches Verhalten, wie bei den anfangs vorgestellten Projekten SETI@home oder Folding@home: ein Computer stellt genau dann seine Rechenleistung für Netzwerksimulationen zur Verfügung, wenn dieser gerade nicht benutzt wird.

Wie in Abbildung 4.1 gezeigt, können sich die verschiedenen Gridnodes in unterschiedlichen lokalen Netzen befinden. Die Gesamtheit aller Knoten, die sich unter einer bestimmten administrativen Kontrolle befinden, wird hier als *Gridnodepool* bezeichnet. Diese Pools sind wiederum über das Netzwerk mit dem Server verbunden.

4.2.3 Client Tool

Das Client-Programm dient dem Endbenutzer als Zugang zu den Dienstleistungen der Simulationsumgebung. Der Benutzer kann damit dem Server einen kompletten Simulationsauftrag übertragen, damit dieser die Organisation der Simulationsausführung übernimmt. Dazu werden alle Dateien zusammengefasst, die für ein Simulationsprojekt benötigt werden, und dem SimpleGrid Client übergeben. Der Client verbindet sich über das Netzwerk mit dem Server und überreicht ihm alle Konfigurationsdaten zusammen mit einer Simulatordatei, die den Netzwerksimulator selbst enthält. Dies wird in der linken Hälfte von Abbildung 4.2 angedeutet.

Das Client Tool kann von einem beliebigen Computer aus benutzt werden. Es können sich grundsätzlich eine beliebige Anzahl von Client-Instanzen mit dem Server verbinden. Diese dürfen sich dabei auch in einem anderen Netzwerk befinden, als der Server.

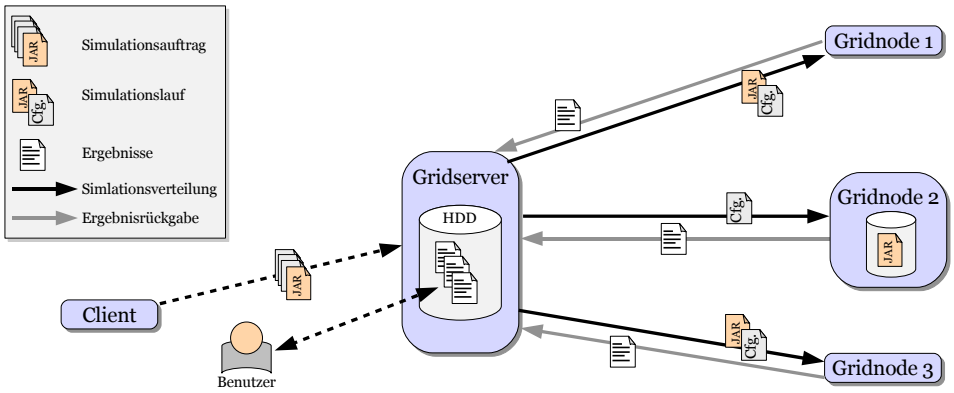


Abbildung 4.2: Schema für die Durchführung eines Simulationsauftrags

4.2.4 Weitere Designentscheidungen für die Softwareimplementierung

SimpleGrid wurde ebenso wie der Netzwerksimulator SimpleSim vollständig in Java implementiert. Dadurch wird die Integration des Simulators in der Simulationsumgebung sehr einfach. Zusätzlich bietet sich durch die Plattformunabhängigkeit der Programmiersprache der Vorteil, dass die drei Module von SimpleGrid auch auf unterschiedlichen Betriebssystemen eingesetzt werden können [38].

Die Behandlung aller netzwerkbezogenen Aufgaben innerhalb der SimpleGrid-Module wurde mit den in der Version 1.4 des Java-Frameworks eingeführten Klassen des *Java NIO*-Pakets (Java New I/O [17, 19]) implementiert. Dieses Paket erlaubt im Gegensatz zu den herkömmlichen Klassen des `java.net`-Pakets die Implementierung von sehr gut skalierenden Netzwerkanwendungen. Mit Java NIO wurde die Möglichkeit eingeführt, nicht-blockierende Netzwerk-Sockets zu verwenden. Dies bedeutet, dass die Funktionen zur Behandlung von Netzwerknachrichten nicht erst auf die Verfügbarkeit von Daten warten müssen. Die entsprechenden Funktionsaufrufe kehren sofort zurück, falls momentan keine neuen Daten anstehen [17]. Damit ist es nicht mehr nötig, für jede bestehende Netzwerkverbindung einen eigenen Thread zu erzeugen. Statt dessen können sich einige wenige Threads um die Behandlung des gesamten Netzwerkverkehrs kümmern. Auf diese Weise kann eine sehr große Zahl an Verbindungen gleichzeitig verwaltet werden, ohne dass der Prozessor durch den Kontextwechsel zwischen Threads ausgebremst wird.

Diese Eigenschaften von Java NIO machen das Paket besonders geeignet für die Implementierung verteilter Rechenanwendungen wie SimpleGrid. Hier ist es nicht ausgeschlossen, dass eine sehr große Anzahl von Gridnodes gleichzeitig mit dem Server verbunden ist und dabei große Datenmengen ausgetauscht werden. Bei der Entwicklung von SimpleGrid wurde darauf geachtet, dass auch solche Situationen problemlos möglich sind. Aus diesem Grund wurde auch nicht auf Techniken verteilter Objekte, wie z. B. RMI [18] oder CORBA [46], zurückgegriffen. Der durch diese Verfahren produzierte Overhead (hervorgerufen z. B. durch Objektserialisierungen) ließ die Techniken nicht für die Anforderung einer guten Skalierbarkeit geeignet erscheinen. Eine Gegenüberstellung

von Java NIO und der herkömmlichen Netzwerkbehandlung mit dem *java.net*-Paket und RMI im Hinblick auf Hochleistungsanwendungen findet sich in [35].

4.3 Arbeitsweise von SimpleGrid

Nachdem in den vorangegangenen Kapiteln die grundlegenden Überlegungen zur Gestaltung einer automatisierten Simulationsumgebung dargelegt und die an sie gestellten Anforderungen umrissen wurden, kann nun eine Beschreibung der konkreten Arbeitsweise von SimpleGrid erfolgen. In den nächsten Abschnitten wird eine Zusammenfassung der Interaktionen zwischen den Modulen Gridserver, Gridnode und Client Tool gegeben.

4.3.1 Durchführung von Benchmarks

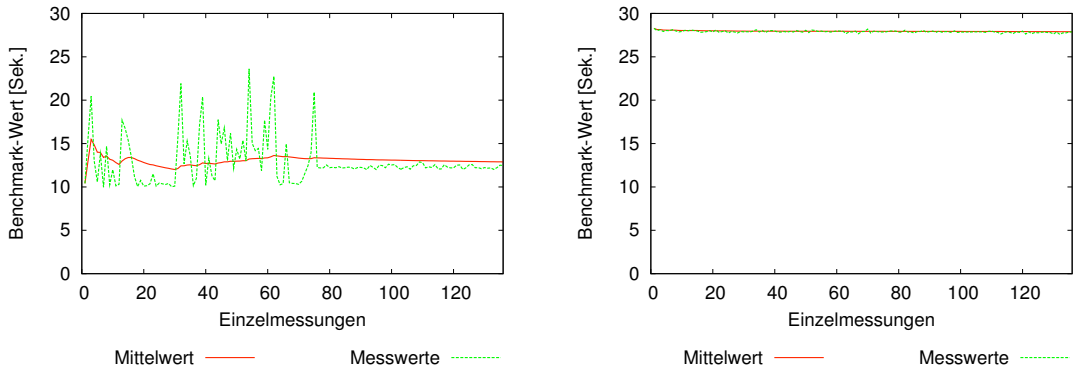
In Abschnitt 3.3.4 wurde das Konzept der Kapazitätsmessung mit Hilfe von Benchmarks eingeführt. In SimpleGrid wird diese Methode eingesetzt, um die Kapazitätswerte unbekannter Rechner zu ermitteln. Meldet sich ein neuer Gridnode am Server an, so muss zunächst dessen Kapazitätswert κ ermittelt werden. Bevor dies nicht geschehen ist, kann er nicht für die Berechnung von Simulationsläufen eingesetzt werden, da für ihn noch keine Schätzung von Rechenzeiten möglich ist. Der Server schickt dem Knoten dazu die für die Benchmarkläufe konfigurierte Simulatordatei mit allen dazugehörigen Konfigurationsdaten. Diese Benchmarksimulation wird nun von dem zu vermessenden Gridnode durchgeführt. Gleichzeitig wird die dabei benötigte Zeit gemessen und anschließend dem Server übertragen.

Die Benchmarkläufe werden mit einem vorgegebenen Zeitintervall wiederholt und die dabei ermittelte Simulationsdauer mit allen zuvor erhobenen Werten zu der Kapazitätskennzahl κ gemittelt. Auf diese Weise erhält man mit κ eine Kennzahl für die Leistungsfähigkeit des Gridnodes, die auf die durchschnittliche Lastsituation des Knotens Rücksicht nimmt.

Die Gridnodes merken sich nach jedem Benchmarklauf den aktuellen Wert von κ in einer Datei. Dadurch ist es für sie möglich, dem Server ihren zuletzt berechneten Wert für κ mitzuteilen, falls sie sich erneut mit ihm verbinden müssen. Es wird damit verhindert, dass vergangene Messungen verloren gehen, falls ein Gridnode die Verbindung zum Server verliert.

Schließlich sei hier noch am Rande erwähnt, dass jegliche Ausgaben, die der Simulator während einer Simulation erzeugt (also Statistiken und Trace-Daten), bei Benchmarkläufen deaktiviert sind. Da für das Benchmarking nur die reine Rechendauer der Simulation zählt und die Benchmarkkonfiguration ohnehin beliebig ist, sind solche Ausgaben hier nicht weiter von Interesse.

Abbildung 4.3 stellt zwei Beispiele für den Verlauf einer Folge von Benchmarkdurchläufen dar. Hier wurden auf zwei verschiedenen Rechnern über einen Zeitraum von 22 Stunden Benchmarks mit einem Abstand von 10 Minuten durchgeführt. Die beiden Graphen zeigen jeweils die gemessenen Benchmarkwerte (in Sekunden) und den laufenden Mittelwert. An letzterem lässt sich der Wert für κ direkt ablesen. Der für Abbildung 4.3(a) vermessene Rechner A wurde während der Messungen für die Erstellung dieser



(a) Rechner A mit stark fluktuierender Last

(b) Rechner B ohne Last

Abbildung 4.3: Messungen der Kapazitätswerte auf zwei Rechnern, die unterschiedliche Lastsituationen aufweisen. Die Benchmarks wurden mit einem Abstand von 10 Minuten und über einen Zeitraum von 22 Stunden durchgeführt.

Arbeit verwendet. Dadurch entstand auf dem Computer in unregelmäßigen Abständen eine starke Rechenlast. Die großen Schwankungen in den gemessenen Benchmarkwerten lassen sich auf diese Tatsache zurückführen. Etwa nach der achtzigsten Messung kann man erkennen, wie sich die Messkurve auf einen Wert von etwa 12 Sekunden einpendelt. Dieser Abschnitt markiert eine Nachtphase, in der an Rechner A nicht gearbeitet wurde. Es fällt weiterhin auf, dass tagsüber bei den Benchmarks Werte von teilweise weniger als 11 Sekunden gemessen werden, während nachts die Werte im Schnitt bei 12 Sekunden liegen. Dieser Unterschied lässt sich auf den Effekt des Bildschirmschoners zurückführen, der einen gewissen Teil der Rechenleistung beansprucht.

Zum Vergleich zeigt Abbildung 4.3(b) die Benchmarkwerte eines zweiten Rechners B mit einer anderen Hardwarekonfiguration als Rechner A. Dieser Computer blieb während der gesamten Messphase unbenutzt. Man sieht hier deutlich, dass die Messwerte relativ stabil bleiben und nur wenig schwanken. Da Rechner B verglichen mit Rechner A weniger leistungsfähig ist, werden hier höhere Werte für die Rechendauer der Benchmarkläufe gemessen.

