Magisterarbeit

Titel der Magisterarbeit

Metaheuristiken für ein stochastisches flexibles Flow-Shop-Problem

Verfasser
Stefan Katzensteiner, Bakk. rer. soc. oec.

angestrebter akademischer Grad
Magister der Sozial- und Wirtschaftswissenschaften
(Mag. rer. soc. oec.)

Wien, 2009

Studienkennzahl lt. Studienblatt:	A 066 926
Studienrichtung lt. Studienblatt:	Magisterstudium Wirtschaftsinformatik
Betreuer:	Univ. Ass. Dipl.-Ing. Dr. Christian Almeder

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Symbol- und Abkürzungsverzeichnis

Abkürzungen

AOC	Active Object Class
FIFO	First-In-First-Out
PPS	Produktionsplanung - und steuerung
RNVS	Reduced Variable Neighborhood Search
SA	Simulated Annealing
VND	Variable Neighborhood Descent
VNS	Variable Neighborhood Search

Symbole

B_{ij}	Beginnzeit von Auftrag j auf Maschine i
C_{ij}	Fertigstellungszeit von Auftrag j auf Maschine i
d_j	Spätester Fertigstellungstermin von Auftrag j
L_j	Abweichung d. Auftrages j vom Fertigstellungstermin
p_{ij}	Bearbeitungszeit von Auftrag j auf Maschine i
r_j	Bereitstellungstermin von Auftrag j
T_j	Verspätung von Auftrag j
U_j	Einheitsstrafe von Auftrag j
w_j	Gewicht von Auftrag j
f	Kostenfunktion für eine Lösung
m	Anzahl der Maschinen
N	Nachbarschaftsfunktion
n	Anzahl der Aufträge
X	Menge der zulässigen Lösungen

Kapitel 1

Einleitung

Ablaufplanung (eng. scheduling) im Allgemeinen beschäftigt sich mit der Zuordnung von bestimmten Aufgaben auf vorhandene Ressourcen um diese möglichst effizient durchzuführen. Reihenfolgeplanung, eine konkrete Form der Ablaufplanung, ordnet Aufträge auf einer Menge von Maschinen an und versucht eine optimale Anordnung im Sinne eines vorgegebenen Bewertungskriteriums zu finden.

Eine spezielle Modellform der Reihenfolgeplanung, die mehrstufige Produktionsprozesse mit mehreren Produktionseinheiten pro Stufe betrachtet, nennt sich flexibles Flow-Shop-Problem. Diese Form wird zum Beispiel in der Textilindustrie, in Automobilfabriken und in der Halbleiterindustrie angewendet.

In solch komplexen Produktionsumgebungen kann mit an Sicherheit grenzender Wahrscheinlichkeit davon ausgegangen werden, dass Produktionsstillstände durch Ausfälle und variierende Bearbeitungszeiten der Aufträge durch sich verändernde Rahmenbedingungen im System (z.B. Abnützung von Maschinen) entstehen. Dies führt zu Unsicherheit (Stochastik) in der Planung, die speziell beachtet und auch in der Planung und Entwicklung von Lösungsalgorithmen berücksichtigt werden muss. Bereits unter Außerachtlassen der erhöhten computationalen Anforderungen von stochastischen Einflüssen sind Probleminstanzen mit geringer Größe (meist weniger als 100 Aufträge), dieses kombinatorischen Optimierungsproblems, durch exakte Verfahren nicht mehr in annehmbarer Zeit lösbar.

Aus diesem Grund werden Lösungsverfahren, so genannte Heuristiken, ver-

wendet, die in kurzer Zeit gute Lösungen liefern ohne jedoch deren Optimalität garantieren zu können. Allgemeine Vorgehensweisen, deren Arbeitsschemata generisch auf verschiedene Probleme angewendet werden können, nennt man Metaheuristiken. Zwei Metaheuristiken, die in dieser Arbeit angewendet werden, sind Variable Neighborhood Search (VNS) und Simulated Annealing (SA).

Im theoretischen Teil der vorliegenden Arbeit (Kapitel 2 und 3) wird ein Überblick über (i) das Feld der Maschinenbelegungsprobleme mit Fokus auf flexible Flow-Shop-Probleme, (ii) in der Literatur verwendete Heuristiken und (iii) Besonderheiten bei stochastischen Problemen dieser Art gegeben. Anschließend wird im Kapitel 4 die praktische Umsetzung ausgewählter Verfahren beschrieben und in Kapitel 5 deren Lösungsqualtität durch die Auswertung künstlicher Testinstanzen überprüft. Das abschließende Kapitel 6 enthält eine Zusammenfassung und einen Ausblick auf mögliche zukünftige Entwicklungen.

Kapitel 2

Reihenfolgeplanung

In Uetz (2002) wird Reihenfolgeplanung formal beschrieben als:

> *... scheduling*[1] *concerns the allocation of limited resources to tasks over time. It is a decision-making process whose goal is the optimization of one or more objectives subject to some constraints.*

Die Schlüsselwörter sind hier *resources* (Ressourcen), *tasks* (Aufgaben), *time* (Zeit), *decision-making* (fällen von Entscheidungen), *objectives* (Zielfunktionen) und *constraints* (Nebenbedingungen). Übersetzt und zusammenfassend bedeutet dies begrenzte Ressourcen, zum Beispiel Maschinen oder Arbeiter, über einen gewissen Zeitraum bestimmten Aufgaben zuzuordnen und dabei einen Prozess zu formen der es ermöglicht ihn in Hinsicht auf ein oder mehrere bestimmte Ziele, unter Einhaltung vorhander Nebenbedingungen, möglichst gut zu betreiben.

Abschnitt 2.1 stellt die Reihenfolgeplanung im Kontext der Produktionsplanung, ein Teil des Gesamtplanungsprozesses des Unternehmens, dar. Standardprobleme der Reihenfolgeplanung werden danach in Abschnitt 2.2 vorgestellt. Über die Behandlung von Unsicherheit in Reihenfolgeplanungsproblemen gibt Abschnitt 2.3 Auskunft und Abschnitt 2.4 stellt das in dieser Arbeit behandelte Modell vor.

[1] Scheduling bezieht sich in diesem Kontex nicht auf die allgemeine Ablaufplanung sondern auf die speziellere From der Reihenfolgeplanung.

2.1 Produktionsplanung und -steuerung

Dieser Abschnitt beschreibt die Produktionsplanung und -steuerung im Unternehmen und führt zur Stellung der Reihenfolgeplanung innerhalb der Produktionsplanung hin. Er orientiert sich dabei an Schuh (2006) und Domschke u. a. (1993)[2].

Ziel eines jeden in der Produktion von Waren tätigen Unternehmens ist es das Produktionssystem so effizient wie möglich zu gestalten. Die dafür eingesetzten Konzepte werden Produktionsplanung - und steuerung (PPS) genannt. Abbildung 2.1 zeigt die Teilgebiete aus denen die Produktionsplanung und -steuerung aufgebaut ist.

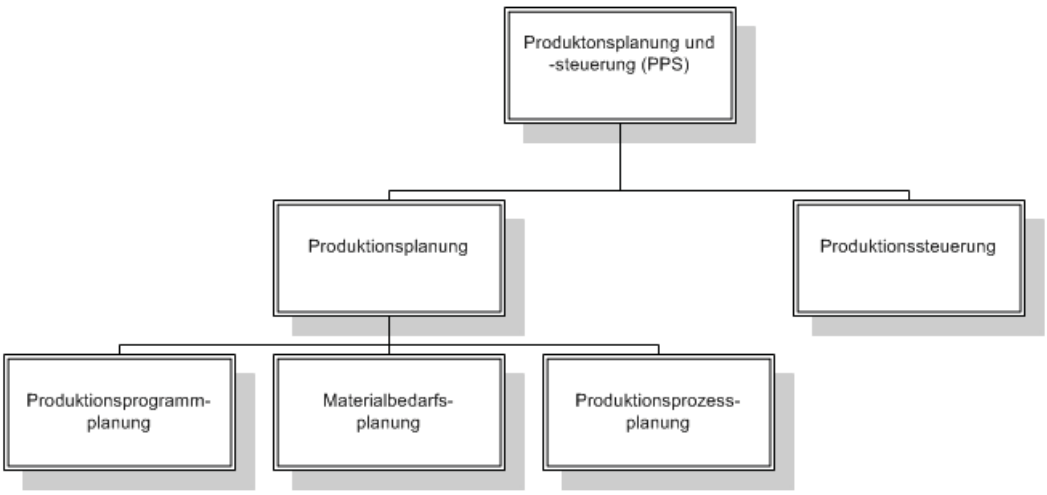

Abbildung 2.1: Aufbau der Produktionsplanung und -steuerung

In der *Produktionsplanung* wird ein mittel- bis kurzfristiger Plan für die Produktion erstellt. Sie besteht aus den Teilgebieten Produktionsprogrammplanung, Materialbedarfsplanung und Produktionsprozessplanung.

Die *Produktionssteuerung* setzt die Aufträge mit den in der Produktionsplanung ermittelten Vorgaben um. Dies beinhaltet die Zuteilung von auftragsbezogenen Arbeitsprozessen zu Produktionsfaktoren, das Überwachen der ordnungsge-

[2]siehe auch: Produktionsplanung und -steuerung, Wikipedia, 2009, http://de.wikipedia.org/wiki/Produktionsplanung_und_-steuerung, Online abgerufen am 16. März 2009

mäßen Umsetzung nach den Vorgaben der Produktionsrichtlinien des Unternehmens und die Rückmeldung von operativen Daten an die Produktionsplanung.

Die *Produktionsprogrammplanung* versucht lang-, mittel- und kurzfristige Vorgaben für die Produktion zu erstellen. Langfristig ist das Ziel vorteilhafte Marktsegmente aufzuspüren und zu erschliessen. Mittelfristig werden Produktgruppen definiert für die kurzfristig die herzustellenden Mengen bestimmt werden.

Wurde festgelegt welche Menge von welchem Produkt hergestellt werden soll, ist es notwendig die Menge an benötigten Rohstoffen und Zwischenprodukten zu bestimmen. Dies geschieht in der *Materialbedarfsplanung*. Eine weitere Aufgabe dieses Funktionsbereiches ist es dafür zu sorgen, dass die benötigten Materialien zeitgerecht vorhanden sind.

Die *Produktionsprozessplanung* (Abbildung 2.2) bestimmt die konkrete Umsetzung der benötigten Produktionsmenge auf den Maschinen. Zu diesem Zweck muss zur Minimierung der Gesamtkosten bestimmt werden, ob bei Aufträgen, die aus mehreren Stücken bestehen, alle Stücke auf einmal produziert werden oder ob sie in Lose aufgeteilt werden (*Losgrößenplanung*). Danach werden mittels der *Durchlaufterminierung* für jeden Auftrag Zeitfenster bestimmt in denen der jeweilige Auftrag abzuarbeiten ist. In der *Kapazitätsterminierung* wird abgeklärt ob für den bis zu diesem Zeitpunkt erstellten Plan die notwendigen Produktionskapazitäten vorhanden sind.

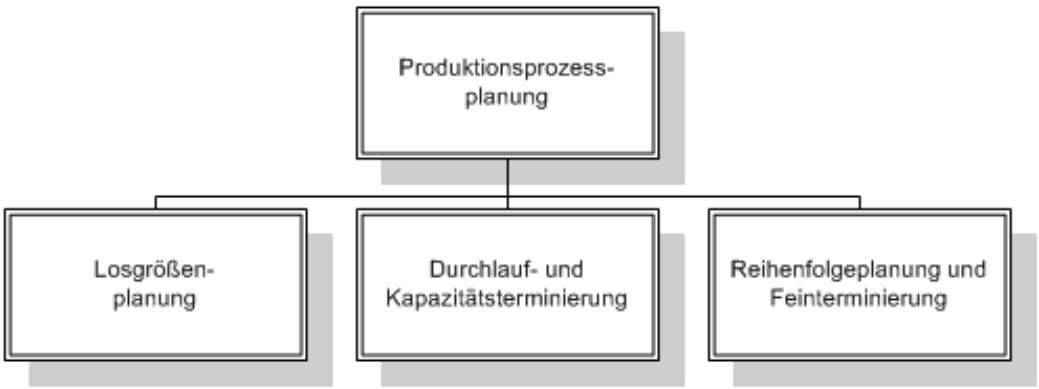

Abbildung 2.2: Aufbau der Produktionsprozessplanung

Wurden die Losgrößenplanung und die Durchlauf- und Kapazitätsterminierung durchgeführt, werden in der *Reihenfolgeplanung/Feinterminierung* (eng. sche-

duling) Maschinenbelegungspläne erstellt, die für jeden Auftrag festlegen in welchem Zeitraum er auf welcher Maschine bearbeitet wird. Meist lassen die vorgegebenen Daten aus den darüber gelagerten Planungsschichten einen gewissen Freiraum, wodurch es möglich ist mehrere Maschinenbelegungspläne zu erstellen, die die Vorgaben erfüllen. Es kann zum Beispiel darauf geachtet werden, dass der letztendlich eingesetzte Maschinenbelegungsplan möglichst robust gegenüber Störungen im Produktionsprozess ist, oder, dass er die entstehenden Lagerhaltungskosten gering hält.

2.2 Standardprobleme

Im Abschnitt 2.1 wurde die Reihenfolgeplanung im Kontext der Produktionsplanung des Unternehmens vorgestellt. Ausgehend von Basisdaten, die für die zu bearbeitenden Aufträge zur Verfügung stehen, ist es zunächst die Aufgabe die Reihenfolge, in der die Aufträge auf den Maschinen bearbeitet werden, zu bestimmen. Aus dieser Reihenfolge wird der Belegungsplan ermittelt, der die genauen Zeiten beinhaltet, zu denen die Bearbeitung der Aufträge auf den Maschinen beginnt und endet. Aus diesem Belegungsplan kann dann ein Maß errechnet werden, das angibt wie gut die verwendeten Auftragsfolgen im Sinne eines angestrebten Optimierungszieles sind. Die Erläuterungen in diesem Abschnitt und seinen Unterabschnitten basieren, wenn nicht explizit anders angegeben, auf Pinedo (2008).

Allgemein besteht ein Modell der Reihenfolgeplanung aus n Aufträgen $A_j (j = 1, ..., n)$ und m Maschinen M_i $(i = 1, ..., m)$. Meist ist vorgegeben, in welcher Reihenfolge ein Auftrag die verschiedenen Maschinen besuchen muss (Maschinenfolge). Als Auftragsfolge wird die Reihenfolge bezeichnet, in der die Maschine die Aufträge bearbeitet. Diese ist Gegenstand der Planung. Des weiteren hat jeder Auftrag eine Reihe von Attributen. Die wichtigsten sind die Bearbeitungszeit (eng. processing time) p_{ij} von Auftrag j auf Maschine i, der Bereitstellungstermin (eng. release date) r_j der angibt ab wann an Auftrag j gearbeitet werden kann, der späteste Fertigstellungstermin (eng. due date) d_j der angibt bis wann der Auftrag j spätestens fertig gestellt sein muss und das Gewicht (eng. weight) w_j des Auftrages j das seine relative Bedeutung gegenüber den anderen Aufträgen angibt.

Es wird zwischen statischen und dynamischen Problemen unterschieden, wobei aber die Verwendung in der Literatur nicht eindeutig ist. In Hartl und Dörner (2006) wird zum Beispiel folgende Definition verwendet:

> *Stehen alle Aufträge zum Zeitpunkt $a_j = 0$ [Anm.: r_j] zur Bearbeitung bereit, bezeichnet man das Problem als statisch. Dem gegenüber nennt man Probleme mit verschiedenen Auftragsfreigabezeitpunkten dynamisch.*

In anderen Werken (vgl. z.B. Nahmias (2000)) werden Probleme, mit zu Beginn bekannter und konstanter Menge an Aufträgen, statische Probleme und jene, wo im Laufe der Zeit neue Aufträge hinzukommen, dynamische genannt. In dieser Arbeit werden Probleme, in denen alle Bereitstellungstermine gleich Null sind, als 'Probleme ohne Bereitstellungstermine', jene Probleme, wo die Menge der Aufträge vor der Optimierung bekannt und konstant ist, als statische und jene Probleme, in denen laufend Aufträge hinzukommen, als dynamische bezeichnet.

In Pinedo (2008) wird eine Variation der bekannten $\alpha|\beta|\gamma$-Klassifikation für Modelle der Reihenfolgeplanung vorgestellt. Das α-Feld beschreibt die Maschinenumgebung und ihre Eigenschaften, das β-Feld gibt Auskunft über Verarbeitungsvorschriften und Nebenbedingeungen und das γ-Feld bestimmt die zu optimierende Zielfunktion.

Die folgenden Unterabschnitte beschreiben anhand der im letzten Absatz vorgestellten Klassifikation einige der bekanntesten Modelle. Abschnitt 2.2.1 beschreibt die bekanntesten Maschinenumgebungen (α-Feld), Abschnitt 2.2.2 führt eine Auswahl der Verarbeitungsvorschriften und Nebenbedingungen an (β-Feld) und Abschnitt 2.2.3 gibt einen Überblick über die möglichen Optimierungsziele (γ-Feld).

2.2.1 Modellformen

Durch Festlegen der Anzahl der Maschinen und/oder dem Muster mit dem die Aufträge die Maschinen besuchen, ergeben sich spezielle Modellformen die es durch ihre besonderen Eigenschaften zulassen, effiziente Lösungsstrategien für sie zu entwerfen.

2.2.1.1 Einmaschinenmodell ($\alpha = 1$)

Ist im betrachteten Modell nur eine Maschine vorhanden, so bezeichnet man dies als Einmaschinenmodell. Für solche Modelle ist im α-Feld das Zeichen „1“ angeführt. Es ist das einfachste aller Modelle und eine Spezialisierung von allen komplexeren Modellen. Trotz der Einfachheit dieses Modells wurde es in vielen Publikationen untersucht, da Erkenntnisse die hier gewonnen werden, einen Ausgangspunkt für Überlegungen bei komplexeren Modellen bilden.

Anhand des folgenden Beispiels und dieses Modells soll dem Leser die grundsätzliche Vorgehensweise bei der Reihenfolgeplanung dargelegt werden. Es sei ein Einmaschinenmodell mit 3 Aufträgen (A_1, A_2, A_3) gegeben. Die Bearbeitungszeiten[3] der Aufträge sind $p_1 = 3$, $p_2 = 1$ und $p_3 = 2$. Eine mögliche Auftragsfolge ist A_2 vor A_1 vor A_3. Daraus ergibt sich folgender Belegungsplan: Auftrag 2 ist von Zeitpunkt 0 bis Zeitpunkt 1, Auftrag 1 von Zeitpunkt 1 bis Zeitpunkt 4 und Auftrag 3 von Zeitpunkt 4 bis Zeitpunkt 6 der Maschine zugeteilt. Abbildung 2.3 zeigt diesen Belegungsplan in einem Gantt-Diagramm[4].

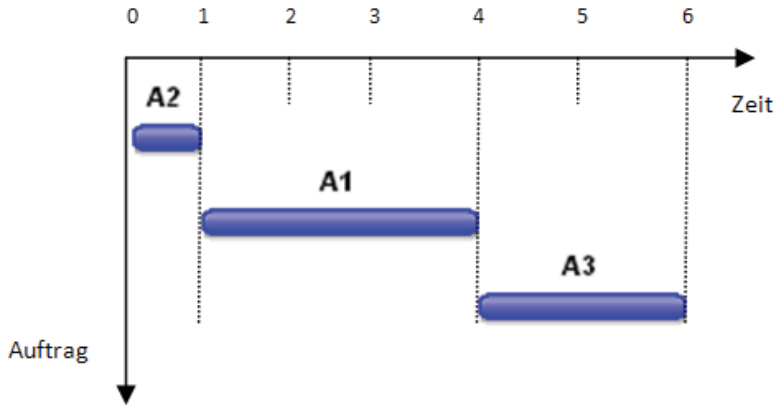

Abbildung 2.3: Gantt-Diagramm Einmaschinenbeispiel

Der Zeitpunkt zu dem mit der Bearbeitung des Auftrages begonnen wird ist die Beginnzeit (B_{ij}) und jener Zeitpunkt an dem die Bearbeitung des Auftrages beendet wird ist die Fertigstellungszeit (C_{ij}). In diesem Beispiel ist $B_1 = 1$, $B_2 = 0$, $B_3 = 4$ und $C_1 = 4$, $C_2 = 1$, $C_3 = 6$.

[3]Der Index für die Maschine kann bei einem Einmaschinenmodell vernachlässigt werden.

[4]siehe Gantt-Diagramm, Wikipedia, 2009, http://de.wikipedia.org/wiki/Gantt-Diagramm, Online abgerufen am 09. April 2009

Auf diesen Belegungsplan wird nun eine Funktion angewendet die diesem eine Bewertungsmaß zuordnet, die so genannte Zielfunktion. Auf mögliche Zielfunktionen wird in Abschnitt 2.2.3 eingegangen.

Allgemeines Ziel jedes Reihenfolgeplanungsproblems ist es, jene Auftragsfolge (im vorherigen Beispiel A_2 vor A_1 vor A_3) zu finden die den Belegungsplan (Beginn- und Fertigstellungszeiten der Aufträge auf der Maschine) mit dem besten Wert des Bewertungsmaßes liefert. Hier treten zwei Probleme auf. Zum Einen ist das Auffinden der besten Auftragsfolge ein kombinatorisches Problem, wodurch meist schon bei geringer Anzahl an Maschinen und Aufträgen eine sehr grosse Anzahl an möglichen Auftragsfolgen existiert. Zum anderen ist es nur bei einfachen Problemen möglich, eine Formel für die Berechnung von Belegungsplänen bei fixierten Auftragsfolgen anzugeben.

Größe des Lösungsraumes

Für ein Einmaschinenproblem besteht der Lösungsraum der möglichen Auftragsfolgen aus allen Permutationen der Reihenfolge der Aufträge. Eine Permutation π wird dargestellt durch $\pi = (\pi_1, ..., \pi_n)$ wobei n die Anzahl der Aufträge ist. Bei 3 Aufträgen (A_j, j = 1 ... 3) wären mögliche (aber nicht alle) Permutationen (A_1, A_2, A_3) oder, wie im Beispiel weiter oben, (A_2, A_1, A_3). Laut Bleymüller u. a. (2000) ist die Anzahl der möglichen Anordnungen n!. Für das obige Beispiel mit 3 Aufträgen ergeben sich also 3! = 6 mögliche Permutationen. Nimmt man zum Beispiel 20 Aufträge erhält man 20! = 2.432.902.008.176.640.000 mögliche Permutationen, in Worten ca. 2 Trillionen Permutationen. Es ist somit leicht nachvollziehbar, dass es bei auch nur mäßig großer Anzahl von Aufträgen nicht mehr möglich ist alle im Zuge eines Lösungsverfahrens zu überprüfen.

Berechnung des Belegungsplanes

Sobald die Auftragsfolgen für alle Maschinen im Modell fixiert sind, gilt es daraus einen Belegungsplan zu errechnen. Für alle Modelle, bei denen es sinnvoll ist, wird ein Algorithmus angegeben wie der Belegungsplan berechnet werden kann. Dabei wird die Notation und Vorgehensweise in Anlehnung an Weng (2000), Jungwattanakit u. a. (2009), Tavakkoli-Moghaddam und Safaei (2007) und Sawik (1993) gewählt.

Im folgenden wird nur die Berechnung der Fertigstellungszeiten angeführt, da die Beginnzeiten (B_j) in den hier behandelten Modellen ohne Preemption einfach

mit $B_j = C_j - p_j$ errechnet werden können. Gegeben sei eine einzelne Maschine, n Aufträge, die Bearbeitungszeiten $p_j (j = 1, \ldots, n)$ der Aufträge j auf der Maschine und die Auftragsfolge π welche eine Permutation der Aufträge darstellt in der die Maschine mit den Aufträgen belegt werden soll. $\pi(j)$ gibt an an welcher Stelle der Auftrag j von der Maschine bearbeitet wird. $\pi^{-1}(p)(p = 1, \ldots, n)$ ist die Umkehrfunktion dieser Permutation und gibt für einen gegebenen Platz p in der Permutation an, welcher Auftrag j diesen Platz hat. Weiters bezeichnet C_j die Fertigstellungszeit von Auftrag j und $C_{\pi^{-1}(p)}$ die Fertigstellungszeit des p-ten Auftrages auf der Maschine. Als Startbedingung wird für einen fiktiven Auftrag mit Index 0 eine Fertigstellungszeit $C_0 = 0$ und eine Position in der Permutation $\pi^{-1}(0) = 0$ definiert. Die Berechnung der Fertigstellungszeiten wird in Abbildung 2.4 gezeigt.

```
1 C_0 = 0;
2 π^{-1}(0) = 0;
3 FOR p = 1 TO n {
4    C_{π^{-1}(p)} = p_{π^{-1}(p)} + C_{π^{-1}(p-1)};
5 }
```

Abbildung 2.4: Berechnung Fertigstellungszeiten Einmaschinenmodell

2.2.1.2 Modell mit parallelen Maschinen ($\alpha = Pm/Qm/Rm$)

Beim Modell mit parallelen Maschinen handelt es sich um eine der möglichen Erweiterung des Einmaschinenmodells. Mehrere Maschinen arbeiten parallel und es gilt erstens die Aufträge best möglich zwischen den Maschinen aufzuteilen und dann zweitens, unter den jeweils einer Maschine zugeordneten Aufträgen, die beste Auftragsfolge zu finden. Dieses Modell kann in die folgenden Untervarianten unterteilt werden.

1. Parallele idente Maschinen (α=Pm). Hier sind für alle Aufträge die Bearbeitungszeiten auf allen Maschinen ident.

2. Parallele Maschinen mit unterschiedlichen Geschwindigkeiten (α=Qm). In dieser Variante hat jeder Auftrag eine Grundverarbeitungszeit p_j die durch die Geschwindigkeit der Maschine s_i geteilt wird.

3. Parallele unterschiedliche Maschinen (α-Feld=Rm). Es gibt für jeden Auftrag j auf jeder Maschine i eine eigene Verarbeitungszeit p_{ij}.