Einen Eindruck, wie einige Benchmarkwerte beschaffen sein können, verschafft beispielhaft Tabelle 4.1. Hier sind die Kapazitätswerte von fünf unterschiedlich leistungsfähigen Computern aufgeführt, die abwechselnd mit drei verschiedenen Benchmarkkonfigurationen vermessen worden sind. Die Messungen fanden allesamt in der Nacht statt, so dass die Benchmarks zumeist die einzigen Benutzerprozesse waren, welche auf den Rechnern ausgeführt wurden. Die in den Testrechnern arbeitende CPU wird in der zweiten Spalte benannt. Diese Angabe dient nur als grober Hinweis auf die jeweiligen Leistungsklassen, denen die Rechner zugeordnet werden können. Natürlich ist die verwendete CPU nicht alleiniger Einflussfaktor für die Kapazitätskennzahlen. In den mit κ_i beschrifteten Spalten ist der jeweilige Kapazitätswert der Rechner für die Benchmarkkonfiguration i ($i = 1, \ldots, 3$) aufgeführt. Dieser ergibt sich als der Mittelwert von 60

Nr.	Maschine	κ_1	s_1	κ_2	s_2	κ_3	s_3
1	AMD Athlon64 3700+ 2,2 GHz	10,191	0,314	56,740	0,932	23,791	0,369
2	Intel Pentium 4 CPU 3,2 GHz	13,258	0,123	55,286	0,199	23,579	0,157
3	AMD Opteron 250 2,4 GHz	9,326	0,633	38,465	1,292	16,666	0,402
4	Intel P4 Mobile CPU 1,6 GHz	29,319	0,218	119,105	0,500	50,313	1,476
5	AMD Athlon 1,4 GHz	27,945	0,103	144,770	0,410	59,826	0,570

Tabelle 4.1: Mittelwerte $\bar{X}_i$ und Standardabweichungen s_i (beide in Sekunden) dreier unterschiedlicher Benchmarkkonfigurationen, die auf 5 verschiedenen Rechnern ausgeführt wurden. Es wurden dafür 60 Benchmarkläufe mit einem Abstand von 10 Minuten durchgeführt.

Messungen, die mit einem Abstand von 10 Minuten stattgefunden haben. Die mit s_i beschrifteten Spalten stellen die jeweilige Standardabweichung der Messungen dar. Man sieht, dass sie relativ gering ausfallen. Dies lässt sich auf die Tatsache zurückführen, dass die Rechner während der Benchmarkläufe die meiste Zeit untätig waren. In diesen konkreten Standardabweichungen drücken sich daher ausschließlich kleinere Hintergrundprozesse aus, die nur wenig Rechenkapazität beanspruchen.

4.3.2 Scheduling von Simulationsaufträgen

Wenn ein Benutzer mit Hilfe des Client-Programms der Simulationsumgebung einen Auftrag zur Bearbeitung übermittelt hat, kann der Scheduler des Gridservers mit seiner Arbeit beginnen. Er geht dafür vor, wie in Abschnitt 3.4 beschrieben wurde. Zunächst bildet er die Mengen der komplexitätsäquivalenten Simulationsläufe und verteilt deren zufällig gewählte Repräsentanten mit Hilfe des List Schedulers auf die schnellsten der verfügbaren Simulationsrechner. Sobald die ersten Komplexitätswerte bekannt sind, können die restlichen Scheduling-Algorithmen zum Einsatz kommen. Der Scheduler wählt den zu benutzenden Algorithmus nach dem folgenden Schema: liegen ihm mehr als 1.000 zu verteilende Simulationsläufe vor, so greift er auf den Greedy Scheduler zurück. Andernfalls verwendet er die Branch And Bound-Methode, bei der die Startlösung mit Hilfe des Simulated Annealing-Verfahrens ermittelt wird. Der Schwellwert von 1.000 Simulationsläufen wurde als ein grober Richtwert auf Grundlage der in Abschnitt 3.1 besprochenen Graphen gewählt. Bei der Verteilung einer größeren Anzahl von Jobs würde die Rechendauer der Simulated Annealing-Methode zu sehr ins Gewicht fallen.

Nachdem der Scheduler einen Maschinenbelegungsplan erstellt hat, können die Simulationsjobs an die ihnen zugewiesenen Gridnodes zur Bearbeitung übertragen werden. Der Schedule wird damit nach und nach abgearbeitet. Hat ein Knoten den ihm zugewiesenen Simulationslauf erledigt, lässt er sich vom Server anschließend den nächsten Job schicken, der laut Schedule von ihm berechnet werden muss.

Hierbei kann ein reibungsloser Ablauf jedoch nicht garantiert werden. Es ist sehr gut möglich, dass einzelne Gridnodes während der Bearbeitung eines Simulationsprojektes

ausfallen oder sich vom Server für eine unbestimmte Zeit abmelden. Weiterhin können Simulationsläufe nach dem Auftreten von Fehlern auf den Knoten abgebrochen werden. Tritt eine solche Situation auf, müsste eigentlich ein neuer Maschinenbelegungsplan berechnet werden, bei dem die ausgefallenen Maschinen nicht mehr berücksichtigt und die unterbrochenen Simulationsläufe erneut eingeteilt werden. Für den ungünstigen Fall, dass unter den am Server angemeldeten Gridnodes eine hohe Fluktuation herrscht oder häufig Simulationen abgebrochen werden müssen, würde dies zu einer relativ häufigen Neuberechnung des Schedules führen. Nun hat man in Kapitel 3.1.1 gesehen, dass die Branch And Bound-Methode für die Berechnung einer optimalen Lösung u. U. eine lange Zeit benötigt. Es kann daher geschehen, dass der Scheduler noch mit der Berechnung des Maschinenbelegungsplans beschäftigt ist, während die nächsten Knoten schon wieder frei werden. Diese müssten dann so lange auf den Scheduler warten, bis er seine Berechnung beendet hat. Dadurch würde die freie Kapazität der Gridnodes verschwendet werden.

Um die Häufigkeit solcher Situationen zu minimieren, verfährt der Gridserver nach der folgenden Strategie. Meldet sich ein Gridnode, der laut Schedule für die Bearbeitung eines Simulationslaufs vorgesehen ist, vom Server ab oder wird ein Simulationslauf auf einem Gridnode abgebrochen, so wird diese Tatsache vom Server vorerst ignoriert. Der vorhandene Maschinenbelegungsplan wird zunächst solange abgearbeitet, bis für einen der Knoten kein weiterer Auftrag mehr bereit steht. Erst dann wird ein neuer Schedule erzeugt. Dieser Fall tritt entweder ein, wenn der Schedule tatsächlich soweit abgearbeitet wurde, dass ein Simulationsrechner alle ihm zugewiesenen Jobs bearbeitet hat, oder wenn zu einem früheren Zeitpunkt eine der oben beschriebenen Situationen eingetreten ist.

Konnte einem freien Gridnode auch dann kein weiterer Simulationslauf zugewiesen werden, nachdem dieser die Erstellung eines neuen Schedules angestoßen hat, so bleibt dieser Rechner unbeschäftigt. Damit wird vermieden, dass das Notwendigwerden der Neuberechnung eines Schedules nicht in eine Endlosschleife gerät.

Trotz der oben beschriebenen Maßnahme zur Minimierung der Häufigkeit von Scheduling-Vorgängen kann es dennoch nicht vollständig verhindert werden, dass ein Knoten frei wird und dieser erst auf die Berechnung eines neuen Maschinenbelegungsplans warten muss. Um die Verschwendung der dabei brachliegenden Rechenkapazität zu vermeiden, wird ihm außer der Reihe derjenige noch ausstehende Simulationslauf zugewiesen, der den geringsten Komplexitätswert hat. Dadurch kann der freie Rechner auch während des Scheduling-Vorgangs weiter beschäftigt werden. Wenn der Scheduler später seine Berechnungen abgeschlossen hat, muss er für alle Jobs, die auf diese Weise freien Rechnern zugewiesen worden sind, überprüfen, ob er sie aus dem neuen Maschinenbelegungsplan entfernen muss. Dies kann in zwei Fällen nötig sein. Wurde ein außerplanmäßig zugewiesener Simulationslauf in der Zwischenzeit zu Ende berechnet, so muss er natürlich aus dem Schedule genommen werden. Anders sieht dies aus, wenn ein solcher Job aktuell noch bearbeitet wird. Dann muss der Scheduler abwägen, welche Vorgehensweise die bessere ist. Sei dazu $M_{aktuell}$ diejenige Maschine, die gerade den Job J außerplanmäßig bearbeitet. Mit M_{Plan} sei die Maschine bezeichnet, die laut des neuen Schedules eigentlich für die Bearbeitung von J verantwortlich ist. Gilt nun

i	Mittlere Komplexität $\bar{\xi}_i$	Standardabweichung s_i	$s_i/\bar{\xi}_i$
1	0,403951	0,100749	24,94 %
2	0,967982	0,216211	22,34 %
3	1,017150	0,211300	20,77 %
4	1,368036	0,343381	25,10 %
5	2,089986	0,518734	24,82 %
6	2,730367	0,843562	30,90 %
7	3,696128	0,958170	25,92 %
8	3,888278	1,077055	27,70 %
9	4,208459	1,175974	27,94 %
10	4,695243	1,330545	28,34 %
11	5,140603	1,489146	28,97 %
12	5,893530	1,316068	22,33 %
13	6,128214	1,508674	24,62 %
14	6,455391	1,968827	30,50 %
15	7,374286	2,191120	29,71 %

Tabelle 4.2: Mittelwert $\bar{\xi}_i$, Standardabweichung s_i und prozentuale Standardabweichung $s_i/\bar{\xi}_i$ für die Komplexitätswerte ξ_{ij} verschiedener Simulationsläufe i ($i = 1, \ldots, 15$), die auf 12 unterschiedlichen Rechnern M_j ($j = 1, \ldots, 12$) gemessen wurden.

$M_{aktuell} = M_{Plan}$, so kann der Scheduler den Auftrag ebenfalls aus dem Maschinenbelegungsplan entfernen, da der Job gerade von der korrekten Maschine bearbeitet wird. Andernfalls schätzt er die Zeit t_{Plan}, die Auftrag J auf M_{Plan} voraussichtlich benötigen wird, und die Restzeit t_{Rest}, die $M_{aktuell}$ noch brauchen wird, um J fertig zu stellen. Ist $t_{Rest} < t_{Plan}$, so kann auch hier J aus dem Schedule entfernt werden. Andernfalls wäre es günstiger, den Job J auf der Maschine M_{Plan} zu berechnen, die laut Schedule für ihn gedacht ist. Der Server bricht daher in diesem Fall die Simulation auf $M_{aktuell}$ ab und belässt J im aktuellen Maschinenbelegungsplan.

Hier sieht man auch, warum der Simulationslauf mit der geringsten Komplexität gewählt wird, um freie Gridnodes während des Scheduling-Vorgangs zu beschäftigen. Damit wird sichergestellt, dass das Abbrechen einer solchen Simulation nicht allzu häufig notwendig wird.

Die Flussdiagramme in den Abbildungen 4.4 und 4.5 fassen diesen Vorgang zusammen.