Die Modellgruppe mit parallelen Maschinen wird oft benutzt, um mehrstufige Probleme zu zerlegen. Uetz (2002) merkt an, dass mit „makespan“ (siehe Abschnitt 2.2.3) als Zielfunktion und wenn keine Vorrangbedingungen zwischen den Jobs gegeben sind, nur die Maschinenzuordnung relevant ist, da die Anordnung der Aufträge auf den einzelnen Maschinen im Falle von makespan als Zielfunktion keine Auswirkung hat.

Größe des Lösungsraumes

Im Fall von 2 parallelen Maschinen lassen sich die verschieden Zuteilungsmöglichkeiten durch eine binär Zeichenfolge der Länge n darstellen, wobei n die Anzahl der Aufträge ist, 0 für die Zuteilung zur ersten Maschine und 1 für die Zuteilung zur zweiten Maschine steht. Es gibt somit 2^n mögliche Zuteilungen. Die Anzahl der Maschine 1 zugeordneten Aufträge entspricht der Anzahl der 0-Elemente in der binären Zeichenfolge. Die Anzahl der binären Zeichenfolgen mit der gleichen Anzahl k an 0-Elementen lässt sich durch den Binomialkoeffizienten bestimmen. Sind k Aufträge der ersten Maschine zugeordnet, dann gibt es k! mögliche Auftragsfolgen für diese Maschine und (n-k)! mögliche Auftragsfolgen für die zweite Maschine. Die Grösse des Lösungsraumes für ein Modell mit 2 parallelen Maschinen gibt Formel 2.1 an.

$$\sum_{k=0}^{n} \binom{n}{k} (n-k)!(k)! \tag{2.1}$$

Für m parallele Maschinen gestaltet es sich wesentlich schwieriger eine geschlossene Form für die genaue Anzahl der Möglichkeiten im Lösungsraum zu finden. Diese zu finden ist jedoch auch nicht notwendig, da für die Abschätzung der Komplexität des Problems eine untere Schranke genügt. Bei n Aufträgen und m Maschinen existieren m^n Möglichkeiten die Aufträge auf die Maschinen aufzuteilen, was eine untere Schranke für die Anzahl der Möglichkeiten ist, da unter Beachtung der Auftragsfolgenpermutationen auf den einzelnen Maschinen sicher noch mehr und nicht weniger Möglichkeiten entstehen. Somit wächst die Anzahl der Möglichkeiten schon allein bei der Aufteilung auf die Maschinen exponentiell mit der Anzahl der Aufträge, wodurch sich auch dieses Problem als schwieriges

kombinatorisches Problem darstellt.

Berechnung des Belegungsplanes

Gegeben sind m Maschinen und n Auftäge und jeder Auftrag muss von genau einer Maschine bearbeitet werden. Job $j(j = 1, \ldots, n)$ hat auf Maschine $i(i = 1, \ldots, m)$ die Bearbeitungszeit p_{ij}. Des weiteren ist für jede Maschine die Reihenfolge π_i, in der die Maschine i die Auftäge abarbeiten muss, gegeben. $a_i = |\pi_i|$ ist die Anzahl der Aufträge die auf Maschine i bearbeitet werden. Die Berechnung der Fertigstellungszeiten $C_{i,\pi^{-1}(p)}$ des p-ten Auftrages, der auf Maschine i bearbeitet wird, ist in Abbildung 2.5 abgebildet.

```
1 C_{i,0} = 0 ∀i;
2 π^{-1}(0) = 0;
3 FOR i = 1 TO m {
4   FOR p = 1 TO a_i {
5     C_{i,π^{-1}(p)} = p_{i,π^{-1}(p)} + C_{i,π^{-1}(p-1)};
6   }
7 }
```

Abbildung 2.5: Berechnung Fertigstellungszeiten Modell mit parallelen Maschinen

Im Grunde genommen werden die einzelnen Maschinen, nachdem bekannt ist welche Aufträge ihnen zugeordnet worden sind, wie Einmaschinenprobleme behandelt.

2.2.1.3 Flow-Shop-Modell ($\alpha = Fm$)

Ein weiteres Modell, Flow-Shop genannt, erhält man wenn die Maschinenfolgen bei allen Aufträgen gleich sind (die Aufträge müssen die Maschinen in der selben Reihenfolge besuchen). Für solche Modelle ist im α-Feld das Zeichen „Fm“ angeführt. Eine Spezialisierung dieses Modells ist der Permutations-Flow-Shop bei welchem die Auftragsfolgen von allen Maschinen gleich sind.

Größe des Lösungsraumes

Im Falle eines Permutations-Flow-Shop sind die Auftragsfolgen auf allen Maschinen gleich wodurch nur eine Auftragsfolge wie beim Einmaschinenproblem durchsucht werden muss. Es gibt somit n! mögliche Auftragsfolgen. Bei einem generellen Flow-Shop gibt es für die erste Maschine n! Möglichkeiten, für jede dieser

Möglichkeiten n! Möglichkeiten auf der 2. Maschine usw. Bei m Maschinen und n Aufträgen ergeben sich somit $(n!)^m$ Möglichkeiten. Diese Funktion wächst exponentiell mit der Anzahl der Maschinen und beim Einmaschinenproblem wurde gezeigt, dass die Fakulätsfunktion auch sehr schnell extrem große Werte annimmt. Somit ist auch dieses Problem als ein schwieriges anzusehen.

Berechnung des Belegungsplanes

Gegeben sind n Aufträge die alle auf m Maschinen bearbeitet werden müssen. Jeder Auftrag $j \in J = 1, 2, ..., n$ läuft durch die Maschinen $i(i = 1, \ldots, m)$ in dieser Reihenfolge und benötigt eine Bearbeitungszeit p_{ij} auf jeder Machine. $\pi(j)$ gibt an als wievielter Auftrag der Auftrag j von der Maschine bearbeitet wird. $\pi^{-1}(p)$ ist die Umkehrfunktion dieser Permutation und gibt für einen gegebenen Platz p in der Permutation an, welcher Auftrag j diesen Platz hat. C_{ij} bezeichnet den Fertigstellungstermin von Auftrag j auf Maschine i und $C_{i\pi^{-1}(p)}$ den Fertigstellungstermin des p-ten Auftrages auf der Maschine i. Die Berechnung der Fertigstellungszeiten ist in Abbildung 2.6 abgebildet.

```
1 C_{i,0} = 0 ∀i;
2 π^{-1}(0) = 0;
3 C_{0,j} = 0 ∀j;
4 FOR i = 1 TO m {
5   FOR p = 1 TO n {
6     C_{i,π^{-1}(p)} = p_{i,π^{-1}(p)} + max(C_{i-1,π^{-1}(p)}, C_{i,π^{-1}(p-1)});
7   }
8 }
```

Abbildung 2.6: Berechnung Fertigstellungszeiten Flow-Shop

In $C_{i,\pi^{-1}(p)} = p_{i,\pi^{-1}(p)} + max(C_{i-1,\pi^{-1}(p)}, C_{i,\pi^{-1}(p-1)})$ ist $C_{i,\pi^{-1}(p)}$ der Fertigstellungstermin des p-ten Auftrages der auf Maschine i bearbeitet wird. Dieser berechnet sich aus der Bearbeitungszeit $p_{i,\pi^{-1}(p)}$ des p-ten Auftrages auf Maschine i plus dem größeren der beiden Werte $C_{i-1,\pi^{-1}(p)}$ und $C_{i,\pi^{-1}(p-1)}$. $C_{i-1,\pi^{-1}(p)}$ ist der Fertigstellungstermin des betrachteten Auftrages auf der vorherigen Maschine und $C_{i,\pi^{-1}(p-1)}$ ist der Fertigstellungstermin des Auftrages der direkt vor dem betrachteten auf derselben Maschine kommt.

2.2.1.4 Flexibles Flow-Shop-Modell ($\alpha = FFc$)

Das zuvor beschriebene Modell mit parallelen Maschinen und das Flow-Shop-Modell sind Spezialisierungen des so genannten flexiblen Flow-Shop-Modells. Für solche Modelle ist im α-Feld das Zeichen „FFc“ angeführt. In diesem Modell gibt es zumindest zwei Stufen die von allen Aufträgen in der gleichen Reihenfolge durchlaufen werden müssen. Zumindest eine der Stufen besteht aus mindestens zwei parallelen Maschinen. In der Literatur ist das flexible Flow-Shop-Modell auch unter den Namen „flow shop with multiple processors (FSMP)“ (Santos u. a., 1996), „hybrid flow shop“ (Allaoui und Artiba, 2006) und „flexible flow line“ (Tavakkoli-Moghaddam u. a., 2009) bekannt.

Größe des Lösungsraumes

Das flexible Flow-Shop-Modell kann für die Berechnung der Größe des Lösungsraumes als mehrere parallele Maschinenprobleme angesehen werden, die seriell angeordnet sind. Es müssen somit die Größen der Lösungsräume der einzelen Stufen miteinander multipliziert werden. In Abschnitt 2.2.1.2 wurde für das Modell mit parallelen Maschinen erläutert, dass dieses Modell als schwierig anzusehen ist. Es kann dadurch geschlossen werden, dass das flexible Flow-Shop-Modell ein noch schwierigeres Problem darstellt und einen immens großen Lösungsraum schon bei kleiner Anzahl von Aufträgen, Stufen und Maschinen besitzt.

Berechnung des Belegungsplanes

Gegeben sei ein **flexibles Flow-Shop-Modell mit unlimitierter Puffergröße** zwischen den Stufen. Der Index der Stufen sei $s (s = 1, \ldots, k)$, wobei k die Anzahl der Stufen ist. Jede Stufe hat m_s Maschinen mit Indizes i. Es gibt n Aufträge mit Indizes j mit den Bearbeitungszeiten p_{ij}^s auf Maschine i der Stufe s. Für jede Maschine i in Stufe s gibt es eine Permutation $\pi_{i^s}(j)$ die angibt an welcher Stelle Auftrag j von Maschine i in Stufe s bearbeitet wird. $a_i^s = |\pi_{i^s}|$ ist die Anzahl der Aufträge die auf Maschine i in Stufe s bearbeitet werden. $\pi_{i^s}^{-1}(p)$ ist die Umkehrfunktion dieser Permutation und gibt für einen gegebenen Platz p auf Maschine i der Stufe s in der Permutation an, welcher Auftrag j diesen Platz hat. Für jeden Auftrag j auf jeder Stufe s gibt es eine Fertigstellungszeit C_j^s. Es wird definiert, dass $\pi_{i^s}^{-1}(0) = 0 \forall i, s$ ist und, dass $C_0^s = 0 \forall s$ ist. Die Fertigstellungszeit für den p-ten Auftrag in π auf einer Maschine i in Stufe s in einem

flexiblen Flow-Shop-Modell mit unbeschränkten Zwischenpuffern kann dann wie folgt berechnet werden:

$$C^s_{i,\pi_{is}^{-1}(p)} = p^s_{i,\pi_{is}^{-1}(p)} + max\left\{C^{s-1}_{i,\pi_{is}^{-1}(p)}, C^s_{i,\pi_{is}^{-1}(p-1)}\right\} \tag{2.2}$$

In Gleichung 2.2 ist $C^s_{i,\pi_{is}^{-1}(p)}$ (die Fertigstellungszeit des p-ten Auftrages auf Maschine i in Stufe s) gleich der Bearbeitungszeit $p^s_{i,\pi_{is}^{-1}(p)}$ des Auftrages plus dem Maximum aus den Werten $C^{s-1}_{i,\pi_{is}^{-1}(p)}$ (die Fertigstellungszeit des Auftrages auf der vorherigen Stufe) und $C^s_{i,\pi_{is}^{-1}(p-1)}$ (die Fertigstellungszeit des Auftrages der auf der selben Maschine direkt vor dem p-ten Auftrag kommt). Der Algorithmus zur Berechnung der Fertigstellungszeiten ist in Abbildung 2.7 abgebildet.

```
1 π_is^-1(0) = 0 ∀ i, s;
2 C_0^s = 0 ∀ s;
3 FOR s = 1 TO k {
4   FOR i = 1 TO m_s {
5     FOR p = 1 TO a_i^s {
6       C^s_{i,π_is^-1(p)} = p^s_{i,π_is^-1(p)} + max{C^{s-1}_{i,π_is^-1(p)}, C^s_{i,π_is^-1(p-1)}}
7     }
8   }
9 }
```

Abbildung 2.7: Berechnung Fertigstellungszeiten flexibler Flow-Shop ohne Puffer

Im Fall der einfachen Modelle (zum Beispiel Einmaschinenprobleme, flexible Flow-Shop-Probleme ohne Puffer) läßt sich die Berechnung des Belegungsplanes relativ einfach in einer beliebigen Programmiersprache (zum Beispiel Java) umsetzten. Das später in dieser Arbeit behandelte Modell ist ein **flexibles Flow-Shop-Modell mit begrenztem gemeinsamen Puffer vor jeder Stufe** (siehe Abbildung 2.8). Im Gegensatz zu den bisher behandelten Problemen ist die Berechnung des Belegungsplanes für dieses spezielle Modelle sehr viel aufwendiger.

Bei diesen komplizierteren Modellen greift man auf die Unterstützung von Simulationsprogrammen wie AnyLogic[5] zurück. Solche Produkte bieten Unterstützung in der Erstellung von teils hoch-komplexen Simulationen, die es dem Forscher erlauben, sich auf das Erstellen des Modells zu konzentrieren und nicht auf die Implementierung einer Ablaufsimulation. Eine Umsetzung der Berechnung

[5]AnyLogic, XJ Technologies Company Ltd., http://www.xjtek.com/

des Belegungsplanes in AnyLogic für das in dieser Arbeit behandelte Modell ist in Abschnitt 4.1 beschrieben.