4.3.3 Schätzung der voraussichtlichen Simulationslaufzeiten

In Abschnitt 3.3.5 wurde beschrieben, wie mit Hilfe des Kapazitätswertes κ_k eines Rechners M_k und des Komplexitätswerts ξ_l einer Simulation J_l eine Schätzung $\hat{T}_{kl}$ für die voraussichtliche Dauer der Simulationsdurchführung von J_l auf dem Rechner M_k erhalten werden kann. Eine entsprechende Vorschrift gibt Gleichung (3.3). Weiterhin wurde eine lineare Beziehung zwischen den einzelnen κ_j und den ξ_i unterstellt. Diese Bezie-

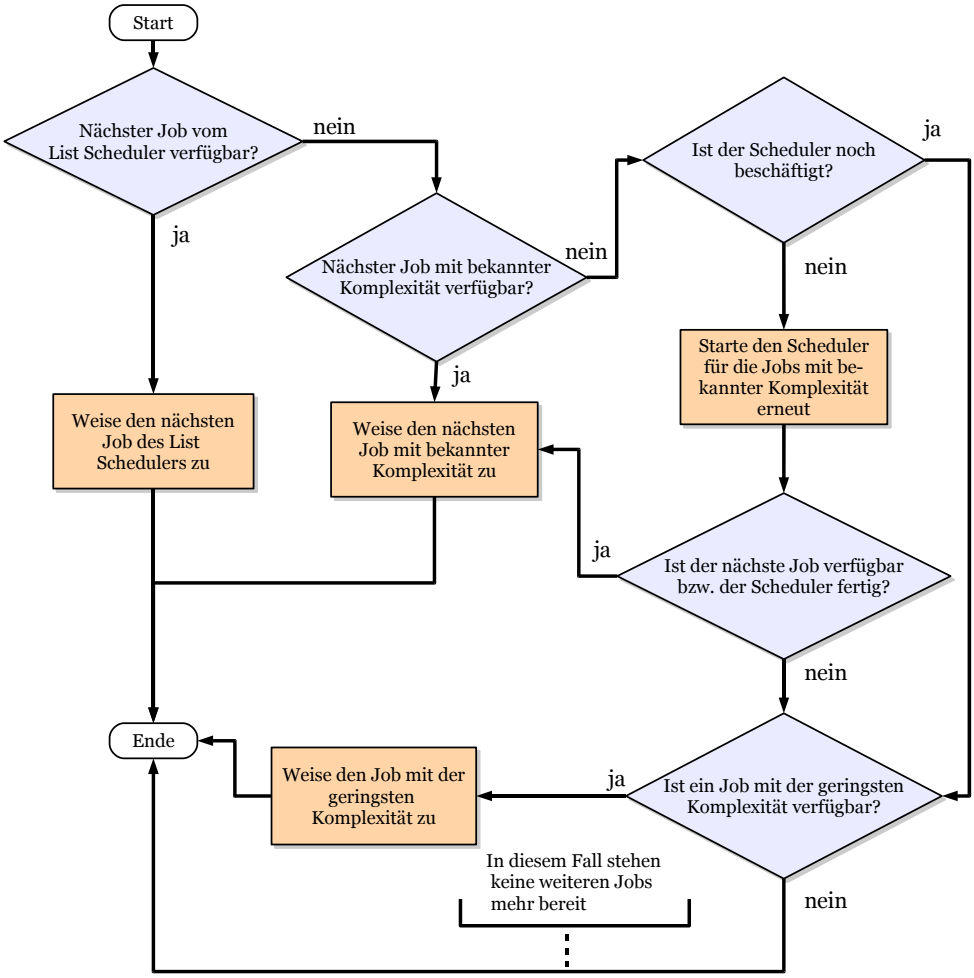


Abbildung 4.4: Flussdiagramm für die Simulationsverteilung des Schedulers. Hier ist der Fall dargestellt, der eintritt, wenn ein Simulationsrechner den nächsten Simulationslauf anfordert.

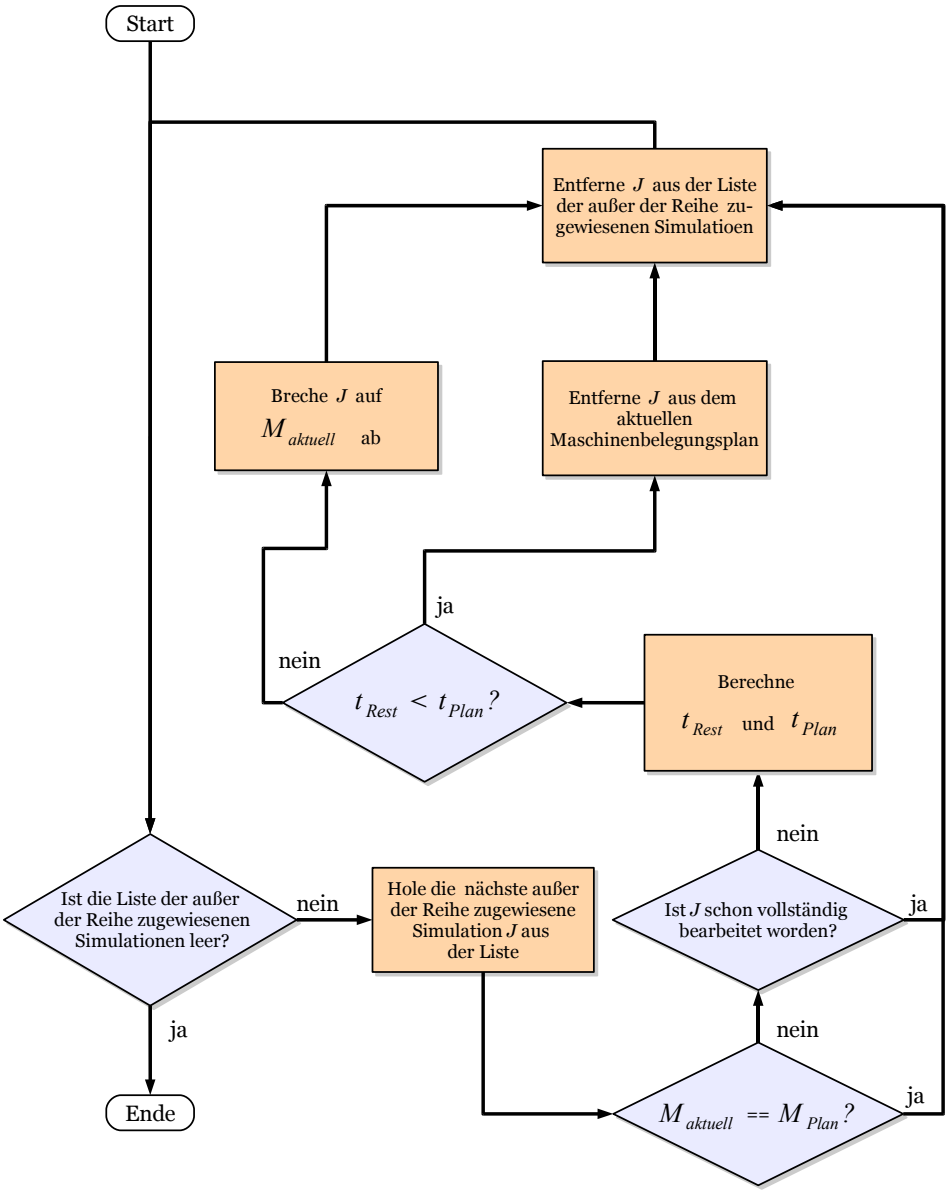


Abbildung 4.5: Flussdiagramm für die Entscheidungen, die nach Beendigung des Scheduling-Vorgangs für die außer der Reihe zugewiesenen Simulationsläufe getroffen werden.

hung ist eine Voraussetzung dafür, dass die Vorschrift (3.3) angewandt werden kann. Aus diesem Grund müsste nun anhand statistischer Tests untersucht werden, ob diese Beziehung tatsächlich besteht. Dabei ergibt sich jedoch das Problem, dass für eine solche Untersuchung eine hohe Anzahl unterschiedlicher Simulationskonfigurationen und eine große Zahl verschiedener Computersysteme benötigt werden, um eine ausreichend große Stichprobe für den notwendigen statistischen Test zu bekommen. Dies war im Rahmen dieser Arbeit technisch nicht möglich.

Um dennoch einen Anhaltspunkt zu erhalten, wurden 268 verschiedene Simulationen auf insgesamt 12 unterschiedlichen Computern durchgeführt. Dabei wurden die jeweiligen Komplexitätswerte ξ_{ij} gemessen, die sich für jede Simulation J_i auf jedem System M_j ergeben. Die Rechner wurden während der Messungen nicht für andere Aufgaben benutzt. Damit sollte eine zu starke Verzerrung der Messwerte verhindert werden.

Anschließend wurde für jede Simulationskonfiguration der Mittelwert $\bar{\xi}_i$ berechnet, indem die für eine bestimmte Konfiguration auf jedem Rechner gemessenen Komplexitätswerte gemittelt wurden. Desweiteren wurde für die Messungen die Standardabweichung s_i berechnet. Um einen Eindruck davon zu erhalten, wie stark die einzelnen ξ_{ij} um den Mittelwert $\bar{\xi}_i$ streuen, wurde außerdem der Anteil $s_i/\bar{\xi}_i$ der Standardabweichung am Mittelwert ausgerechnet.

Es wird erwartet, dass die Streuung der gemessenen Werte möglichst gering ist. Ist dies der Fall, so würde das darauf hindeuten, dass die Komplexitätswerte eines Simulationslaufs tatsächlich weitgehend unabhängig von einem bestimmten Rechner sind. Es würde dann auf jedem Computersystem für einen Simulationslauf etwa der gleiche Komplexitätswert ermittelt werden. Somit wäre mit Gleichung (3.3) eine gute Schätzung der voraussichtlichen Simulationsdauer möglich.

Tabelle 4.2 zeigt einen Ausschnitt aus den ermittelten Werten. Man kann erkennen, dass der Anteil der Standardabweichung an den Komplexitätswerten ungefähr gleichbleibt. Der Mittelwert dieser Anteile für alle 268 Simulationskonfigurationen liegt bei 24,89%. Die Standardabweichung zu diesem Mittelwert beträgt 3,77%.

Aus diesen Zahlen lässt sich nun der Hinweis ableiten, dass zwar die absoluten Schwankungen der gemessenen Komplexitätswerte mit rechenintensiver werdenden Simulationen zunimmt, diese aber prozentual immer etwa gleich ausfällt. Das bedeutet, dass auch der Fehler bei der Schätzung von $\hat{T}_{kl}$, welcher durch die Schwankungen entsteht, prozentual konstant bleibt.

Es kann also – unter Vorbehalt – angenommen werden, dass mit Gleichung (3.3) tatsächlich eine gute Schätzung der voraussichtlichen Simulationsdauer möglich ist.

Damit der Scheduler von SimpleGrid möglichst mit dem repräsentativeren Komplexitätswert $\bar{\xi}_i$ anstatt mit dem Wert einer einzigen Messung ξ_{ij} arbeiten kann, wird in der Simulationsumgebung die folgende Strategie angewandt. Der für eine Menge komplexitätsäquivalenter Simulationsläufe stehende Wert ξ_i wird jedesmal aktualisiert, wenn eine weitere Simulation aus dieser Menge durchgeführt wurde. Das heißt also, dass bei jeder Simulationsdurchführung gleichzeitig die Komplexität dieser Simulation berechnet wird, damit das laufende Mittel der dabei erhaltenen Messwerte gebildet werden kann.

4.3.4 Umgang mit Fehlersituationen

Als eine wichtige Anforderung an SimpleGrid wurde in Abschnitt 4.1 die Fehlertoleranz genannt. Das Auftreten von Fehlern innerhalb der Simulationsumgebung darf nicht dazu führen, dass die Funktionalität von SimpleGrid beeinträchtigt wird. Die häufigsten Fehlersituationen können bei der Durchführung von Simulationsläufen entstehen. SimpleGrid verfolgt dabei besondere Strategien, um sich von solchen Fehlern zu erholen und sie gleichzeitig zu vermeiden.

Eine Quelle möglicher Probleme findet sich in der Speicherverwaltung der Simulationsrechner. Verfügt ein Gridnode nicht über ausreichend Hauptspeicher, um eine Simulation durchzuführen, so hält diese mit der Java-Ausnahme `OutOfMemoryError` [19] an. Der Simulator fängt diesen Fehler ab und meldet den Vorfall dem Gridserver. Um eine solche Situation von vornherein zu vermeiden, wendet die Simulationsumgebung den in Abschnitt 3.3.6 vorgestellten Mechanismus an: es wird von den Gridnodes bei jedem Simulationslauf J_i gemessen, wieviel Speicher dabei maximal verbraucht worden ist. Der Scheduler wird dann die zu J_i komplexitätsäquivalenten Jobs in $\bar{\mathcal{P}}_i$ nicht mehr auf Knoten legen, die über weniger Hauptspeicher verfügen, als für J_i verbraucht wurde.

Eine weitere Situation, die erkannt und behandelt werden muss, tritt ein, sobald der Simulationsrechner exzessiv auf Auslagerungsspeicher zurückgreift. Das dabei entstehende Memory Swapping verzögert die Durchführung von Simulationen derart extrem, dass die gleichzeitig stattfindenden Komplexitätsmessungen zu stark verzerrt werden. Aus diesem Grund überwacht ein Gridnode während er eine Simulation durchführt den Zugriff auf den Auslagerungsspeicher und bricht die Simulation sofort ab, falls er exzessives Memory Swapping entdeckt.

Schließlich sind auch Fehlersituationen denkbar, die durch Programmierfehler entstehen können. Es ist gut möglich, dass nach Erweiterungen oder Änderungen am Programmcode des Simulators oder der Simulationsumgebung neue Fehler eingeführt werden, die zu einem unerwarteten Verhalten der Simulationsumgebung führen. So ist es z. B. denkbar, dass nach einer Änderung am Quelltext des Simulators in manchen Fällen eine Simulation in eine Endlosschleife gerät. Dies würde dazu führen, dass der Knoten, auf dem diese Simulation läuft, so lange blockiert ist, bis ihn ein Administrator neu startet. Ein solcher Fall kann von den Knoten nicht direkt erkannt werden. Aus diesem Grund wurde für SimpleGrid ein *Timeout*-Konzept implementiert. Dauert die Durchführung eines Simulationslaufs länger als ein vorgegebener Timeout, so wird dies als ein Absturz des Simulators gewertet und die Simulation abgebrochen. Eine Ursache für ein solches Abstürzen kann z. B. eine nicht gefangene Exception oder eine der oben erwähnten Endlosschleifen sein.

Nun ist es schwierig, für Simulationen, deren Laufzeiten sich prinzipiell sehr stark voneinander unterscheiden können, einen sinnvollen Timeout festzulegen. In SimpleGrid wird daher die folgende Heuristik zum Festlegen der Maximallaufzeit von Simulationen angewandt. Für Simulationsläufe, deren Komplexitätswerte bekannt sind, kann der Scheduler die voraussichtlich benötigte Rechenzeit schätzen. Als Timeout kann man nun ein Vielfaches dieser geschätzten Zeit festlegen. Wird ein solcher Multiplikator bspw. auf 2 gesetzt, so wird die Simulation genau dann abgebrochen, wenn sie länger als das doppel-

te der geschätzten Zeit läuft. Ist für einen Simulationslauf noch kein Komplexitätswert bekannt, so muss man einen festen Timeout wählen, da hier die Verwendung eines Multiplikators nicht möglich ist. Dieser feste Timeout sollte groß genug gewählt werden, so dass damit auch besonders komplexe Simulationsläufe nicht verfrüht abgebrochen werden. Man kann sich bspw. für einen Timeout von einer Stunde entscheiden, wenn man derart lange Rechenzeiten auch für fehlerfreie Simulationsläufe ohnehin nicht zulassen möchte.

Alle bisher aufgeführten Fehlersituationen lassen sich in zwei Klassen einteilen. Die Ursachen für die erste Kategorie von Fehlern sind lokal beschränkt und betreffen nur einen bestimmten Simulationsrechner. Bspw. ist das Auftreten von Memory Swapping ein rein lokales Ereignis auf einem bestimmten Knoten. In die zweite Kategorie fallen alle Fehlersituationen, die ihre Ursache in der Simulationsdurchführung selbst haben. Das Auftreten einer Endlosschleife im Netzwerksimulator ist ein Beispiel hierfür.

Simulationsläufe, die aufgrund von Problemen der ersten Kategorie abgebrochen wurden, können möglicherweise auf einem anderen Rechner erfolgreich wiederholt werden. Dies ist bei der zweiten Fehlerklasse nicht möglich. So ist z. B. die Wiederholung einer Simulation, die beim ersten Versuch mit einer Exception abgestürzt ist, auf einem anderen Rechner nicht sinnvoll. Dort ist das gleiche Verhalten zu erwarten.

Die Simulationsumgebung versucht nun, aufgetretene Fehler nach diesem Schema zu klassifizieren und entsprechend zu handeln. Simulationen, bei denen Fehler mit lokaler Ursache aufgetreten sind, werden auf einem anderen Rechner wiederholt. Die restlichen Simulationsläufe werden nicht wiederholt. Für sie wird im Simulationsprotokoll das Scheitern der Ausführung vermerkt.

Kapitel 5

Erweiterungsmöglichkeiten und Fazit

Neben der in der vorliegenden Arbeit vorgestellten Grundfunktionalität der automatisierten Simulationsumgebung sind für die Zukunft noch einige weitere Programmmerkmale denkbar, mit denen sich SimpleGrid erweitern lässt. Betrachtet man die Funktionsweise und die einzelnen Mechanismen der Simulationsumgebung genauer, ergeben sich viele weitere Ideen, wie diese noch verbessert und ergänzt werden können. Einige dieser Ideen sollen an dieser Stelle kurz angerissen werden.

5.1 Komplexitätsschätzung durch Deduktion

Wie in Abschnitt 4.3.2 dargelegt wurde, muss der Scheduler von SimpleGrid bei der Einplanung der Jobs mit einem gravierenden Nachteil zurechtkommen. Um die einzelnen Simulationsläufe optimal auf die Gridnodes verteilen zu können, braucht er eigentlich Kenntnis über deren zu erwartenden Rechenaufwand. Dieser ist jedoch zu Beginn noch nicht bekannt und lässt sich auch nicht a priori aus den Konfigurationsdateien berechnen. Er muss daher auf den List Scheduler zurückgreifen, um zunächst die Repräsentanten aller Teilmengen komplexitätsäquivalenter Simulationsläufe auf die schnellsten der freien Rechner zu verteilen. Erst nachdem diese Repräsentanten bearbeitet worden sind, stehen ihre jeweiligen Komplexitätswerte fest.

Es wäre nun von Vorteil, wenn der Scheduler die Möglichkeit hätte, die Komplexitätswerte im Voraus zu berechnen. Damit ließe sich die suboptimale Verteilung mit dem List Scheduler vermeiden. Eine denkbare Möglichkeit, dies zu erreichen, ist die Deduktion der Komplexitätswerte aus den einzelnen Belegungen der Konfigurationsparameter. Dafür müsste man den Anteil berechnen, den die einzelnen Parameter an der Komplexität eines Simulationslaufs haben. Auch dafür ist wieder die konkrete Durchführung des entsprechenden Simulationslauf auf einem Gridnode nötig. Jetzt merkt man sich allerdings die berechneten Komplexitätsanteile der einzelnen Parameter z. B. in einer Datenbank. Möchte man dann später eine Simulation mit unbekannter Komplexität durchführen, so lässt sich diese aus den gespeicherten Komplexitätsanteilen vergangener Simulationsläufe ableiten.

5.2 Vereinfachung des Schedulings durch Preemption

Wird die Durchführung eines Simulationslaufs auf einem Knoten abgebrochen, so sind die dabei gewonnenen Zwischenergebnisse unweigerlich verloren. Die entsprechende Si-

mulation muss dann auf einem anderen Rechner wiederholt werden. Führt man nun die Möglichkeit ein, Simulationsläufe zu unterbrechen und auf einem anderen Gridnode fortzuführen, ohne neu beginnen zu müssen, so lassen sich für die Simulationsumgebung zwei nützliche Eigenschaften hinzugewinnen. Zum einen vereinfacht sich die Arbeit des Schedulers, da Scheduling-Probleme mit erlaubtem Preemption einfacher zu lösen sind [6]. Das für den SimpleGrid-Scheduler dann zu behandelnde Optimierungsproblem ließe sich nun als ein Problem der Art $Q \mid pmtn \mid C_{max}$ klassifizieren.

Der zweite gewonnene Vorteil liegt in der Verkürzung der Gesamtsimulationsdauer. Freie Gridnodes, für die keine weitere Arbeit mehr verfügbar ist, könnten die Arbeit von langsameren Rechnern übernehmen.

Für das Abbrechen und Fortführen eines Simulationslaufs auf einem anderen Rechner muss der komplette Zustand des Simulators zum Zeitpunkt des Abbruchs auf den neuen Simulationsrechner transferiert werden [7]. Für diese Aufgabe bietet sich die Fähigkeit der Java-Bibliothek an, Objekte zu serialisieren und über ein Netzwerk zu schicken [38]. Mit Hilfe der Objektserialisierung ließe sich das Hauptobjekt des Simulators zum Zeitpunkt des Abbruchs mit all seinen transitiv über dessen Elementzeiger erreichbaren Unterobjekte auf einen neuen Rechner übertragen und dort wieder in den alten Zustand zurücktransformieren. Die Simulation kann dann an der Stelle fortgeführt werden, an der sie unterbrochen wurde.

5.3 Quality of Service und Benutzerverwaltung

Quality of Service (QoS) stellt ein Konzept dar, bei dem einzelnen Benutzergruppen oder Benutzern Zusicherungen hinsichtlich der Einhaltung bestimmter Qualitätskriterien von Dienstleistungen gemacht werden [49]. Ein solches Konzept lässt sich auch auf die von einer Simulationsumgebung angebotenen Dienstleistungen anwenden.

In der bestehenden Implementierung von SimpleGrid unterscheidet der Scheduler des Gridservers nicht, von welchem Benutzer die verarbeiteten Simulationsaufträge stammen. Es kann gut möglich sein, dass zwei Benutzer gleichzeitig ein Simulationsprojekt in Auftrag geben. Der Scheduler achtet jedoch nicht darauf, ob er die verfügbare Rechenkapazität der Simulationsumgebung gerecht zwischen den beiden Benutzern aufteilt.

Führt man ein QoS-Konzept zusammen mit einer Benutzerverwaltung in SimpleGrid ein, lässt sich dieses Verhalten vermeiden. Mit Hilfe einer Benutzerverwaltung hat der Server die Möglichkeit, Simulationsaufträge bestimmten Personen zuzuordnen. Man kann nun bspw. privilegierte Benutzer bestimmen, deren Simulationsprojekte bevorzugt behandelt werden. Durch das Festlegen von QoS-Richtlinien, kann man z. B. bestimmten Anwendergruppen das Aufwenden eines gewissen Prozentsatzes an der verfügbaren Gesamtrechenkapazität für deren Simulationsaufträge garantieren.

5.4 Einführung von Sicherheitsaspekten

Sicherheitsaspekte wurden aus der bisherigen Betrachtung völlig ausgeklammert. Dabei ist dieser Punkt für den realistischen Betrieb einer verteilten Anwendung von besonderer Wichtigkeit. In der Regel sind bei Anwendungen wie SimpleGrid mehrere Parteien involviert. Dies sind zum einen die Benutzer, welche die Dienste der Simulationsumgebung in Anspruch nehmen. Zum anderen sind eine Reihe von Administratoren beteiligt, die bestimmte Rechner unter ihrer Verwaltung der Simulationsumgebung als Gridnodes zur Verfügung stellen. Schließlich ist ein Administrator für den Betrieb von SimpleGrid selbst verantwortlich.

Jede dieser Parteien hat bestimmte Interessen, die von der Simulationsumgebung beachtet werden müssen. So ist es für die Administratoren der Gridnode-Rechner wichtig, dass der normale Betrieb dieser Computer nicht durch die SimpleGrid-Software gestört wird. Die SimpleGrid-Module müssen daher dafür sorgen, dass auch tatsächlich nur Instanzen des SimpleSim-Netzwerksimulators auf den Gridnodes ausgeführt werden, und keine andere potenziell schädliche Software.

Der Verwalter des Gridservers kann bspw. daran interessiert sein, nur bestimmten Personen Zugriff auf die Dienste der Simulationsumgebung zu erlauben. Es muss daher eine vor Mißbrauch schützende Benutzerverwaltung implementiert werden.

5.5 Messung der verfügbaren Netzwerkbandbreite für die Berücksichtigung beim Scheduling

Auch die Effekte der verfügbaren Netzwerkbandbreite wurde in dieser Arbeit vorerst ausgeklammert. Sind alle an der Simulationsumgebung beteiligten Rechner mit einer ausreichend hohen Bandbreite mit dem Gridserver verbunden und hält sich die Menge der verschickten Daten (also Statistik- und Trace-Dateien etc.) in Grenzen, so ist der Effekt der Netzwerkbandbreite für das Scheduling vernachlässigbar. Dies ist z. B. der Fall, wenn sich alle Rechner der Simulationsumgebung im selben lokalen Netz befinden. Gibt es allerdings Gridnodes, die nur mit geringer Bandbreite am Gridserver angemeldet sind, so muss diese Tatsache auch beim Scheduling berücksichtigt werden. Andernfalls kann es vorkommen, dass die langsame Verbindung zu solchen Knoten einen möglichen Geschwindigkeitsvorteil, den diese gegenüber mit höherer Bandbreite angebundenen Knoten haben, wieder zunichte macht.