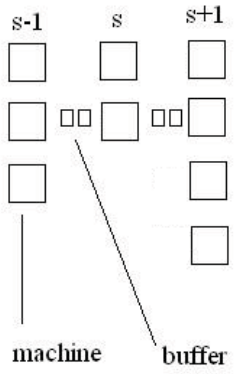

Abbildung 2.8: Flexibler Flow-Shop mit gemeinsamen Puffer vor Stufe

2.2.1.5 Job-Shop-Modell ($\alpha = Jm$)

In einem Job-Shop-Modell (α-Feld „Jm“) kann jeder Auftrag die Maschinen in einer anderen Reihenfolge besuchen. Es kann also jeder Auftrag eine andere Maschinenfolge haben. Jeder Auftrag muss jedoch auf jeder Maschine bearbeitet werden. Es wird unterschieden zwischen der Variante, in der ein bestimmter Auftrag auf ein und derselben Maschine höchstens ein Mal bearbeiten werden darf und der Variante, in der manche Aufträge mehrmals dieselbe Maschine besuchen müssen. Die zweite Variante wird Wiedereintrittssystem (eng. reentrant system) genannt und hat im β-Feld zusätzlich den Eintrag „rcrc“ für Rezirkulation (eng. recirculation) vermerkt.

2.2.1.6 Flexibles Job-Shop-Modell ($\alpha = FJc$)

Ein flexibles Job-Shop-Modell ist die Generalisierung des Job-Shop-Modells und des Modells mit parallelen Maschinen. Für solche Modelle ist im α-Feld das Zeichen „FJc“ angeführt. Mindestens eine der m Maschinen im Job-Shop-Modell wird durch mindestens 2 parallele Maschinen ersetzt und diese Gruppe an Maschinen wird Arbeitsstätte (eng. work center) genannt. Jeder Auftrag kann eine eigene Maschinenfolge haben. Hat ein Auftrag eine Arbeitsstätte in seiner Maschinenfolge dann wird genau eine der Maschine der Arbeitsstätte ausgewählt.

2.2.1.7 Open-Shop-Modell ($\alpha = Om$)

In einem Open-Shop-Modell (α-Feld „Om“) gibt es m Maschinen, es müssen jedoch nicht alle Aufträge alle m Maschinen besuchen. Es gibt keine Einschränkung für die Maschinenfolgen der Aufträge und Aufträge können unterschiedliche Maschinenfolgen haben.

Auf eine weitere Betrachtung der Job-Shop- und Open-Shop-Modelle wird in dieser Arbeit nicht näher eingegangen, da es den Rahmen sprengen würde.

2.2.2 Verarbeitungsvorschriften und Nebenbedingungen

Wie in Pinedo (2008) beschrieben kann es zusätzlich zu der Anzahl der Maschinen und/oder dem Muster mit dem die Aufträge die Maschinen besuchen (α-Feld) noch Beschränkungen geben denen das Maschinenbelegungsproblem unterworfen ist (β-Feld). Mehrere dieser Beschränkungen können in diesem Feld eingetragen sein.

- **Bereitstellungstermine ($\beta = r_j$)**
  Wenn Aufträge in einem Modell Bereitstellungstermine (eng. release dates) besitzen dann wird im β-Feld “r_j” angegeben und es kann nicht vor dem jeweiligen Bereitstellungstermin r_j begonnen werden den Auftrag j zu bearbeiten.

- **Unterbrechung ($\beta = prmp$)**
  Der Eintrag “prmp” im β-Feld zeigt die Möglichkeit der Unterbrechung der Bearbeitung eines Auftrages an (eng. preemptions). Es ist möglich den Auftrag während der Bearbeitung von der Maschine zu entfernen und ihn später, möglicherweise auf einer anderen gleichwertigen Maschine, fertig zustellen.

- **Vorrangbedingungen ($\beta = prec$)**
  Vorrangbedingungen (eng. precedence constraints) werden im β-Feld durch “prec” angezeigt und bedeuten, dass manche Aufträge fertig gestellt sein müssen bevor mit anderen Aufträgen begonnen werden darf.

- **Reihenfolge-abhängige Rüstzeiten ($\beta = s_{jk}$)**
  Gibt es reihenfolge-abhängige Rüstzeiten (eng. sequence dependent setup

time) wird dem β-Feld der Eintrag "s_{jk}" hinzugefügt. s_{jk} ist dabei die Rüstzeit welche entsteht wenn Auftrag k nach Auftrag j kommt. Wenn die Rüstzeiten zwischen Auftrag j und k abhängig von der Maschine sind, kommt zusätlich der Index i hinzu, also s_{ijk}.

- **Auftragsgruppen ($\beta = fmls$)**
  Können die Aufträge eines Modells unterschiedlichen Auftragsgruppen (eng. job families) zugeordnet werden, steht im β-Feld der Eintrag "fmls". Zwischen Aufträgen der selben Gruppe entstehen keine Rüstzeiten, zwischen Aufträgen unterschiedlicher Gruppen g und h entstehen jedoch Rüstzeiten s_{gh}.

- **Stapelverarbeitung ($\beta = batch(b)$)**
  In manchen Modellen können Maschinen bis zu b Aufträge auf einmal bearbeiten. Dies wird Stapelverarbeitung (eng. batch processing) genannt und im β-Feld mit "batch(b)" bezeichnet. Die Verarbeitungszeiten innerhalb eines Stapels können unterschiedlich sein, der gesamte Stapel ist jedoch erst fertig gestellt, wenn der Auftrag mit der längsten Bearbeitungszeit fertig gestellt ist.

- **Maschinen-Verfügbarkeitseinschränkung ($\beta = brkdwn$)**
  Kann es im Modell zu Maschinenausfällen kommen, bezeichnet man dies als Maschinen-Verfügbarkeitseinschränkung (eng. machine availability constraint) und fügt dem β-Feld "brkdwn" (für eng. breakdowns) hinzu.

- **Maschinen-Eignungseinschränkung ($\beta = M_j$)**
  Die Maschinen-Eignungseinschränkung (eng. machine eligibility restriction) (β-Feld "M_j") kann bei Modellen mit parallelen Maschinen auftauchen. Diese Einschränkung besagt, dass nicht alle Maschinen fähig sind den Auftrag j zu bearbeiten. Die Menge M_j beinhaltet die Maschinen die den Auftrag j bearbeiten können.

- **Permutation ($\beta = prmu$)**
  Wie in Abschnitt 2.2.1.3 beschrieben, ist eine Spezialform eines Flow-Shop-Modells jene, in welchem alle Aufträge die selbe Auftragsfolge auf allen Maschinen haben. Diese Form wird Permutations-Flow-Shop genannt und

im β-Feld mit "prmu" gekennzeichnet. Für flexible Flow-Shop-Modelle bedeutet dieser Eintrag, dass die Auftragsfolge jeder Stufe gleich ist.

- **Blockieren von Maschinen ($\beta = block$)**
  Kommt es zum Blockieren von Maschinen (eng. blocking) wird im β-Feld "block" verzeichnet. Zur Blockade von Maschinen kann es kommen, wenn vor den Maschinen Puffer mit begrenzter Größe vorgesehen sind. Ist der Puffer vor einer Maschine, die das Ziel eines soeben fertig gestellten Auftrages ist, voll, kann dieser Auftrag die Maschine in der er fertiggestellt wurde nicht verlassen und blockiert sie.

2.2.3 Optimierungsziele

Nach Pinedo (2008) ist der letzte Eintrag der in Abschnitt 2.2 vorgestellten Klassifikation das γ-Feld. Dieses Feld gibt die Zielfunktion, also das Bewertungsmaß für einen Belegungsplan, an. Viele der Zielfunktionen für Standardprobleme sind Funktionen der Fertigstellungszeiten ($C_j = max_i(C_{ij})$) der Aufträge, die im Belegungsplan verzeichnet sind, und der spätesten Fertigstellungstermine (d_j) der Aufträge.

Für Zielfunktionen, die von spätesten Fertigstellungsterminen abhängen, müssen einige zusätzliche Begriffe definiert werden. Die Abweichung vom spätesten Fertigstellungstermin (eng. latness) ist definiert als $L_j = C_j - d_j$. Die Verspätung (eng. tardiness) ist definiert als $T_j = max(C_j - d_j, 0) = max(L_j, 0)$. Die Einheitsstrafe (eng. unit penalty) eines Auftrages j ist $U_j = 1$ wenn $C_j > d_j$, sonst 0.

Beispiele für Zielfunktionen sind:

- **Makespan ($\gamma = C_{max}$)**
  Die Fertigstellungszeit des letzten fertig gestellten Auftrages wird im Englischen als Makespan bezeichnet und ist definiert als $max_j(C_j)$. Die Minimierung des makespan sorgt im Normalfall für eine gute Auslastung der Maschinen. Im γ-Feld wird für diese Zielfunktion "C_{max}" eingetragen.

- **Maximale Abweichung vom spätesten Fertigstellungstermin ($\gamma = L_{max}$)**
  Die Maximale Abweichung vom spätesten Fertigstellungstermin (eng. maximum lateness) (γ-Feld "L_{max}") berechnet sich aus $max_j(L_j)$.

- **Gesamte gewichtete Fertigstellungszeit ($\gamma = \sum w_j C_j$)**
  Die gesamte gewichtete Fertigstellungszeit (eng. total weighted completion time) wird im γ-Feld mit "$\sum w_j C_j$" notiert. Sie ist ein Indikator für die Lagerhaltungskosten welche vom zugrunde liegenden Belegungsplan verursacht werden.

- **Gesamte gewichtete Abweichung vom spätesten Fertigstellungstermin ($\gamma = \sum w_j L_j$)**
  Die gesamte gewichtete Abweichung vom spätesten Fertigstellungstermin (eng. total weighted lateness) (γ-Feld "$\sum w_j L_j$") bildet den Versuch Belegungspläne zu erstellen, in denen die Aufträge möglichst nahe an ihren spätesten Fertigstellungsterminen vollendet werden. Diese Zielfunktion spielt zum Beispiel bei der Just-In-Time Produktion eine Rolle.

- **Gesamte gewichtete Verspätung ($\gamma = \sum w_j T_j$)**
  Die gesamte gewichtete Verspätung (eng. total weighted tardiness) (γ-Feld "$\sum w_j T_j$") versucht die Verspätungen der Aufträge zu minimieren.

- **Gewichtete Anzahl der verspäteten Aufträge ($\gamma = \sum w_j U_j$)**
  Die gewichtete Anzahl der verspäteten Aufträge (eng. weighted number of tardy jobs) (γ-Feld "$\sum w_j U_j$") versucht die Anzahl der verspäteten Aufträge zu minimieren.

2.3 Stochastische Modelle

Die im letzten Abschnitt verwendete Klassifiktation nach Pinedo (2008) beschäftigt sich nur mit statischen, deterministischen Problemen in der Klassifikation nach Vieira u. a. (2003). Statisch bedeudet in diesem Zusammenhang, dass die Menge der zu bearbeitenden Aufträge endlich ist. Deterministisch bedeutet, dass alle Informationen über das Problem im Voraus bekannt sind.

Demgegenüber sind viele reale Produktionsumgebungen dynamisch in dem Sinn, dass laufend Aufträge hinzukommen und in der Planung berücksichtigt werden müssen. Des Weiteren sind Produktionssysteme komplexe, stochastische Systeme, in denen es viele Arten von Störungen wie zum Beispiel Maschinenausfälle, Bearbeitungszeitverzögerungen, Prioritätsbestellungen, Qualitätsprobleme und Materialknappheit gibt.

Trotzdem muss versucht werden einen Belegungsplan zu erstellen, der für jeden Auftrag auf jeder in Frage kommenden Maschine Beginn- und Fertigstellungszeiten beinhaltet. Nachdem dieser Belegungsplan erstellt ist, beginnt die Produktion. Durch unerwartete Ereignisse kommt es zu Abweichungen von diesem Plan, wobei kleinere Abweichungen meist toleriert werden. Bei großen Abweichungen muss eine Neuplanung (eng. rescheduling) erfolgen, da die Ereignisse in der Produktionsstätte im schlimmsten Fall Ergebnisse genau entgegegesetzt zum Ziel den ursprünglichen Belegungsplanes verursachen.

Um auch Probleme, die nicht in die Klassifikation nach Pinedo (2008) passen, einordnen zu können, wurde in Vieira u. a. (2003) eine Klassifikation für Reihenfolgeplanungsprobleme mit Neuplanung vorgestellt. Diese Klassifikation teilt die Probleme anhand der vier folgenden Unterscheidungsmerkmalen ein.

- Neuplanungsumgebungen (eng. rescheduling environments)
- Neuplanungstrategien (eng. rescheduling strategies)
- Neuplanungspolitiken (eng. rescheduling policies)
- Neuplanungmethoden (eng. rescheduling methods)

Neuplanungsumgebungen identifizieren die Menge der Aufträge die beplant werden muss. Neuplanungstrategien beschreiben, ob Belegunspläne generiert werden oder nicht. Neuplanungspolitiken spezifizieren, wann Neuplanung gemacht werden soll. Neuplanungmethoden beschreiben, wie Belegungspläne generiert und angepasst werden.