Eine zukünftige Erweiterung von SimpleGrid muss also die verfügbare Bandbreite zu den angemeldeten Knoten überwachen und bei langsamen Verbindungen einen entsprechenden Strafterm auf die Kapazität dieser Gridnodes aufschlagen, um somit das Scheduling entsprechend zu steuern.

5.6 Fazit

Die in dieser Arbeit vorgestellte automatisierte Simulationsumgebung SimpleGrid erweist sich als ein nützliches Hilfsmittel für die Reduktion der Gesamtlaufzeit von Si-

mulationsaufträgen. Während man bei der herkömmlichen Methode der Simulationsdurchführung auf einem einzelnen Rechner mitunter sehr lange Rechenzeiten in Kauf nehmen muss, kann man mit Hilfe der Simulationsumgebung auf diese Zeit direkten, steuernden Einfluss nehmen. Durch die flexible Architektur von SimpleGrid lassen sich sehr leicht weitere Rechner der Simulationsumgebung hinzufügen und damit die zur Verfügung stehende, freie Kapazität erhöhen. Auf diese Weise können auch schwächere Rechner, die alleine sonst gar nicht für die Bearbeitung von Simulationsprojekten in Frage kämen, sinnbringend für die Berechnung von Simulationsläufen eingesetzt werden.

Durch die gute Skalierbarkeit der Simulationsumgebung ergibt sich ein weiterer Vorteil. Hat man eine ausreichende Anzahl von verfügbaren Rechnern, die man als Gridnodes an SimpleGrid teilnehmen lassen kann, so ist es nun auch möglich, die Anzahl der Wiederholungen eines Simulationslaufs heraufzusetzen. Wie in Abschnitt 2.1.1 dargelegt wurde, möchte man durch die möglichst häufige Durchführung derselben Simulation mit unterschiedlichen Anfangswerten für den Zufallsgenerator erreichen, dass sich mit den Simulationsergebnissen statistisch valide Aussagen treffen lassen. Bedingt durch die langen Rechenzeiten ist bisher allerdings nur ein sehr geringer Wert für die Anzahl der Wiederholungen möglich gewesen. Mit Hilfe von SimpleGrid und unter Verfügbarkeit einer entsprechend großen Menge an Rechnern kann man diesen Wert nun theoretisch beliebig erhöhen.

Zu Beginn der vorliegenden Arbeit wurden die theoretischen Grundlagen diskutiert, die für die Entwicklung einer automatisierten Simulationsumgebung benötigt werden. Zunächst wurde eine kurze Einführung in den Themenkomplex der diskreten Ereignissimulationen gegeben. SimpleSim als ein Programm zur Durchführung derartiger Simulationen für Computernetzwerke bildet das Fundament für die Arbeit der automatisierten Simulationsumgebung.

Anschließend wurde eine Reihe von Scheduling-Algorithmen vorgestellt. Diese haben die Aufgabe, eine Anzahl von Aufträgen so auf verfügbare Maschinen zu legen, dass dabei ein Optimalitätskriterium erfüllt wird. Für den vorliegenden Fall wurde angestrebt, die maximale Bearbeitungszeit eines Simulationsauftrags zu minimieren. Es wurden die Eigenschaften und Besonderheiten der Algorithmen Simulated Annealing, Branch And Bound, Greedy Scheduling und List Scheduling diskutiert.

Im dritten Teil dieser Arbeit wurde das grundlegende Konzept für eine automatisierte Simulationsumgebung erarbeitet. Dafür wurde zunächst eine Analyse der zuvor eingeführten Scheduling-Algorithmen durchgeführt. Die jeweiligen Stärken und Schwächen der einzelnen Verfahren wurden diskutiert und darauf aufbauend eine Strategie herausgearbeitet, nach der die Algorithmen sinnvoll für die Simulationsverteilung eingesetzt werden können.

Es folgte eine Untersuchung der Eigenschaften von Simulationsaufträgen und die Herleitung eines Verfahrens für die Verteilung solcher Aufträge auf freie Simulationsrechner. Dabei wurden eine Reihe von Problemen identifiziert, die sich hierbei stellen. Daraufhin wurden Lösungswege für die Behandlung dieser Probleme herausgearbeitet.

Für die Erhebung der Kapazitäten aller vorhandenen Simulationsrechner und die

Ermittlung der Komplexitäten aller zu verteilenden Simulationsläufe wurde jeweils ein eigenes Verfahren vorgestellt. Beide Verfahren bauen auf der direkten Vermessung der interessierenden Variablen auf. Während Rechnerkapazitäten mit Hilfe von Benchmarks gemessen werden, ermittelt man Simulationskomplexitäten durch das Ausführen der Repräsentanten aller Mengen komplexitätsäquivalenter Simulationen.

Im anschließenden Teil wurde die Implementierung von SimpleGrid vorgestellt. Nach einer Auflistung der Anforderungen, die in dieser Arbeit an SimpleGrid gestellt werden, folgte eine Beschreibung der Architektur der Simulationsumgebung. Die gesamte Arbeit einer Simulationsdurchführung wird unter den drei Modulen Gridserver, Gridnode und Client Tool aufgeteilt. Der konkrete Einsatz der vorgestellten Scheduling-Algorithmen und die Vorgehensweise dieser Module bei der Behandlung der diskutierten Schwierigkeiten bei der Simulationsverteilung wurde schließlich besprochen.

Zum Abschluss dieser Arbeit wurden einige Verbesserungs- und Erweiterungsmöglichkeiten vorgestellt, mit denen sich die Simulationsumgebung zukünftig noch nutzbringender gestalten lässt.

Anhang A

Berechnung der Untergrenze für die Branch And Bound-Methode

Im Folgenden soll erläutert werden, wie die Untergrenzen der Teilprobleme für die Branch And Bound-Methode berechnet werden. Einfache Beispiele sollen das Verfahren zusätzlich verdeutlichen.

Bei der Branch And Bound-Methode wird die Untergrenze eines Teilproblems benötigt, um zu entscheiden, ob der zu diesem Teilproblem gehörende Unterbaum des Suchbaums weiter durchsucht werden muss. Diese Entscheidung wird durch einen Vergleich der Untergrenze mit der aktuellen Obergrenze getroffen. Die Untergrenze muss dabei den besten Zielfunktionswert angeben, der in dem Unterbaum gefunden werden kann. Dieses Konzept wurde in Abschnitt 2.2.2.3 eingeführt.

Da die Ermittlung der bestmöglichen Lösung in einem Unterbaum wieder dem ursprünglichen Optimierungsproblem entspricht, müssen die für die Untergrenze geltenden Nebenbedingungen abgeschwächt werden. Dann können sie einfach berechnet werden, ohne dafür ein Optimierungsproblem lösen zu müssen. Man führt daher eine Relaxation des ursprünglichen Problems ein, die das Problem wesentlich einfacher werden lässt [9].

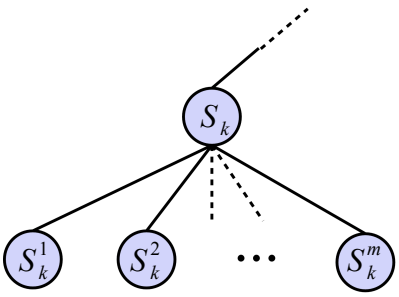
Für die Lösung der in dieser Arbeit behandelten Scheduling-Probleme besteht diese Relaxation darin, Preemption zuzulassen. Damit können einzelne Aufträge aufgeteilt und auf mehreren Maschinen bearbeitet werden.

Sei eine Menge J_i $(i = 1, \ldots, n)$ von zu verteilenden Jobs gegeben, deren Produktionsaufwand durch ξ_i festgelegt ist. Sei weiterhin eine Menge M_j $(j = 1, \ldots, m)$ von verfügbaren Maschinen mit den Kapazitäten κ_j gegeben. Die Produktionsdauer π_{ij} eines Auftrags J_i auf der Maschine M_j wird durch $\pi_{ij} = \xi_i \cdot \kappa_j$ berechnet. Die gesamte Produktionsdauer C lässt sich bei erlaubtem Preemption mit der folgenden Vorschrift berechnen:

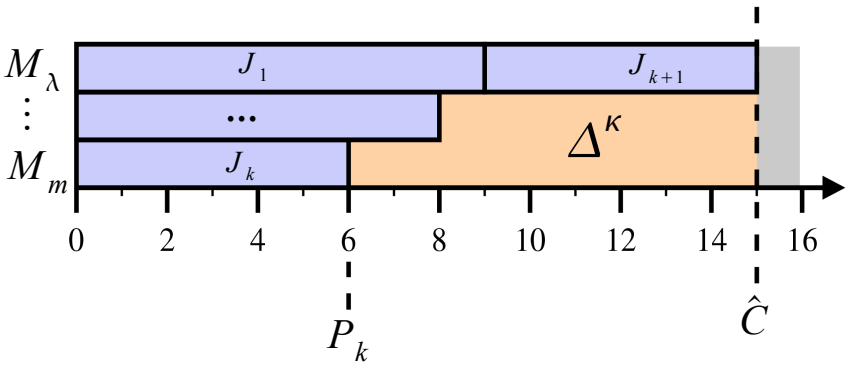
$$C = \frac{\sum_{i=0}^{n} \xi_i}{\sum_{j=0}^{m} \frac{1}{\kappa_j}} \tag{A.1}$$

Ein Beispiel soll dies verdeutlichen. Angenommen 4 Aufträge mit einem Bearbeitungsaufwand von jeweils $1,0$, $2,0$, $4,0$, und $5,0$ sollen auf 3 Maschinen mit den Kapazitäten $1,0$, $2,0$ und $3,0$ verteilt werden. Wenn alle Aufträge gleichmäßig verteilt werden können, so sind die Maschinen allesamt genau

$$\frac{(1,0 + 2,0 + 4,0 + 5,0)}{(1 + \frac{1}{2} + \frac{1}{3})} \approx 6,54$$



(a) Schema eines Knotens S_k des Suchbaums und seiner Kindknoten S_k^j



(b) Gantt-Diagramm für eine Teillösung

Abbildung A.1: Variablen für die Berechnung der Untergrenze einer Teillösung des Branch And Bound-Verfahrens

Zeiteinheiten beschäftigt.

Für die Berechnung der Untergrenze bei der Branch And Bound-Methode lässt sich dies wie folgt einsetzen. Es befinde sich dafür der Algorithmus in der Ebene k des Suchbaums. D. h. es wurden die ersten k Aufträge auf entsprechende Maschinen verteilt. Die aktuelle Teillösung an dieser Stelle sei mit S_k bezeichnet. Der nächste Schritt des Algorithmus besteht darin, den Auftrag J_{k+1} auf eine der m Maschinen zu legen. Es ergeben sich damit also m Kindknoten für die aktuelle Teillösung. Diese Kindknoten seien mit S_k^j $(j = 1, \ldots, m)$ bezeichnet. Abbildung A.1(a) illustriert dieses Schema.

Es soll nun betrachtet werden, wie die Untergrenze für einen dieser Kindknoten berechnet wird. Sei dafür mit λ der Index derjenigen Maschine bezeichnet, die in S_k^j als letztes ihre Arbeit beendet. Der Zeitpunkt, zu dem die Maschine M_λ fertig ist, sei mit $\hat{C}$ benannt. Abbildung A.1(b) stellt dies anhand eines Gantt-Diagramms für eine Teillösung dar.