Abbildung 2.9 aus Vieira u. a. (2003) stellt das gesamte Framework dar. Die folgenden beiden Abschnitte beschreiben die Neuplanungsumgebungen und die Neuplanungstrategien. Auf Neuplanungspolitiken und Neuplanungmethoden wird

in dieser Arbeit nicht näher eingegangen, da sie für das geplante Lösungsverfahren in dieser Arbeit nicht relevant sind.

Rescheduling environments				
Static (finite set of jobs)		Dynamic (infinite set of jobs)		
Deterministic (all information given)	Stochastic (some information uncertain)	No arrival variability (cyclic production)	Arrival variability (flow shop)	Process flow variability (job shop)

Rescheduling strategies				
Dynamic (no schedule)		Predictive-reactive (generate and update)		
Dispatching rules	Control-theoretic	Rescheduling policies		
		Periodic	Event-driven	Hybrid

Rescheduling methods				
Schedule generation		Schedule repair		
Nominal schedules	Robust schedules	Right-shift rescheduling	Partial rescheduling	Complete regeneration

Abbildung 2.9: Neuplanungsklassifikation aus Vieira u. a. (2003)

2.3.1 Neuplanungsumgebungen

Nach Vieira u. a. (2003) identifiziert die Neuplanungsumgebung die Menge an Aufträgen, welche beplant werden muss. Planungsprobleme lassen sich nach diesem Merkmal in *statische Neuplanungsprobleme* mit einer endlichen Menge an Aufträgen und *dynamische Neuplanungsprobleme* mit einer nicht endlichen Menge an Aufträgen unterteilen. Mit letzterem ist gemeint, dass die Aufträge laufend über einen unendlichen Zeithorizon im System anlangen.

Die statischen Neuplanungsumgebungen lassen sich weiter in deterministische und stochastische einteilen.

- *Deterministische, statische Neuplanungsumgebungen* können als Spezialfall der Neuplanung angesehen werden, in denen es eine endliche Menge an Aufträgen gibt und keine Unsicherheit herrscht. Der angestrebte Belegungsplan

kann also ohne Veränderungen umgesetzt werden, da es keine unbekannten Einflüsse gibt.

- Auch bei *stochastischen, statischen Neuplanungsumgebungen* ist die Menge der Aufträge endlich, aber einige Variablen sind unscharf. Es können zum Beispiel die Bearbeitungszeiten der Aufträge als Zufallsvariablen modelliert sein, wodurch die tatsächlich auftretenden Beginn- und Fertigstellungszeiten von den erwarteten aus dem Belegungsplan abweichen.

Dynamische Neuplanungsumgebungen treten in drei wichtigen Formen auf.

- Bei *dynamischen Neuplanungsumgebungen ohne Ankunftsvariabilität* (eng. dynamic rescheduling environments without variability in the arrival process) kommen die Aufträge immer in dem gleichen bekannten Muster im System an und der Belegungsplan kann kontinuierlich wiederholt werden.

- Im Falle der *dynamischen Neuplanungsumgebungen mit Ankunftsvariabilität* (eng. dynamic rescheduling environments with arrival variability) ist zwar nicht bekannt, wann die Aufträge im System einlangen, alle Aufträge haben jedoch die selbe Route durch das Produktionssystem und die Ankuftsrate ist gleichmässig.

- Die dritte Variante sind *dynamische Neuplanungsumgebungen mit Ankunftsvariabilität und Routenvariabilität* (eng. dynamic rescheduling environments with process flow and arrival variability), bei denen nicht bekannt ist, wann die Aufträge in das Sytem kommen und jeder Auftrag eine eigene Route haben kann, die in machen Fällen erst beim Einlangen des Auftrages bekannt wird.

2.3.2 Neuplanungstrategien

Laut Vieira u. a. (2003) beschreiben Neuplanungstrategien, ob Belegungspläne generiert werden oder nicht. Die beiden vorgestellten Strategien sind dynamische Planung (eng. dynamic scheduling) und vorhersagend-reaktive Planung (eng. predictive-reactive scheduling).

Dynamische Planung erzeugt keine Belegungspläne. Stattdessen verteilen dezentrale Produktionskontrollmethoden die Aufträge wann immer nötig und mit den zu dem jeweiligen Zeitpunkt vorhandenen Informationen. Es werden Prioritätsregeln oder andere Heuristiken benutzt, um zu bestimmen welcher Auftrag wo als nächstes bearbeitet wird. Manche Autoren bezeichnen dynamische Planung auch als Online-Planung (eng. online scheduling) oder reaktive Planung (eng. reactive scheduling).

Die *vorhersagend-reaktive Planung* erzeugt Belegungspläne, in denen für jeden Auftrag auf jeder Maschine festgelegt ist, wann er startet und wann er endet. Sie besteht aus zwei Hauptschritten.

1. Im ersten Schritt wird ein Belegungsplan erstellt. In Jensen (2003) wird dieser auch Vorplan (eng. preschedule) genannt.

2. Im zweiten Schritt wird der Belegungsplan aktualisiert, wenn eine Störung oder ein anderes Ereigniss eintritt, um die negativen Auswirkungen auf die Bewertungsmaße des Produktionssystem zu minimieren.

2.4 Ein stochastisches 2-stufiges FFc

Das in dieser Arbeit behandelte Problem ist ein flexibles Flow-Shop-Problem mit einer Maschine in der 1. Stufe, zwei unterschiedlichen Maschinen in der 2. Stufe und limitiertem Puffer zwischen den beiden Stufen (siehe Abbildung 2.10). Es stellt den Produktionsbetrieb eines in der Metallverarbeitung tätigen Unternehmens abstrakt dar. In der 1. Stufe werden in diesem Betrieb Rohmaterialien zu einem Zwischenprodukt verarbeitet welches danach in der 2. Stufe verfeinert wird.

Die Anzahl der Aufträge und ihre Ankunftszeiten sind vor Beginn der Optimierung bekannt, wodurch das Problem ein statisches ist. Alle Bereitstellungstermine sind 0 und es liegen keine spätesten Fertigstellungstermine und keine Rüstzeiten vor. Die Bearbeitungszeiten der Aufträge sind mit Unsicherheit behaftet und es gibt Maschinenausfälle, wodurch das Problem zur Klasse der stochastischen Probleme gehört.

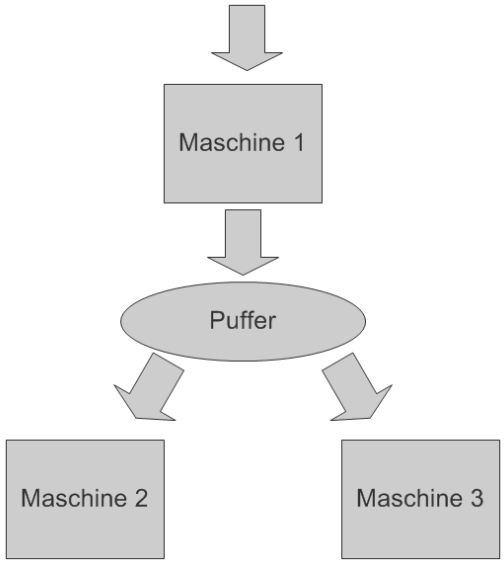

Abbildung 2.10: Schematische Darstellung des Modells

Formel 2.3 stellt die Einteilung dieses Modells nach der in Abschnitt 2.2 vorgestellten Notation dar.

$$FFc|brkdwn,block,prmu|\sum w_jC_j,cust \tag{2.3}$$

Grundsätzlich wäre das Problem ohne "prmu" im β-Feld. Um jedoch die Komplexität des Modells zu vereinfachen (siehe weiter unten) wird nur die Auftragsfolge der ersten Maschine verändert.

Für das γ-Feld ist anzumerken, dass alle w_j gleich 1 sind und, dass "cust" für von den Standardzielfunktionen aus Abschnitt 2.2.3 abweichende Zielfunktionen steht. Da Maschine 1, im Vergleich zu den Maschinen der Stufe 2, ein relativ teures Aggregat ist, kann es als zweckmäßig angesehen werden dafür zu sorgen, dass diese Maschine gut ausgelastet ist. Aus diesem Grund muss die *Auslastung der Maschine 1* als Optimierungsziel berücksichtigt werden. Dies geschieht, indem die kumulierten Stillstandzeiten der Maschine minimiert werden.

Auch ein voller Puffer kann zu einer Blockade der ersten Maschine führen, also zu einem Zustand der sich negativ auf das Ziel einer hohen Maschinenauslastung auswirkt. Aus diesem Grund wird auch die Minimierung des *durchschnittlichen Pufferstandes* als Optimierungsziel wahrgenommen. Dies geschieht aus dem Ge-

danken heraus, dass je niedriger der Pufferstand ist, Maschine 1 umso länger bei Störungen in Stufe 2 weiter arbeiten kann.

Größe des Lösungsraumes

In Abschnitt 2.2.1.1 wurde erläutert, dass die Größe des Lösungsraumes für ein Einmaschinenproblem $n!$ ist, wobei n die Anzahl der Aufträge ist. Abschnitt 2.2.1.2 gibt die Grösse des Lösungsraumes für ein Modell mit 2 parallelen Maschinen mit $\sum_{k=0}^{n} \binom{n}{k}(n-k)!(k)!$ an. Im konkreten Modell mit einer Maschine in der ersten Stufe und zwei Maschinen in der zweiten Stufe müssen für jede Auftragsfolge der ersten Stufe alle möglichen Auftragsfolgen der zweiten Stufe untersucht werden. Die Größe des Lösungsraumes für das hier behandelte Problem ist somit $n! \sum_{k=0}^{n} \binom{n}{k}(n-k)!(k)!$.

Bei 10 Aufträgen sind es bereits mehr als 144 Billionen Möglichkeiten. Gehen wir von 10^{-6} Sekunden (einer millionsten Sekunde) pro Möglichkeit aus, um eine Auswertung durchzuführen, dann würde es ca. 31 Jahre dauern um den kompletten Lösungsraum zu durchsuchen. Eine Variante dieser Flut von möglichen Anordnungen zu begegnen (siehe zum Beispiel Jungwattanakit u. a., 2009) ist es, nur die Anordnung in der 1. Stufe (im Modell in dieser Arbeit auf der ersten Maschine) zu variieren und die Aufteilung und Anordnung auf den anderen Stufen einfachen Regeln wie First-In First-Out (FIFO) zu überlassen. Bei dieser Methode, die auch in dieser Arbeit angewendet wird, wird jener Auftrag, der am längsten im Puffer vor einer Stufe ist, der nächsten frei werdenden Maschine zugeordnet. Die Größe des Lösungsraumes wird dadurch auf n!, also die eines Einmaschinenproblems, reduziert.

Berechnung des Belegungsplanes

Wie bereits in Abschnitt 2.2.1.4 eingeführt, gibt es für diese spezielle Problemstellung keine einfache Möglichkeit der Berechnung des Belegungsplanes. Daher greift man im Allgemeinen auf die Unterstützung von Simulationsprogrammen wie AnyLogic™zurück[6]. Solche Produkte bieten Unterstützung in der Erstellung von teils hoch-komplexen Simulationen, die es dem Forscher erlauben, sich auf das Erstellen des Modells zu konzentrieren und nicht auf die Implementierung einer Ablaufsimulation. Die Umsetzung in AnyLogic™, des im Rahmen dieser Arbeit behandelten Modells, wird in Abschnitt 4.1 vorgestellt.

[6] AnyLogic, XJ Technologies Company Ltd., http://www.xjtek.com/

Stochastik

Nach Abschnitt 2.3 ist das hier behandelte Problem den stochastischen, statischen Neuplanungsumgebungen zuzuordnen da die Menge der Aufträge im System endlich ist, es jedoch zu Maschinenausfällen kommt und die Bearbeitungszeiten Zufallsvariablen sind. Die angedachten Lösungsverfahren (siehe Abschnitt 3 und Abschnitt 4) bauen auf der Idee von Prioritätsregeln auf, sind also den dynamischen Neuplanungsstrategien zuzuordnen.

Wie bereits erwähnt, ist eine Möglichkeit der Unsicherheit in diesem Modell die *Variabilität der Bearbeitungszeiten*. In deterministischen Modellen wird davon ausgegangen, dass ein Auftrag j eine fixe Bearbeitungszeit p_{ij} auf Maschine i hat. In einer realen Umgebung kann diese jedoch zum Beispiel wegen Verschlechterung des Maschinenzustandes oder wegen variierender Effizienz des Personals, das die Maschine bedient, schwanken.

Liegen empirische Daten aus der Produktion über die, von den einzelnen Aufträgen realisierten Bearbeitungszeiten, vor, kann die daraus gewonnene empirische Verteilung zum Berechnen der tatsächlichen Bearbeitungszeit herangezogen werden. In dieser Arbeit wird angenommen, dass diese Daten nicht zur Verfügung stehen, es jedoch durch einen erfahrenen Mitarbeiter oder aus statistisch Erhebungen möglich ist eine Schätzung über den Wert der Bearbeitungszeit zu machen. Diese wird beste Schätzung der Bearbeitungszeit (eng. best estimate of the processing time) genannt und wird um einen Wert, den man aus einer Verteilung zieht die die Abweichungen repräsentiert, verändert um die eigentliche Bearbeitungszeit (eng. actual processing times) zu erhalten.