Danach muss berechnet werden, wie der Zielfunktionswert des theoretischen Optimums für die Teillösung S_k^j ist, wenn die noch nicht zugewiesenen Aufträge J_l

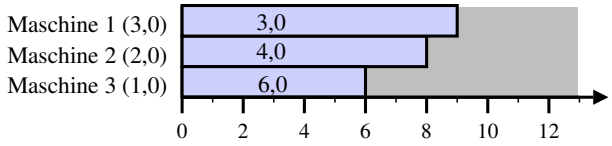


Abbildung A.2: Der Teilschedule S_3 des Branch And Bound-Algorithmus in der dritten Suchbaumebene

$(l = k + 2, \ldots, n)$ mit erlaubtem Preemption auf die Maschinen verteilt werden. Dieser Zielfunktionswert stellt die gesuchte Untergrenze dar. Sei dazu

$$\Lambda = \sum_{i=k+2}^{n} \xi_i$$

die Summe des Bearbeitungsaufwandes aller noch zuzuweisender Aufträge. Λ soll nun auf alle Maschinen verteilt werden. Dafür muss zunächst die freie Kapazität all derer Maschinen gefüllt werden, die vor Maschine M_λ ihre Arbeit beendet haben. Diese freie Kapazität sei mit Δ^κ bezeichnet. Sie ist in Abbildung A.1(b) orangefarben markiert. Δ^κ berechnet sich wie folgt:

$$\Delta^\kappa = \sum_{j=0;j\neq\lambda}^{M} \frac{\hat{C} - P_j}{\kappa_j}$$

Hierbei werden durch die P_j diejenigen Zeitpunkte beschrieben, zu denen die jeweiligen Maschinen M_j mit $j \neq \lambda$ in dem Teilschedule S_k^j ihre Arbeit beendet haben.

Mit diesen Informationen lässt sich die Untergrenze berechnen:

$$LB = \max\left(\hat{C}, \hat{C} + \frac{\Lambda - \Delta^\kappa}{\sum_{j=0}^{m} \frac{1}{\kappa_j}}\right)$$

Der Betrag $\Lambda - \Delta^\kappa$ ist der übrigbleibende Bearbeitungsaufwand, der nicht mehr auf die freie Kapazität Δ^κ verteilt werden kann. Hat diese Differenz einen negativen Wert, so bedeutet das, dass mehr freie Kapazität zur Verfügung steht, als noch Aufträge zu verteilen sind. In diesem Fall bildet $\hat{C}$ selbst die Untergrenze. Andernfalls wird mit Hilfe von Vorschrift (A.1) die Zeit berechnet, die alle Maschinen benötigen, um den restlichen Aufwand $\Lambda - \Delta^\kappa$ zu bearbeiten. Diese Zeit wird auf $\hat{C}$ aufgeschlagen und liefert so die Untergrenze.

Zwei Beispiele sollen dieses Vorgehen verdeutlichen. Es soll dafür das in Abschnitt 2.2.2.3 aufgeführte Beispiel als Grundlage herangezogen werden. Es wird zunächst angenommen, dass sich der Algorithmus in der dritten Ebene des Suchbaums befindet, der in Abbildung 2.7(b) in Abschnitt 2.2.2.3 dargestellt ist, d. h. also $k = 3$. Es wurde somit mit den ersten drei Aufträgen ein Teilschedule gebildet. Dieser wird in Abbildung A.2 gezeigt.

In der dritten Ebene des Suchbaums soll nun der Job mit einem Bearbeitungsaufwand von $2,0$ auf eine der Maschinen verteilt werden. Zuerst wird der Auftrag dafür auf

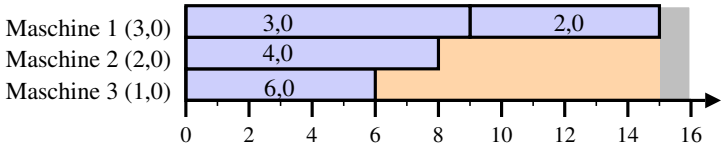


Abbildung A.3: Der nächste Auftrag mit einem Bearbeitungsaufwand von 2,0 wurde auf Maschine 1 gelegt. Der Teilschedule S_3^1 hat nun eine Bearbeitungsdauer von 15,0 Zeiteinheiten.

Maschine 1 gelegt. Der sich nun ergebende neue Teilschedule S_3^1 ist in Abbildung A.3 dargestellt. Für ihn ergibt sich die Bearbeitungsdauer $\hat{C} = 15,0$ Zeiteinheiten mit $\lambda = 1$. Da in diesem Beispiel nur noch ein weiterer Auftrag mit einem Aufwand von $1,0$ verteilt werden muss, ergibt sich für $\Lambda = 1,0$. Die freie Kapazität Δ^κ beträgt hier

$$\Delta^\kappa = \frac{15,0 - 8,0}{2,0} + \frac{15,0 - 6,0}{1,0} = 12,5$$

Diese freie Kapazität ist natürlich mehr als ausreichend, um den verbleibenden Bearbeitungsaufwand Λ zu verarbeiten. Daher ergibt sich hier als Untergrenze $LB = 15,0$.

Ein weiteres Beispiel ist in Abbildung A.4 dargestellt. Dieser Teilschedule entspricht dem ersten Schritt des Branch And Bound-Verfahrens in Abbildung 2.7(b) und damit S_1^3. Es soll hier die Untergrenze berechnet werden, die sich ergibt, wenn der erste Auftrag mit dem Aufwand 6,0 auf die schnellste Maschine gelegt wird. Es gilt dabei $\hat{C} = 6,0$.

Die freie Kapazität der übrigen Maschinen beträgt hier

$$\Delta^\kappa = \frac{6,0 - 0,0}{3,0} + \frac{6,0 - 0,0}{2,0} = 5,0$$

Die Summe der restlichen Aufträge ist $\Lambda = 1,0 + 2,0 + 3,0 + 4,0 = 10,0$. Man sieht hier wegen $\Lambda - \Delta^\kappa = 5,0 > 0$, dass im Gegensatz zum vorangegangenen Beispiel die freie Kapazität Δ^κ nicht reicht, um die restlichen Aufträge zu bearbeiten. Aus diesem Grund ergibt sich hier für die Untergrenze der Wert

$$LB = 6,0 + \frac{10,0 - 5,0}{1 + \frac{1}{2} + \frac{1}{3}} \approx 8,7$$

Dieser Wert bedeutet, dass unabhängig davon, wie die restlichen Aufträge auf die Maschinen verteilt werden, kein Schedule einen besseren Zielfunktionswert von etwa 8,7 Zeiteinheiten haben kann, wenn der erste Auftrag auf Maschine 3 gelegt wird.

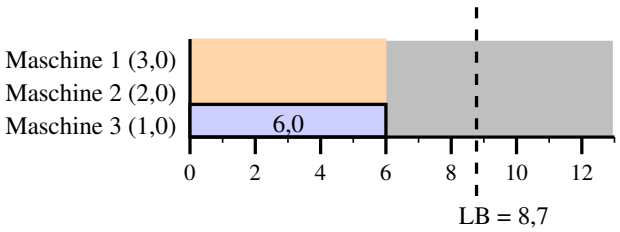


Abbildung A.4: Zweites Beispiel für die Berechnung der Untergrenze

Anhang B

Konfiguration der SimpleGrid-Module

Die einzelnen Module der Simulationsumgebung SimpleGrid Gridserver, Gridnode und Client Tool müssen vor Inbetriebnahme konfiguriert werden. Dies geschieht über einfache Konfigurationsdateien oder über Kommandozeilenparameter. In diesem Kapitel soll eine kurze Beschreibung aller Konfigurationsmöglichkeiten der einzelnen Module gegeben werden.

Die Konfigurationsdateien der SimpleGrid-Module haben die gleiche Syntax wie die Konfigurationsdateien für den SimpleSim Netzwerksimulator. Auch sie werden über die Java-Klasse `Properties` [19] eingelesen. Diese Dateien sind daher wie folgt aufgebaut. Kommentare werden grundsätzlich mit dem Nummernzeichen (#) eingeleitet. Konfigurationsoptionen werden in der Form

```
<Schlüssel>=<Wert>
```

übergeben. Der `Schlüssel` ist dabei einer der angebotenen Konfigurationsparameter und `Wert` die gewünschte Belegung.

Für die Kommandozeilenparameter gibt es zumeist eine kurze und eine lange Version. Welche Form man verwendet, ist dem Benutzer freigestellt. In den nachfolgenden Auflistungen der Kommandozeilenparameter werden beide Versionen mit Komma getrennt aufgeführt.

Einige der Parameter sind optional. Lässt man diese offen, so werden von SimpleGrid Standardwerte verwendet. Im Folgenden werden solche *Default*-Belegungen in eckigen Klammern neben den jeweiligen Parameternamen angegeben.

B.1 Gridserver

Für den Gridserver stehen die folgenden Parameter zur Verfügung:

ListenPort Legt den TCP-Port fest, an dem der Server eingehende Verbindungsversuche der Gridnodes entgegennimmt.

ApplicationLogFile [`server.log`] Der Server vermerkt alle wichtigen Ereignisse, die innerhalb der Simulationsumgebung stattfinden, in einer gesonderten Protokolldatei. Dazu gehören z. B. An- und Abmeldevorgänge der Gridnodes und die durchgeführten Simulationsaufträge. Mit diesem Parameter kann ein Name für diese Datei gewählt werden.

StorageDir Für seine Arbeit benötigt der Server ein eigenes Verzeichnis auf dem Hintergrundspeicher. Dieses dient ihm zum einen als Ablageort für temporäre Daten. Zum anderen erwartet er unter diesem Verzeichnis die für den Betrieb des Servers benötigten Daten. Hierzu zählen z. B. alle Dateien für die Durchführung der Benchmarks.

BenchmarkJar Legt den Namen der Benchmark JAR-Datei fest. Diese Datei enthält die Version des SimpleSim Netzwerksimulators, welche für die Durchführung der Benchmarks verwendet werden soll.

BenchmarkProperties Spezifiziert die Konfigurationsdatei für die Benchmarksimulation. Alle darin aufgeführten Szenariodateien, wie z. B. Knotenbewegungs- und Kommunikationsdateien, werden relativ zu dem Speicherort der Konfigurationsdatei gesucht. D. h., Pfadangaben für diese Dateien beziehen sich auf das Verzeichnis, in dem die Benchmarkkonfiguration zu finden ist. Der Parameter `SimulatorOutput` wird für Benchmarksimulationen automatisch auf `OFF` gesetzt, wodurch der Simulator alle Ausgaben unterdrückt.

SimulatorOutput [`ON`] Hiermit lassen sich die Ausgaben des Simulators wahlweise ausschalten oder aktivieren. Setzt man diesen Parameter auf `OFF`, so werden keinerlei Ausgabedateien während der Simulation erzeugt.

SimulationResultsDirectory Gibt das Verzeichnis an, welches als Ablageplatz für sämtliche Simulationsergebnisse verwendet werden soll. Für jeden bearbeiteten Simulationsauftrag wird darin ein eigenes Unterverzeichnis erstellt, das alle Ausgaben des Simulators aufnimmt. Dabei wird für jeden einzelnen Simulationslauf wiederum ein eigenes Unterverzeichnis angelegt, das alle zu diesem Lauf gehörenden Ergebnisdateien erhält.

UserLogFile [`simulation.log`] Während der Server einen Simulationsauftrag bearbeitet, führt er über alle dabei stattfindenden und den Benutzer interessierenden Ereignisse Buch. Die dabei entstehenden Meldungen schreibt er in eine Datei, dessen Name mit diesem Parameter festgelegt werden kann. So wird darin bspw. vermerkt, wann ein Simulationslauf welchem Gridnode zugewiesen wurde oder aus welchem Grund eine Simulation abgebrochen werden musste.

LoggingProperties Spezifiziert den Speicherort einer Konfigurationsdatei für das Java Logging-Subsystem. Mit dieser Datei lassen sich die Einstellungen des Pakets `java.util.logging` für den Server vornehmen. Wird die hier angegebene Datei nicht gefunden oder gar keine Datei angegeben, so wird die Standardeinstellung von Java verwendet.

MaxSimulationTaskRetry [`0`] Falls ein Simulationslauf auf einem Simulationsrechner abgebrochen wurde und diese Simulation wiederholt werden kann, gibt dieser Parameter die Anzahl der Versuche an, mit der maximal versucht werden soll, den Simulationslauf zu wiederholen.