Aus Mangel an empirischen Daten über die Abweichungen der tatsächlichen Bearbeitungszeiten von der besten Schätzung werden die tatsächlichen Bearbeitungszeiten in dieser Arbeit wie in Kutanoglu und Sabuncuoglu (2001) nach der Formel 2.4 berechnet.

$$p'_{ij} = (1 + V \times U[-1.0, +1.0]) \times p_{ij} \tag{2.4}$$

wobei p'_{ij} die eigentliche Bearbeitungszeit und p_{ij} die beste Schätzung der Bearbeitungszeit von Auftrag j auf Maschine i ist. $U[-1.0, +1.0]$ ist eine Zahl, die

aus der Gleichverteilung zwischen -1 und 1 gezogen wird. V ist ein experimenteller Faktor der zwischen 0 und 1 variiert werden kann, um mehr oder weniger stochastischen Einfluß zu simulieren.

Die zweite Quelle für Unsicherheit in diesem Modell ist der *Ausfall von Maschinen*. Diese können wie in Kutanoglu und Sabuncuoglu (2001) nach dem Belegungszeit-Ansatz (eng. busy time approach), der von Law und Kelton (Law und Kelton, 1999) vorgeschlagen wurde, modelliert werden. Eine zufällige Betriebszeit wird aus der so genannten Belegungszeit-Verteilung (eng. busy time distribution) gezogen. Die Maschine arbeitet so lange die kumulierten Bearbeitungszeiten der von ihr bearbeiteten Aufträge kleiner sind als die Betriebszeit. Dann wird die Ausfallszeit, jene Zeitdauer in welcher die Maschine auf Grund des Ausfalles keine Aufträge bearbeiten kann, aus einer Ausfallszeit-Verteilung (eng. downtime distribution) gezogen und die Maschine bearbeitet für diese Dauer keine Aufträge mehr.

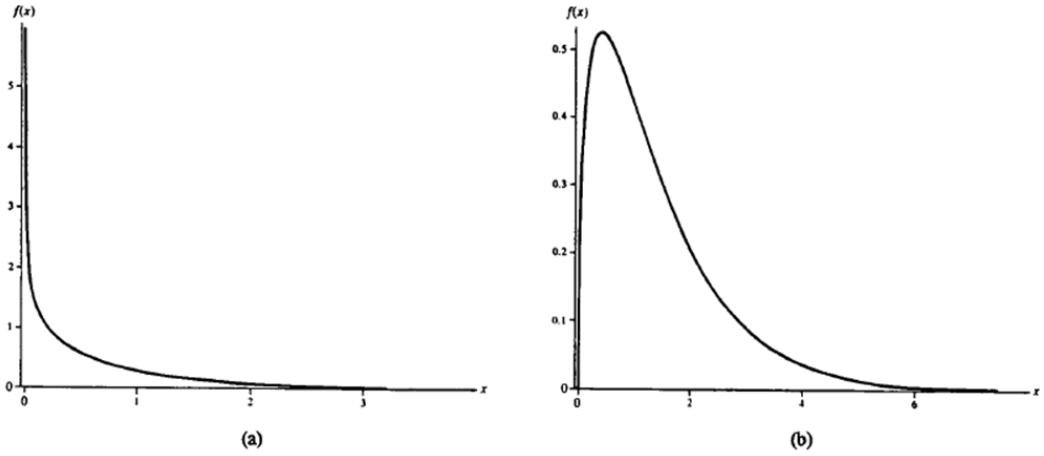

Abbildung 2.11: (a) Belegungszeit-Verteilung Gamma(0,7;1)
(b) Ausfallszeit-Verteilung Gamma(1,4;1)
aus Law (1990)

Sowohl die Belegungszeit-Verteilung als auch die Ausfallszeit-Verteilung können nach Law und Kelton als Gammaverteilungen angenommen werden. Diese empfehlen, wenn keine realen Daten vorliegen, eine Gammaverteilung mit den Parametern $\alpha_b = 0,7$ und β_b entsprechend der Versuchsanordung für die Belegungszeit-Verteilung und eine Gammaverteilung mit Parameter $\alpha_d = 1,4$ und

β_d entsprechend der Versuchsanordung für die Ausfallszeit-Verteilung. Die Häufigkeit von Maschinenfehler wird durch Verfügbarkeit gemessen, welche die Langzeitrate von Betriebszeit durch Betriebszeit plus Ausfallszeit ist. Die Dauer jedes Ausfalles wird in Kutanoglu und Sabuncuoglu (2001) aus Gamma($\alpha_d = 1,4; \beta_d = d_{avg}/1,4$) und die Betriebszeit zwischen zwei Ausfällen aus Gamma($alpha_b = 0,7; beta_b = d_{avg} * (\frac{\epsilon}{0,7*(1-\epsilon)})$ berechnet. d_{avg} ist die mittlere Ausfallszeit und ϵ ist der Effizienzlevel. Beispiele für die verwendeten Verteilungen bei $d_{avg} = 1,4$ und $\epsilon = \frac{1}{3}$ sind in Abbildung 2.11 dargestellt.

Kapitel 3

Lösungsverfahren

Dieses Kapitel beschreibt die zur Verfügung stehenden Lösungverfahren für die in Kapitel 2 vorgestellten Modelle. Die bevorzugten Verfahren, jedoch bei vielen Modellen und Anwendungen wegen des riesigen Lösungsraumes nicht möglichen, sind jene für welche bewiesen wurde, dass sie die optimale Lösung liefern. Solche Verfahren werden *exakte Verfahren* genannt und in Abschnitt 3.1 vorgestellt.

Bei vielen kombinatorischen Optimierungsproblemen ist es nicht möglich exakte Verfahren zu verwenden, da diese viel zu lange benötigen würden, eine Lösung zu liefern. Formal bezeichnet man diese als NP-schwer. Das bedeutet, dass die Laufzeit für jeden bisher bekannten Algorithmus, der garantiert das optimale Ergebnis liefert, exponentiell mit der Größe des Problems wächst. Beispiele für solche Probleme sind das Problem des Handlungsreisenden (eng. Travelling Salesman Problem) und auch viele Maschinenbelegungsprobleme. Auf Grund der Komplexität des hier behandelten Problems und der Tatsache, dass die meisten Maschinenbelegungsprobleme mit mehreren Maschinen NP-schwer sind, ist anzunehmen, dass das hier behandelte flexible Flow-Shop-Probleme ebenfalls NP-schwer ist.

Daher werden kombinatorische Optimierungsprobleme im allgemeinen mit *Heuristiken* behandelt. Diese Verfahren versuchen ein gegebenes Problem relativ gut in akzeptabler Zeit zu lösen, ohne jedoch beweisen zu können, das optimale Ergebnis zu erzielen.

In Abschnitt 3.1 werden exemplarisch exakte Verfahren vorgestellt. Abschnitt 3.2 beschäftigt sich mit einfachen Heuristiken und Abschnitt 3.3 geht auf Metaheuristiken ein. Zuletzt enthält Abschnitt 3.4 Ergänzungen die für stochastische Modelle notwendig sind.

3.1 Exakte Verfahren

Exakte Verfahren haben die Eigenschaft, dass nach fehlerfreier Durchführung des Verfahrens der optimale Wert im Sinne der gegebenen Zielfunktion gefunden wurde. Abschnitt 3.1.1 zeigt Modelle für die Lösungsstrategien existieren die einfach und schnell aber trotzdem exakt sind. Abschnitt 3.1.2 beschreibt eine mathematische Formulierung für Maschinenbelegungsprobleme.

3.1.1 Einfache Probleme

Es existieren Spezialfälle von einfachen Maschinenbelegungsproblemen, für die es möglich ist, schnell eine exakte Lösung zu finden, d.h. es gibt einen polynomialen Lösungsalgorithmus. Im Einmaschinenfall zum Beispiel kann mit der Shortest Processing Time (SPT) Prioritätsregel die minimale durchschnittliche Durchlaufzeit exakt bestimmt werden ($1||\sum_j F_j$). Für die selbe Modellannahme ist es möglich, die maximal auftretende Abweichung vom Fertigstellungstermin mit der Earliest Due Date (EDD) Prioritätsregel zu minimieren ($1||max_j(L_j)$).

Moore (Nahmias, 2000) hat für das Einmaschinenproblem einen Algorithmus entwickelt, der die Anzahl der Aufträge minimiert, die nach ihrem Fertigstellungstermin fertiggestellt werden ($1||\sum U_j$). Im ersten Schritt werden die Aufträge aufsteigend nach ihren Fertigstellungsteminen sortiert (EDD). Schritt 2 sucht nach dem ersten verspäteten Auftrag in der soeben erstellten Auftragsfolge und bezeichnet seinen Platz in der Folge als [i]. Falls keiner gefunden wurde, wird mit Schritt 4 fortgesetzt. Schritt 3 betrachtet die Aufträge [1], ..., [i] und verwirft jenen Auftrag mit der größten Bearbeitungszeit. Danach wird zu Schritt 2 zurückgekehrt. Schritt 4 bildet die optimale Auftragsfolge, indem zur aktuellen Folge die verworfenen Aufträge am Ende hinzugefügt werden. In welcher Reihenfolge die verworfenen Aufträge angefügt werden ist, irrelevant da sie ohnehin zu spät sind.

Ein ebenfalls einfaches und schnelles Verfahren existiert für den Fall, dass zwei Maschinen mit n Aufträgen belegt werden sollen. Die Aufträge müssen alle zuerst auf Maschine A und danach auf Maschine B bearbeitet werden und die Zielfunktion ist die größte Fertigstellungszeit aller Aufträge zu minimieren ($F2|prmu|C_{max}$). Wichtig ist hier die Erkenntnis, dass für dieses Problem die optimale Anordung immer eine Permutationsanordnung (eng. permutation schedule) ist. Dies bedeutet, dass nur Anordnungen zu betrachten sind, die auf beiden Maschinen die selbe Reihung der Aufträge haben.

Ein effizienter Algorithmus um dieses Problem zu lösen wurde von *Johnson* entwickelt (Nahmias, 2000). Nennen wir die beiden Maschinen A und B und alle Aufträge müssen zuerst auf Maschine A und danach auf Maschine B bearbeitet werden. Es gibt n Aufträge und die Bearbeitungszeit auf Maschine A sei mit A_i und auf Maschine B mit B_i bezeichnet, wobei $1 \leq i \leq n$ in beiden Fällen ist. Johnson kam zu dem Ergebnis, dass folgende Regel den makespan minimiert: Auftrag i kommt vor Auftrag i+1 wenn $min(A_i, B_{i+1}) < min(A_{i+1}, B_i)$.

Diese Regel kann wie folgt umgesetzt werden:

1. Erstelle 2 Spalten mit den Werten A_i in der 1. und den Werten B_i in der 2.
2. Suche das kleinste verbleibende Element in den beiden Spalten. Wenn es in Spalte A ist, dann wird es als nächstes verwendet, wenn es in Spalte B ist dann als letztes.
3. Die Zeilen von Aufträgen die verwendet wurden werden gestrichen und es wird abgebrochen wenn alle Aufträge gestrichen sind.

3.1.2 Gemischt-ganzzahlige lineare Modellformulierung

Um eine formal eindeutige Darstellung von Optimierungsproblemen zu erhalten, können diese als lineare Programme (eng. Linear Program, LP) oder gemischtganzzahlige lineare Programme (eng. Mixed Integer Linear Program, MILP) formuliert werden. Eine ausführliche Einführung ist in Hillier und Lieberman (2002) zu finden. Außerdem gibt es freie und kommerzielle Computerprogramme, Löser (eng. solver) genannt, welche in der Lage sind, solch eine Formulierung automatisch zu lösen. Es ist zu beachten, dass im Gegensatz zu linearen Programmen

für gemischt-ganzzahlige lineare Programme kein effizienter (in polynomialer Zeit lösender) Algorithmus existiert.

Als Beispiel für ein gemischt-ganzzahliges lineares Programm soll ein Einmaschinenproblem mit Minimierung der gesamten gewichteten Fertigstellungszeit als Zielfunktion ($1||\sum w_j C_j$) aus Pinedo (2007) dienen. Dieses Problem kann zwar mit der Weighted Shortest Processing Time (WSPT) Regel optimal gelöst werden, die Problemformulierung lässt sich jedoch auf allgemeinere Fälle erweitern.