BenchmarkInterval [10] Legt die Zeit fest, welche zwischen zwei aufeinanderfolgenden Benchmarks liegen soll. Dieser Wert wird in Minuten angegeben.

DoBenchmarks [yes] Kann entweder mit `yes` oder `no` belegt werden und legt fest, ob Benchmarks durchgeführt werden sollen oder nicht.

SimulationTimeoutMultiplicator [4] Legt den Multiplikator für den Timeout eines Simulationslaufs mit bekannter Komplexität fest. Der konkrete Timeout solcher Simulationen berechnet sich, indem man die geschätzte Rechenzeit für diese Simulation mit dem hier angegebenen Wert multipliziert.

MaxSimulationRunningTime [7200] Legt einen absoluten Timeout in Sekunden für Simulationsläufe mit noch unbekannter Komplexität fest.

Neben den Optionen für die Konfigurationsdatei des Gridservers existieren noch weitere Kommandozeilenparameter für das Server-Programm:

-c, --config Dient zur Angabe des Speicherortes der Konfigurationsdatei für den Server.

-l0, --logging_off Deaktiviert sämtliche Logging-Ausgaben des `java.util.logging`-Pakets.

-h, --help Gibt eine kurze Hilfeseite auf der Konsole aus.

--disable_console Deaktiviert die Kommandokonsole des Servers.

B.2 Gridnode

Gridnodes können mit den folgenden Konfigurationsoptionen eingestellt werden:

ListenPort Startet ein Gridnode den Netzwerksimulator für die Durchführung eines Simulationslaufs, so besteht die erste Aktion des Simulatorprozesses darin, sich mit dem Gridnode-Prozess lokal zu verbinden. Über diese Verbindung können die beiden Prozesse miteinander kommunizieren. Der Gridnode öffnet dafür vor dem Aufruf des Simulators einen TCP-Port, dessen Nummer mit `ListenPort` angegeben werden kann.

ServerAddress [localhost] Der Rechnername oder die IP-Adresse des Gridservers.

ServerPort Der TCP-Port, über den sich der Gridnode mit dem Server verbinden kann. Dieser Wert muss der Belegung von `ListenPort` in der Server-Konfigurationsdatei entsprechen.

StorageDir Wie der Gridserver benötigen auch die Gridnodes ein eigenes Verzeichnis auf dem Hintergrundspeicher, um temporäre Daten ablegen zu können. Mit dem Parameter `StorageDir` lässt sich ein solches Verzeichnis festlegen.

NodeInfoPropertiesFile [`nodeinfo.properties`] In dem mit `StorageDir` angegebenen Speicherort legt das Gridnode-Programm zusätzlich eine Datei an, in der alle knotenspezifischen Daten, wie z. B. der aktuelle Benchmarkwert, abgelegt werden. Mit diesem Parameter lässt sich ein bestimmter Name für diese Datei festlegen.

ReconnectInterval Sollte der Gridnode die Verbindung zum Server verlieren, so versucht er, sich erneut zu verbinden. Schlägt das fehl, probiert der Knoten immer wieder eine Neuanmeldung bis er entweder eine Verbindung zustande bringt oder das Gridnode-Programm beendet wird. Der Abstand zwischen den einzelnen Verbindungsversuchen wird mit dem Parameter `ReconnectInterval` angegeben.

PhysicalMemory [`100`] Gibt die Menge an verfügbarem physikalischen Hauptspeicher in Megabyte an, der für einen Simulationslauf verwendet werden kann. Dieser Wert darf nicht größer sein, als der tatsächlich vorhandene Speicher. Es kann jedoch ein kleinerer Wert angegeben werden. Der hier konfigurierte Wert wird später den beiden Parametern '`-Xms`' und '`-Xmx`' des Java-Interpreters übergeben, mit dem der Netzwerksimulator gestartet wird.

VmstatLocation [`/usr/bin/vmstat`] Die Erkennung von exzessivem Memory Swapping wird von der Java-Bibliothek nicht direkt unterstützt. Daher wird für diese Aufgabe auf ein externes Programm zurückgegriffen, mit dem der Zugriff auf den Auslagerungsspeicher überwacht werden kann. Auf Linux-Systemen wird dafür der Befehl `vmstat` verwendet. Mit dem Parameter `VmstatLocation` kann man den Speicherort dieses Befehls angeben.

LoggingProperties Dient zur Angabe des Speicherorts einer Logging-Konfigurationsdatei. Siehe dazu den gleichnamigen Parameter bei der Server-Konfiguration.

JavaBinLocation Der Netzwerksimulator wird für einen Simulationslauf als ein eigenständiger Java-Prozess gestartet. Der Gridnode verwendet dafür im Normalfall den Befehl `java` ohne weiteren Präfix. Es wird dadurch diejenige Java-Version verwendet, die im aktuellen Suchpfad für den Gridnode zu finden ist. Soll eine andere Version verwendet werden oder befindet sich der `java`-Befehl nicht im Suchpfad, so kann man mit `JavaBinLocation` einen alternativen Pfad bestimmen. Gibt man hier bspw. `/opt/jdk1.5.0_05/bin/` an, so wird der in diesem Verzeichnis zu findende Java-Interpreter benutzt.

Auch dem Gridnode-Programm können zur Konfiguration zusätzlich spezielle Kommandozeilenparameter übergeben werden:

-c, --config Dient zur Angabe des Speicherortes der Konfigurationsdatei für den Gridnode.

-l0, --logging_off Schaltet alle Logging-Nachrichten des `java.util.logging`-Pakets ab.

-s, --server Dient zur alternativen Angabe eines Servers, mit dem sich der Knoten verbinden soll. Der hier angegebene Wert überdeckt die in der Konfigurationsdatei spezifizierte Serveradresse.

B.3 Client Tool

Die Konfigurationsdatei des Programms enthält die nachfolgenden Parameter. Ein konkretes Anwendungsbeispiel für die Benutzung des Client-Programms findet sich in Anhang C.

ServerAddress Der Rechnername oder die IP-Adresse des Gridservers.

ServerPort Der TCP-Port, mit dem sich der Gridnode am Server verbinden kann. Dieser Wert muss der Belegung von `ListenPort` in der Server-Konfigurationsdatei entsprechen.

Die folgenden Kommandozeilenparameter sind für das Client Tool definiert:

-p, --property_files Mit diesem Parameter lässt sich eine Liste von Konfigurationsdateien für die einzelnen Simulationsläufe eines Simulationsauftrags übergeben. Jede dieser Dateien legt einen einzelnen Simulationslauf fest, der von SimpleGrid durchgeführt werden soll.

-c, --config_file Spezifiziert den Speicherort der Konfigurationsdatei für das Client Tool.

-s, --simulator Erwartet den Pfad zu der JAR-Datei, die diejenige Simulator-Version enthält, welche für die Simulation verwendet werden soll.

Anhang C

Anwendungsbeispiel für die Durchführung eines Simulationsprojekts

Im Folgenden soll beispielhaft dargestellt werden, wie die einzelnen SimpleGrid-Module zusammen eingesetzt werden können, um ein kleines Simulationsprojekt durchzuführen. Es wird dafür angenommen, dass die drei Module mit den in Anhang B aufgeführten Parametern korrekt konfiguriert worden sind. Weiterhin sollen die drei Module in jeweils einer ausführbaren JAR-Datei enthalten sein. Der Gridserver befindet sich dazu in der Datei `gridserver.jar`, das Gridnode-Programm in `gridnode.jar` und das Client Tool in `client.jar`. Es wird angenommen, dass sich der Java-Interpreter, der Java-Programme für die Version 1.5 ausführen kann, im aktuellen Suchpfad befindet.

Die Simulationsumgebung soll hier wie folgt aufgebaut werden. Der Gridserver wird auf dem Rechner mit dem Namen `diogenes` ausgeführt. Man startet dafür das Programm auf der Konsole dieses Rechners mit

```
$ java -jar gridserver.jar --config gridserver.properties
```

Der Server wartet daraufhin auf Verbindungswünsche der Gridnodes und Client-Programme.

Als nächstes sollen auf zwei weiteren Rechnern jeweils eine Instanz des Gridnode-Programms gestartet werden. Man ruft dafür auf diesen Rechnern das Programm wie folgt auf:

```
$ java -jar gridnode.jar --config gridnode.properties \
> --server diogenes
```

Die Knoten verbinden sich mit dem Server und warten auf die Zuteilung von Simulations- oder Benchmarkläufen. Man hat nun eine kleine Simulationsumgebung eingerichtet, die für die Bearbeitung von Simulationsaufträgen bereit ist.

Ein Simulationsprojekt kann folgendermaßen gestaltet werden. Es sollen hier beispielhaft vier verschiedene Simulationskonfigurationen behandelt werden. Jede dieser Konfigurationen soll fünfmal mit einem unterschiedlichen Random Seed wiederholt werden. Als Szenario wird ein mobiles Ad-Hoc-Netzwerk definiert. Die einzelnen Konfigurationen unterscheiden sich dabei zum einen in dem simulierten Routing-Protokoll und zum anderen in der Anzahl der vorhandenen Netzwerkknoten. Es sollen abwechselnd die Protokolle AODV [33] und DSR [20] zum Einsatz kommen. Weiterhin sollen einmal 50 Knoten und einmal 100 Netzwerkknoten simuliert werden.

```
MaximumTime=300.0
MovementFileName={move-50nodes.ssm|move-100nodes.ssm}
NodeClassName=\
  {org.pi4.simplesim.network.routing.aodv.test.SimpleNode |
   org.pi4.simplesim.network.routing.dsr.test.SimpleNode}
NumberOfNodes={50|100}
StatisticsFilename=statistics.sss
Terrain.xLength=5000.000000
Terrain.yLength=5000.000000
Terrain.zLength=10.000000
TraceFileFlushAlways=false
TraceFileName=trace.sst
EventInitiatorClassName=\
  {org.pi4.simplesim.network.routing.aodv.test.SimpleEventInitiator |
   org.pi4.simplesim.network.routing.dsr.test.SimpleEventInitiator}
EventInitiatorResourceFile=\
  {comm-50nodes-20links-30packets.ssa |
   comm-100nodes-20links-30packets.ssa}
RandomSeed={0|1|2|3|4}
```

Tabelle C.1: Schablone für die Konfigurationsdateien des Anwendungsbeispiels. Die in den geschweiften Klammern angegebenen Alternativen werden abwechselnd eingesetzt, so dass sich die Gesamtheit der gewünschten 20 Simulationsläufe ergibt.

sim-50nodes-AODV-0.properties
sim-50nodes-AODV-1.properties
sim-50nodes-AODV-2.properties
sim-50nodes-AODV-3.properties
sim-50nodes-AODV-4.properties
sim-100nodes-AODV-0.properties
sim-100nodes-AODV-1.properties
sim-100nodes-AODV-2.properties
sim-100nodes-AODV-3.properties
sim-100nodes-AODV-4.properties
sim-50nodes-DSR-0.properties
sim-50nodes-DSR-1.properties
sim-50nodes-DSR-2.properties
sim-50nodes-DSR-3.properties
sim-50nodes-DSR-4.properties
sim-100nodes-DSR-0.properties
sim-100nodes-DSR-1.properties
sim-100nodes-DSR-2.properties
sim-100nodes-DSR-3.properties
sim-100nodes-DSR-4.properties

Tabelle C.2: Konfigurationsdateien für das Anwendungsbeispiel

In Tabelle C.1 ist schablonenhaft dargestellt, wie die jeweiligen Konfigurationsdateien aufgebaut sind. Dabei sind die in geschweiften Klammern aufgeführten und mit einem senkrechten Strich getrennten Alternativen die Werte, die jeweils abwechselnd in die einzelnen Dateien eingesetzt werden müssen. So werden die beiden Parameter `NodeClassName` und `EventInitiatorClassName` jeweils mit `simplesim.network.routing.aodv.test.SimpleNode` und `simplesim.network.routing.aodv.test.SimpleEventInitiator` belegt, falls die Simulation für das AODV-Protokoll konfiguriert werden soll. Bei Einsatz von 50 Netzknoten werden `MovementFileName`, `NumberOfNodes` und `EventInitiatorResourceFile` jeweils mit `move-50nodes.ssm`, 50 und `comm-50nodes-20links-30packets.ssa` belegt. Die anderen Fälle werden analog erzeugt.

Es ergeben sich damit die in Tabelle C.2 aufgeführten 20 Dateien.

Jede der benötigten Dateien befindet sich für dieses Beispiel im selben Verzeichnis. Neben den Konfigurationsdateien gehören dazu die Bewegungsmusterdateien `move-50nodes.ssm` und `move-100nodes.ssm`, die Kommunikationsmusterdateien `comm-50nodes-20links-30packets.ssa` und `comm-100nodes-20links-30packets.ssa` und die ausführbare JAR-Datei `simplesim.jar` mit dem Simulator. Im Folgenden soll dieses Verzeichnis `~/simconfig/` heißen.

Ein solchermaßen definiertes Simulationsprojekt kann nun an den Gridserver geschickt werden. Man ruft dabei auf einem beliebigen Rechner, mit dem man sich mit dem Server verbinden kann, das Client-Programm auf:

```
$ java -jar client.jar --config_file client.properties \
> --property_files ~/simconfig/sim-* --simulator simplesim.jar
```

Damit wird das Client Tool beauftragt, alle Dateien im Verzeichnis `~/simconfig/` und die Simulatordatei `simplesim.jar` an den Server zur Bearbeitung zu schicken. Der Gridserver nimmt all diese Dateien entgegen und verteilt sie entsprechend auf die beiden Gridnodes.

Hat der Server schließlich seine Arbeit beendet, befinden sich alle Simulationsergebnisse in einem eigenen Unterverzeichnis in dem mit dem Server-Parameter `SimulationResultsDirectory` definierten Verzeichnis. Der Name dieses Verzeichnisses setzt sich aus dem Anmeldenamen des Benutzers und einem aktuellen Zeitstempel zusammen.

In dem Ergebnisverzeichnis findet man nun die folgenden Dateien und Unterverzeichnisse vor. Zum einen befinden sich darin alle für die Simulation notwendigen Hilfsdateien. Das sind alle Bewegungs- und Kommunikationsmusterdateien (`move-50nodes.ssm`, `move-100nodes.ssm`, `comm-50nodes-20links-30packets.ssa` und `comm-100nodes-20links-30packets.ssa`) und die Simulatordatei. Weiterhin wurde dort das Simulationsprotokoll in einer Datei mit dem Namen `simulation.log` abgelegt. Schließlich wurde in dem Ergebnisverzeichnis für jede Konfigurationsdatei ein gleichnamiges Unterverzeichnis erzeugt, in denen sich die Konfigurationsdatei selbst und alle Simulationsergebnisse (also die Dateien `trace.sst` und `statistics.sss`) befinden.

Literaturverzeichnis

[1] Aarts, Emile ; Korst, Jan: *Simulated Annealing and Boltzmann Machines.* New York, NY, USA : John Wiley & Sons, Inc., 1989

[2] Anderson, Oskar ; Popp, Werner ; Schaffranek, Manfred: *Schätzen und Testen.* 2. Ausgabe. Heidelberg : Springer-Verlag, 1997

[3] Beck, Kent: *Test Driven Development. By Example.* Reading, Massachusetts, USA : Addison Wesley, 2002

[4] Beck, Kent: *Extreme Programming Explained: Embrace Change.* 2nd edition. Reading, Massachusetts, USA : Addison-Wesley, 2004

[5] Berman, Fran (Hrsg.) ; Fox, Geoffrey (Hrsg.) ; Hey, Tony (Hrsg.): *Grid Computing – Making the Global Infrastructure a Reality.* 1st edition. New York, NY, USA : John Wiley & Sons, Inc, 2003

[6] Błażewicz, Jacek ; Ecker, Klaus H. ; Pesch, Erwin ; Schmidt, Günter ; Weglarz, Jan: *Scheduling Computer and Manufacturing Processes.* 2. Ausgabe. Heidelberg : Springer-Verlag, 2001

[7] Bouchenak, Sara: Making Java Applications Mobile or Persistent. In: *Proceedings of the 6th Conference on Object-Oriented Technologies and Systems (COOTS 2001)*, 2001, S. 159—172

[8] Brucker, Peter: *Scheduling Algorithms.* 4. Ausgabe. Heidelberg : Springer-Verlag, 2004

[9] Clausen, Jens: *Branch and Bound Algorithms - Principles And Examples.* `www.imada.sdu.dk/Courses/DM85/TSPtext.pdf`, 1999. – [Online; Stand 28. Juni 2006]

[10] Cormen, Thomas H. ; Leiserson, Charles E. ; Rivest, Ronald L.: *Introduction To Algorithms.* McGraw-Hill, 1999

[11] Coulouris, George ; Dollimore, Jean ; Kindberg, Tim: *Distributed systems: concepts and design.* 4th edition. Reading, Massachusetts, USA : Addison-Wesley, 2005

[12] Domschke, Wolfgang ; Drexl, Andreas: *Einführung in Operations Research.* 4. Ausgabe. Heidelberg : Springer-Verlag, 1998

[13] Fishman, George S.: *Discrete-Event Simulation.* 1st edition. Heidelberg : Springer, 2001

[14] Graham, R. L.: Bounds for certain multiprocessing anomalies. In: *Bell System Tech. Journal* 45 (1966), S. 1563–1581

[15] Graham, R. L. ; Lawler, E. L. ; Lenstra, J. K. ; Kan, A. H. G. R.: Optimization and approximation in deterministic sequencing and scheduling: A survey. In: *Annals of Discrete Mathematics* 5 (1979), S. 287–326

[16] Harchol-Balter, Mor: Task Assignment with Unknown Duration. In: *International Conference on Distributed Computing Systems*, 2000, S. 214–224

[17] Hitchens, Ron: *Java NIO*. 1st edition. O'Reilly, 2002

[18] *The Java Programming Language*. `http://java.sun.com`, 2006. – [Online; Stand 28. Juni 2006]

[19] *Java 2 Platform Standard Edition 5.0 API Specification. Sun Microsystems.* `http://java.sun.com/j2se/1.5.0/docs/api/index.html`, 2006. – [Online; Stand 28. Juni 2006]

[20] Johnson, David B. ; Maltz, David A.: Dynamic Source Routing in Ad Hoc Wireless Networks. In: Imielinski, Tomasz (Hrsg.) ; Korth, Hank (Hrsg.): *Mobile Computing* Bd. 353. Kluwer Academic Publishers, 1996, S. 153–181

[21] *JUnit - Java Unit Testing Framework*. `http://www.junit.org`, 2006. – [Online; Stand 28. Juni 2006]

[22] Kirkpatrick, S. ; Gelatt, C. D. ; Vecchi, M. P.: Optimization by Simulated Annealing. In: *Science* 220 (1983), S. 671–680

[23] Knuth, Donald E.: *The Art of Computer Programming - Seminumerical Algorithms*. Bd. 2. 3rd edition. Reading, Massachusetts, USA : Addison-Wesley, 1997

[24] Law, Averill M. ; Kelton, W. D.: *Simulation Modeling Analysis*. 3. Ausgabe. McGraw-Hill, 2000

[25] Liesegang, Dietfried G.: *Möglichkeiten zur wirkungsvollen Gestaltung von Branch and Bound-Verfahren dargestellt an ausgewählten Problemen der Reihenfolgeplanung*, Universität Köln, Diss., 1974

[26] Maekawa, Mamoru ; Oldehoeft, Arthur E. ; Oldehoeft, Rodney R.: *Operating systems: advanced concepts*. Benjamin/Cummings Publishing, 1987

[27] Metropolis, N. ; Rosenbluth, A. ; Rosenbluth, M. ; Teller, A. ; Teller, E.: Equation of state calculations by fast computing machines. In: *Journal of Chemical Physics* 21 (1953), S. 1087–1092

[28] Nemhauser, George L. ; Wolsey, Laurence A.: *Integer and combinatorial optimization*. New York, NY, USA : John Wiley & Sons, Inc., 1988

[29] *The ns-2 network simulator.* `http://www.isi.edu/nsnam/ns/`, 2006. – [Online; Stand 28. Juni 2006]

[30] *OTcl - Object Tool Command Language.* `http://otcl-tclcl.sourceforge.net/otcl/`, 2006. – [Online; Stand 28. Juni 2006]

[31] Papoulis, Athanasios: *Probability, random variables, and stochastic processes.* 3rd edition. McGraw-Hill, 1991

[32] Perkins, Charles E. (Hrsg.): *Ad hoc networking.* Reading, Massachusetts, USA : Addison-Wesley, 2004

[33] Perkins, Charles E. ; Royer, Elizabeth M.: Ad-Hoc On-Demand Distance Vector Routing. In: *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA).* New Orleans, LA, February 1999, S. 90–100

[34] Phipps, Geoffrey: Comparing observed bug and productivity rates for Java and C++. In: *Software – Practice and Experience* 29 (1999), Nr. 4, S. 345–358

[35] Pugh, William ; Spacco, Jaime: MPJava: High-Performance Message Passing in Java Using Java.nio. In: *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003)*, 2003, S. 323–339

[36] *The REAL network simulator.* `http://www.cs.cornell.edu/skeshav/real/overview.html`, 2006. – [Online; Stand 28. Juni 2006]

[37] Rubow, Gerhard: *Die Lösung von Reihenfolgeproblemen aus dem Bereich der Maschinenbelegungsplanung mit Hilfe des Branch and Bound Algorithmus*, Freie Universität Berlin, Diss., 1969

[38] Schader, Martin ; Schmidt-Thieme, Lars: *Java – Eine Einführung.* 4. Ausgabe. Heidelberg : Springer-Verlag, 2003

[39] Sgall, J.: On-line scheduling – a survey. In: Fiat, A. (Hrsg.) ; Woeginger, G. (Hrsg.): *On-Line Algorithms.* Springer-Verlag, Berlin, 1997 (Lecture Notes in Computer Science)

[40] Stallings, William: *Operating systems: internals and design principles.* 4th edition. Prentice Hall, 2001

[41] Stroustrup, Bjarne: *The C++ programming language.* Reading, Massachusetts, USA : Addison-Wesley, 1991

[42] Tanenbaum, Andrew S.: *Modern operating systems.* 2nd edition. Prentice Hall, 2001

[43] Vidal, René V. V. (Hrsg.): *Applied simulated annealing.* Heidelberg : Springer-Verlag, 1993

[44] The VINT project (Hrsg.): *The ns manual*. The VINT project, 2003. `http://www.isi.edu/nsnam/ns/ns-documentation.html`. – [Online; Stand 28. Juni 2006]

[45] Wikipedia: *Cancer Research Project — Wikipedia, Die freie Enzyklopädie*. `http://de.wikipedia.org/w/index.php?title=Cancer_Research_Project&oldid=17759475`. – [Online; Stand 28. Juni 2006]

[46] Wikipedia: *Common Object Request Broker Architecture — Wikipedia, Die freie Enzyklopädie*. `http://de.wikipedia.org/w/index.php?title=Common_Object_Request_Broker_Architecture&oldid=18278621`. – [Online; Stand 29. Juni 2006]

[47] Wikipedia: *Folding@home — Wikipedia, Die freie Enzyklopädie*. `http://de.wikipedia.org/w/index.php?title=Folding%40home&oldid=18227806`. – [Online; Stand 28. Juni 2006]

[48] Wikipedia: *Greedy-Algorithmus — Wikipedia, Die freie Enzyklopädie*. `http://de.wikipedia.org/w/index.php?title=Greedy-Algorithmus&oldid=17686598`. – [Online; Stand 28. Juni 2006]

[49] Wikipedia: *Quality of Service — Wikipedia, Die freie Enzyklopädie*. `http://de.wikipedia.org/w/index.php?title=Quality_of_Service&oldid=18157395`. – [Online; Stand 27. Juni 2006]

[50] Wikipedia: *SETI@home — Wikipedia, Die freie Enzyklopädie*. `http://de.wikipedia.org/w/index.php?title=SETI%40home&oldid=18078760`. – [Online; Stand 24. Juni 2006]

[51] *XML - Extensible Markup Language*. `http://www.w3.org/XML/`, 2006. – [Online; Stand 28. Juni 2006]