Sei x_{jk} die binäre Entscheidungsvariable, welche den Wert 1 annimmt wenn Auftrag j irgendwann vor Auftrag k kommt und sonst 0 ist. Die Variablen x_{jj} sind 0 für alle j. Die Fertigstellungszeit von Auftrag j wird berechnet durch die Formenl $\sum_{k=1}^{n} p_k x_{kj} + p_j$, wobei p_k die Bearbeitungszeit von Auftrag k ist. Das gemischt-ganzzahlige Programm ist in den Formeln 3.1 bis 3.5 dargestellt.

$$Minimiere \sum_{j=1}^{n} \sum_{k=1}^{n} w_j p_k x_{kj} + \sum_{j=1}^{n} w_j p_j \tag{3.1}$$

$$x_{kj} + x_{jk} = 1 \qquad \forall j,k = 1,...,n, j \neq k \tag{3.2}$$

$$x_{kj} + x_{lk} + x_{jl} \geq 1 \qquad \forall j,k,l = 1,...,n, j \neq k, j \neq l \tag{3.3}$$

$$x_{jj} = 0 \qquad \forall j = 1,...,n \tag{3.4}$$

$$x_{jk} \in 0,1 \qquad \forall j,k = 1,...,n \tag{3.5}$$

Formel 3.1 ist die Zielfunktion zum Minimieren der gesamten gewichteten Fertigstellungszeit. Die ersten Nebenbedingungen (3.2) legen fest, dass ein Auftrag nicht gleichzeitig Vorgänger und Nachfolger eines anderen Auftrages sein kann. Die zweiten Nebenbedingungen (3.3) stellen sicher, dass jeder Auftrag mindestens einen Vorgänger oder einen Nachfolger hat. Die dritten Nebenbedingungen (3.4) verhindern, dass ein Auftrag sein eigener Vorgänger ist und die letzten Nebenbedingungen (3.5) legen fest, dass die Entscheidungsvariablen binär sind.

Es ist sicher möglich, auch für das in dieser Arbeit behandelte Modell, ein gemischt-ganzzahliges lineares Modell, wenn auch ungleich komplexer, herzuleiten (für MILPs aus ähnlichen Fragestellungen siehe Jungwattanakit u. a. (2009) und Sawik (1993)). Da der Fokus in dieser Arbeit jedoch auf heuristischen Ansätzen liegt und ohnehin bei einer realistischen Anzahl von Aufträgen die Lösung des

gemischt-ganzzahligen linearen Models zu lange dauern würde, wird dem hier nicht weiter nachgegangen.

3.2 Heuristiken

Am Anfang dieses Kapitels wurde bereits ausgeführt, warum es nicht möglich ist, immer exakte Verfahren zum Lösen von Optimierungsproblemen zu verwenden. Als Alternative kann eine Heuristik verwendet werden, welche ein Verfahren ist, das versucht ein gegebenes Problem relativ gut in akzeptabler Zeit zu lösen, ohne jedoch den Beweis zu haben, dass das optimale Ergebnis erzielt wird.

Beispiele für Heuristiken bei Maschinenbelegungsproblemen sind Prioritätsregeln (Abschnitt 3.2.1), die schon zuvor vorgestellt worden sind, da sie für einfache Modelle exakte Verfahren sein können, und die lokale Suche (Abschnitt 3.2.2).

3.2.1 Prioritätsregeln

Prioritätsregeln (eng. dispatching oder scheduling rules) errechnen anhand der Auftragsdaten für jeden Auftrag einen Kennwert und sortieren die Aufträge aufsteigend nach diesem Kennwert. Sie werden gerne verwendet, da sie meist sehr einfach umzusetzen sind und eine niedrige Zeitkomplexität aufweisen. Eine ausführliche Zusammenfassung von Regeln ist in Panwalkar und Iskander (1977) zu finden.

Einfache Regeln sind zum Beispiel Shortest Processing Time (SPT), bei der der Kennwert die Bearbeitungszeit ist, oder Earliest Due Date (EDD) mit Kennwert Fertigstellungstermin. Diese beiden Regeln wurden auch bereits bei den exakten Verfahren vorgestellt, da sie für das Einmaschinenmodell die durchschnittliche Durchlaufzeit, bzw. maximal auftretende Abweichung vom Fertigstellungstermin exakt minimieren.

3.2.2 Lokale Suche

Das Verfahren der lokalen Suche geht davon aus, kleine Veränderungen an einer Lösung vorzunehmen und zu überprüfen, ob diese Veränderung zu einer

Verbesserung im Sinne der Zielfunktion führt. Wurde eine Verbesserung gefunden, dann wird die Methode auf die neue Lösung angewendet. Das wird solange wiederholt bis keine Verbesserung mehr gefunden wird und somit ein lokales Optimum erreicht ist.

Der Abschnitt 3.2.2.1 basiert auf Vaessens u. a. (1996) und gibt einen Überblick über den Grundgedanken von lokaler Suche. Der Abschnitt 3.2.2.2 basiert auf den Besten und Stützle (2001) und bechreibt Nachbarschaften für das Einmaschinenproblem.

3.2.2.1 Grundgedanke

Für eine formelle Spezifikation des Algorithmus ist es notwendig, folgende Notation einzuführen. Die Menge X besteht aus allen zulässigen Lösungen. Zwei Funktionen werden auf X definiert. Die Kostenfunktion $f : X \rightarrow R$ und die Nachbarschaftsfunktion N die für jede Lösung $x \in X$ eine Nachbarschaft $N(x) \subseteq X$ definiert. Jede Lösung aus $N(x)$ wird Nachbar von x genannt. Eine Lösung $x \in X$ wird lokales Optimum auf der Nachbarschaftsfunktion N genannt wenn $f(x) \leq f(y)$ für alle $y \in N(x)$ gilt.

Eine simple Variante, um ein lokales Optimum zu finden, ist die einfache lokale Suche. Man beginnt mit einer Ausgangslösung und deren Nachbarschaft wird nach einer Lösung mit geringeren Kosten durchsucht. Wird so eine Lösung gefunden, wiederholt man den Vorgang. Wird keine Lösung gefunden, ist die aktuelle Ausgangslösung ein lokales Optimum.

Es gibt mehrere Möglichkeiten eine Nachbarschaft zu durchsuchen: Bei „erster Verbesserung“ (eng. first improvement) nimmt man die erste gefundene Lösung, die besser ist als die Ausgangslösung und bei „bester Verbesserung“ (eng. best improvement) nimmt man jene Lösung, die die höchste Verbesserung gegenüber der Ausgangslösung bietet. Abbildung 3.1 zeigt die Variante mit bester Verbesserung.

3.2.2.2 Nachbarschaften für Maschinenbelegung

Im letzten Abschnitt wurde $N(x)$ als Nachbarschaftsfunktion für die Lösung x definiert. Für verschiedene Probleme gibt es unterschiedliche Möglichkeiten Nachbarschaftsfunktionen zu definieren. Für Permutationsschedulingprobleme, wie das

```
1 Gegeben sei die Kostenfunktion f;
2 Gegeben sei die Menge der zulässigen Lösungen X;
3 Gegeben sei die Nachbarschaft N(x);
4 Wähle eine Ausgangslösung x;
5 Wiederhole{
6   Suche x' = argmin_{y∈N(x)} f(y);
7   Wenn f(x') < f(x) dann x := x';
8   Sonst stoppe das Verfahren;
9 }
```

Abbildung 3.1: Pseudo-Code der einfachen lokalen Suche mit bester Verbesserung

in dieser Arbeit behandelte Problem, geben den Besten und Stützle (2001) als Standardnachbarschaften Transpose, Interchange und Insert an. Von den drei genannten ist die kleinste Nachbarschaft die *Transpose-Nachbarschaft*. Sie besteht aus allen Permutationen, die durch vertauschen zweier benachbarter Aufträge entstehen und ist $(n-1)$ Elemente groß.

Die *Interchange-Nachbarschaft* (in den Besten und Stützle, 2001; als Pairwaise-Interchange-Nachbarschaft in Jungwattanakit u. a., 2009 bezeichnet) wird gebildet durch alle Permutationen die entstehen, wenn jeweils zwei Elemente ihre Plätze tauschen (ein Element kann mit sich selbst nicht Platz tauschen), unabhängig ob sie in der Lösungsdarstellung nebeneinander sind oder nicht. In dieser Nachbarschaft gibt es $\frac{n*(n-1)}{2}$ Elemente.

Die *Insert-Nachbarschaft* (in den Besten und Stützle, 2001; als Shift-Move-Nachbarschaft in Jungwattanakit u. a., 2009 bezeichnet) besteht aus allen Permutationen die entstehen, wenn man ein Element entfernt und an einer anderen Position (nicht an jener wo es entfernt wurde) wieder einfügt. Diese Nachbarschaft hat $(n-1)^2$ Elemente.

Die Transpose-Nachbarschaft ist eine Teilmenge der Interchange- und Insert-Nachbarschaft. Die Interchange-Nachbarschaft und die Insert-Nachbarschaft sind komplementäre Teile einer generelleren 2-opt Nachbarschaft, die sowohl Permutationen beinhaltet, die durch normales Tauschen von Elementen entstehen, als auch Permutationen die entstehen, wenn man ein Element mit den „Leer-Element“ tauscht.

3.3 Metaheuristiken

Eine Metaheuristik ist ein abstraktes Verfahren das versucht, unabhängig von der konkreten Problemstellung, zu definieren welche Schritte nötig sind, um ein bestimmtes Ziel zu erreichen. Der Name kombiniert sich aus dem griechischen Vorwort "meta", zu Deutsch "nach" (im Sinne von Weiterentwicklung), und "heuristik", zu Deutsch "finden". Im Operations Research sind damit meist Verfahren gemeint, die zum Lösen von kombinatorischen Optimierungsproblemen verwendet werden. Der oft große Lösungsraum dieser Probleme macht die meisten exakten Verfahren unbrauchbar, da sie zu viel Zeit beanspruchen. Es wird daher ein Verfahren benötigt das in annehmbarer Zeit zu einem relativ guten Ergebnis durch den riesigen Lösungsraum navigiert.

Metaheuristiken benutzten dabei Terminologie, die (theoretisch) unabhängig von einer spezifischen Problemstellung ist, zum Beispiel die Nachbarschaftsfunktion $N(x)$. Die konkrete Funktion, die $N(x)$ für ein spezifisches Problem umsetzt, wird als Black-Box betrachtet. Die Verfahren sollten somit auf jegliches kombinatorisches Optimierungsproblem angewendet werden können, wenn es möglich ist die abstrakten Konzepte mit problem-spezifischen Implementationen zu ersetzen. Metaheuristiken werden ausführlich in Glover und Kochenberger (2003) behandelt. Beispiele für Metaheurisitken sind Tabu Search (Glover, 1986), Ant Colony Optimization (Dorigo und Stützle, 2004), Variable Neighborhood Search (Mladenović und Hansen, 1997) und Simulated Annealing (Kirkpatrick u. a., 1983), wobei diese Liste keinen Anspruch auf Vollständigkeit erhebt.

In dieser Arbeit werden, zusätzlich zu den heuristischen Prioritätsregeln, zwei Metaheuristiken auf das gegebene Modell angewendet. Die erste ist *Variable Neighborhood Search* (VNS). In Abschnitt 3.2.2 wurde das Prinzip der lokalen Suche vorgestellt. In dieser wird eine Nachbarschaft durchsucht, bis ein lokales Optimum in Bezug auf diese Nachbarschaft gefunden wurde. VNS basiert auf der Annahme, dass ein lokales Optimum in Bezug auf eine Nachbarschft nicht auch ein lokales Optimum in Bezug auf eine andere Nachbarschaft sein muss. Es werden daher in "intelligenter" Weise unterschiedliche Nachbarschaften durchsucht. VNS wird ausführlich in Abschnitt 3.3.1 behandelt.

Simulated Annealing (SA) ist die zweite Metaheuristik die hier vorgestellt wird. Auch dieses Verfahren basiert auf der Idee der lokalen Suche und bewegt sich,

wie diese, in einer Nachbarschaft. Das Verfahren versucht jedoch zu verhindern, in lokalen Optima gefangen zu werden, indem es kontrolliert auch Schritte zulässt, die zu einer Verschlechterung des Zielfunktionswertes führen. SA wird ausführlich in Abschnitt 3.3.2 behandelt.

3.3.1 Variable Neighboorhood Search

Variable Neighborhood Search (VNS) basiert auf der Idee systematisch die Nachbarschaften innerhalb einer lokalen Suche zu verändern. Folgende Annahmen führten laut Glover und Kochenberger (2003) zu dieser Idee.

1. Ein lokales Optimum in Bezug auf eine Nachbarschaft muss nicht auch ein lokales Optimum in Bezug auf eine andere Nachbarschaft sein.
2. Ein globales Optimum ist ein lokales Optimum in Bezug auf alle Nachbarschaften.
3. Für viele Probleme liegen lokale Optima in Bezug auf verschiedene Nachbarschaften nahe beieinander.

Die Abschnitte über Variable Neighborhood Descent (Abschnitt 3.3.1.1), Reduced Variable Neighborhood Search (Abschnitt 3.3.1.2) und Variable Neighborhood Search (Abschnitt 3.3.1.3) basieren auf Hansen und Mladenovic (2003) und Glover und Kochenberger (2003).

3.3.1.1 Variable Neighborhood Descent

Die einfache lokale Suche basiert darauf, in einer Nachbarschaft so lange zu suchen, bis das lokale Optimum erreicht ist. Eine mögliche Erweiterung der einfachen lokalen Suche ist das *Variable Neighborhood Descent* (VND) genannte Verfahren. Hier wird zwischen mehreren Nachbarschaften deterministisch gewechselt. VND basiert auf der Erkenntnis, dass ein lokales Minimum für eine Nachbarschaft nicht notwendigerweise ein lokales Minimum für eine andere Nachbarschaft ist, wodurch es vorteilhaft sein kann verschiedene Nachbarschaften zu kombinieren.

Es wird eine Menge an Nachbarschaften N_l mit $l = 1, ..., l_{max}$ definiert. Die Suche startet in der ersten Nachbarschaft, in der nach dem besten Nachbarn

der Ausgangslösung x gesucht wird. Stellt dieser Nachbar eine Verbesserung dar, wird dieser als aktuelle Lösung übernommen und die Suche in der ersten Nachbarschaft fortgesetzt. Wurde kein besserer Nachbar gefunden, wird in der zweiten Nachbarschaft gesucht. Dieses Prinzip setzt sich bis zur l-ten Nachbarscht fort, jedoch wird ab der zweiten Nachbarschaft zur ersten zurück gesprungen, wenn eine Verbesserung gefunden wurde. Abbildung 3.2 zeigt das Verfahren.

```
1  Gegeben sei die Kostenfunktion f;
2  Gegeben sei eine Menge an Nachbarschaften N_l mit l = 1, ..., l_max;
3  Wähle eine Ausgangslösung x;
5  Setze l := 1;
6  Wiederhole bis l = l_max + 1{
7     Suche x' = argmin_{y∈N_l(x)} f(y);
8     Wenn f(x') < f(x) dann x := x'; l := 1;
9     Sonst l := l + 1;
10 }
```

Abbildung 3.2: Pseudo-Code Variable Neighborhood Descent

Häufig werden die Nachbarschaften aufsteigend nach ihrer Komplexität angeordnet, was meist der Anzahl ihrer Elemente $|N(x)|$ entspricht, und oft sind diese Nachbarschaften auch ineinander geschachtelt ($N_1(x) \subseteq N_2(x) \subseteq ... \subseteq N_{l_{max}}(x)$). Außerdem wird zur kleinsten Nachbarschaft zurückgekehrt, wenn eine Verbesserung gefunden wurde. Der Gedanke dahinter ist, nahe liegende Nachbarschaften öfter (und dadurch genauer) zu durchsuchen, wenn dort aber keine Verbesserung mehr erzielt werden kann die Suche auf eine größere Nachbarschaft auszudehnen. Durch diese Vorgehensweise kann relativ schnell ein lokales Optimum gefunden werden. Wenn dieses erreicht ist kann die Suche auf einen größeren Bereich ausgedehnt werden um dort zu überprüfen, ob ein noch besseres lokales Optimum gefunden werden kann.

3.3.1.2 Reduced Variable Neighborhood Search

Das Verfahren Reduced Variable Neighborhood Search (RNVS) entsteht wenn man in Abbildung 3.2 in Zeile 7 statt nach dem besten Nachbarn zu suchen, einen zufälligen Nachbarn auswählt (siehe Abbildung 3.3). Diese Änderung kann bei großen Instanzen sinnvoll sein, in denen die lokale Suche aufwendig ist (die benötigt wird um $argmin_{y \in N_l(x)} f(y)$ zu finden). Letzteres zeigt auf, dass der Un-

terschied zwischen den Methoden wie ein Nachbar gewählt wird, im Zielkonflikt zwischen erwarteter Verbesserung und Dauer der Berechnung liegt.

```
1 Gegeben sei die Kostenfunktion f;
2 Gegeben sei eine Menge an Nachbarschaften N_l mit l = 1, ..., l_max;
3 Wähle eine Ausgangslösung x;
4 Wiederhole bis ein Abbruchkriterium erfüllt wird{
5   Setze l := 1;
6   Wiederhole bis l = l_max + 1{
7     Setze x' als zufälliges Element aus N_l(x);
8     Wenn f(x') < f(x) dann x := x'; l := 1;
9     Sonst l := l + 1;
10  }
11 }
```

Abbildung 3.3: Pseudo-Code Reduced Variable Neighborhood Search

3.3.1.3 Variable Neighborhood Search

Die Bausteine von Variable Neighborhood Search sind Schütteln (eng. shaking), lokale Suche (eng. local search) und Akzeptanz (eng. acceptance) welche bis zur Erfüllung eines Abbruchkriteriums wiederholt werden.

- **Schütteln**: Die aktuelle Lösung verändern.
- **Lokale Suche**: Von der veränderten Lösung aus ein lokales Optimum finden.
- **Akzeptanz**: Ist das gefundene lokale Optimum besser als die aktuelle Lösung wird dieses die neue aktuelle Lösung.

In der einfachen Variable Neighborhood Search wird wie bei RVNS ein zufälliger Nachbar der aktuelle Lösung für den Schüttelschritt gewählt und dieser mit irgendeinem lokalen Suchverfahren verbessert. Verwendet als lokales Suchverfahren VND und verbessert man die Initiallösung mit RVNS erhält man das Schema der generellen VNS (siehe Abbildung 3.4).

Das Abbruchkriterium kann zum Beispiel eine maximale Laufzeit, eine Obergrenze an Iterationen oder eine Obergrenze für die maximale Anzahl der Iterationen zwischen zwei Verbesserungen sein. Zu beachten ist, dass $x^{'}$ in Zeile 8

von Abbildung 3.4 auch aus dem Grund zufällig generiert wird, um zyklisches Verhalten zu verhindern.

```
1  Gegeben sei die Kostenfunktion f;
2  Gegeben sei eine Menge an Nachbarschaften N_k mit k = 1, ..., k_max;
3  Gegeben sei eine Menge an Nachbarschaften N_l mit l = 1, ..., l_max;
4  Wähle eine Ausgangslösung x und verbessere sie mit RVNS;
5  Wiederhole bis ein Abbruchkriterium erfüllt wird{
6    Setze k := 1;
7    Wiederhole bis k = k_max + 1{
8      //SCHÜTTELN
9      Setze x' als zufälliges Element aus N_k(x);
10     //LOKALE SUCHE
11     Setze l := 1;
12     Wiederhole bis l = l_max + 1{
13      Suche x'' = argmin_{y∈N_l(x')} f(y);
14       Wenn f(x'') < f(x') dann f(x') := x''; l := 1;
15       Sonst l := l + 1;
16     }
17     //AKZEPTANZ
18     Wenn f(x') < f(x) dann x := x'; k := 1;
19     Sonst k := k + 1;
20   }
21 }
```

Abbildung 3.4: Pseudo-Code Generelle Variable Neighborhood Search

3.3.2 Simulated Annealing

Simulated Annealing (SA) ist die zweite Metaheuristik die auf das in dieser Arbeit beschriebene Modell angewendet wird. Folgende Ausführungen zu SA basieren auf Eglese (1990). Siehe auch Glover und Kochenberger (2003).

SA ist ein allgemeiner stochastischer Meta-Algorithmus, um in guter Näherung das globale Optimum einer gegebenen Funktion in einem großen Suchraum zu finden. Der Name und die Inspiration kommen von der Abkühlung von Metallen, wo eine langsame Abkühlung nach dem Erhitzen dafür sorgt, das die Moleküle sich langsam anordnen und dadurch eine bessere Kristallstruktur erzeugt wird.

SA basiert auf dem Prinzip der lokalen Suche. Dort tritt jedoch das Problem auf, dass der Algorithmus möglicherweise in einem lokalen Optimum stecken

bleibt. Die Strategie von SA ist, es aus lokalen Optima zu entkommen, indem manchmal Schritte zu Nachbarn akzeptiert werden, die eine Verschlechterung der Zielfunktion mit sich bringen. Ob ein Schritt, der eine Verschlechterung bedeutet, angenommen oder abgelehnt wird, hat eine gewisse Wahrscheinlichkeit. Diese wird durch die Akzeptanzfunktion bestimmt.

Sei $f(x)$ der Zielfunktionswert der Ausgangslösung, $f(y)$ der Zielfunktionswert des gewählten Nachbarn und $\Delta = f(y) - f(x)$. Wenn $\Delta < 0$ dann wird die Nachbarlösung y angenommen. Ist jedoch $\Delta \geq 0$, der Nachbar liefert den gleichen oder einen schlechteren Zielfunktionswert, dann wird er trotzdem mit der Wahrscheinlichkeit $exp(-\Delta/T)$, auch *Akzeptanzfunktion* genannt, angenommen. T ist ein Kontrollparameter, der der Temperatur in der Analogie des natürlichen Abkühlungprozesses entspricht.

Die Akzeptanzfunktion ist so angelegt, dass Nachbarn mit einer kleineren Verschlechterung eher angenommen werden, als jene mit einer grösseren. Außerdem werden viele Verschlechterungen angenommen wenn T gross ist, wenn sich T jedoch Null nähert, werden die meisten Verschlechterungen abgelehnt. T wird zu Beginn groß gewählt um zu verhindern, dass der Algorithmus in einem lokalen Optimum stecken bleibt. Die Temperatur wird langsam gesenkt, wobei auf jedem Temperaturlevel eine bestimmte Anzahl an Nachbarn überprüft werden. Das Verfahren ist in Abbildung 3.5 skizziert.

Alternativ zu der in SA verwendeten Akzeptanzfunktion ($exp(-\Delta/T)$) haben Dueck und Scheuer (1990) den Threshold-Accepting-Algorithmus vorgeschlagen mit der Akzeptanzfunktion nach Formel 3.6.

$$a_k(\Delta_{x,y}) = \begin{cases} 1 & wenn\ Q_k \geq \Delta_{x,y} \\ 0 & sonst \end{cases} \tag{3.6}$$

In Formel 3.6 ist $a_k(\Delta_{x,y})$ die Wahrscheinlichkeit, dass der Nachbar y als aktuelle Lösung angenommen wird, wenn der Unterschied zwischen x und y größer oder gleich $\Delta_{x,y}$ ist. Q_k ist der Schwellwert (eng. threshold value) in der Iteration k. Q_k ist normalerweise eine deterministische, nicht steigende Funktion von k. Es hängt also nicht wie bei SA vom Zufall ab ob ein schlechterer Nachbar akzeptiert wird. Bei manchen Problemen ist diese alternative Vorgehensweise dem Vorgehen in SA überlegen.

Die Auswahl der Initialtemperatur, wie die Temperatur verringert wird, die Anzahl der Iterationen pro Temperaturlevel und das Abbruchkriterium bezeichnet man als *Abkühlungsplan* (eng. cooling schedule). Die Zusammenstellung des Abkühlungsplans ist essenziell für die Qualität der Ergebnisse des Verfahrens.

Manchmal wird eine proportionale Temperaturanpassungsfunktion benutzt, zum Beispiel $T(t+1) = \alpha * T(t)$ wobei α eine Konstante ist. Typische Werte für α in der Praxis liegen zwischen 0,8 und 0,99. Zu beachten ist, dass solch eine Funktion zu kleineren Senkungen der Temperatur führt je näher die Temperatur bei Null ist.

Ein anderes einfaches Schema hält $N(t)$, die Anzahl der durchsuchten Nachbarn pro Temperaturlevel, konstant oder setzt es proportional zur Größe des Problems oder der verwendeten Nachbarschaft.

```
1  Wähle eine Initiallösung x aus dem Lösungsraum;
2  Wähle eine Initialtemperatur T > 0;
3  Setze den Temperaturwechselzähler t = 0;
4  Wiederhole{
5    Setze den Temperaturwiederholungszähler n = 0;
6    Wiederhole{
7      Erzeuge Lösung y als Nachbar von x;
8      Berechne Δ = f(y) − f(x);
9      Wenn Δ < 0 dann
10       x := y;
11     sonst wenn zufall(0,1) < exp(−Δ/T) dann
12       x := y;
13     n := n + 1;
14   } bis n = N(t);
15 t := t + 1;
16 T := T(t);
17 } bis Abbruchkriterium erfüllt
```

Abbildung 3.5: Pseudo-Code des Simulated Annealing Verfahrens

Zusammengefasst ergeben sich folgende Parameter die bei SA gesetzt werden können:

- Wahl der Initiallösung
- Wahl der Initialtemperatur

- Anzahl der Wiederholungen innerhalb eines Temperaturlevels
- Art der Nachbarschaft
- Temperaturanpassungsschema

Modifikationen

Es gibt einige Möglichkeiten die Standardvorgehensweise des Simulated Annealing Verfahrens zu verändern/verbessern. Einfache Varianten beinhalten die beste bisher erzielte Lösung zu speichern, die Nachbarn ohne "zurücklegen" auszusuchen und alternative Akzeptanzwahrscheinlichkeiten zu wählen. Komplexere Varianten sind unter anderem SA mit anderen Methoden zu kombinieren, zum Beispiel mit anderen Metaheuristiken, und parallele Versionen für verteilte Computerarchitekturen.

3.4 Ergänzungen für stochastische Probleme

Geht man von einem stochastischen Modell aus, müssen Anpassungen in der Art und Weise vorgenommen werden, wie Lösungen verglichen werden. Abschnitt 3.4.1 erläutert die notwendigen Veränderungen und ihre Hintergründe. Abschnitt 3.4.2 zeigt das Verfahren wie in dieser Arbeit zwei stochastische Lösungen verglichen werden.

3.4.1 Erörterungen zum Vergleich von Zufallsvariablen

Bei allen Verfahren, welche iterativ nach besseren Lösungen suchen, ist der Vergleich zweier Lösungen eine zentrale Komponente (siehe beispielsweise Akzeptanz in Abschitt 3.3.1.3 und Akzeptanzfunktion in Abschnitt 3.3.2). Für deterministische Probleme ist jeder Auftragsfolge x_i ein bestimmter Zielfunktionswert $z_i = f(x_i)$ zugeordnet. Lösung x_j ist besser als Lösung x_i, und wird als derzeitige Lösung angenommen, wenn $z_i > z_j$.

In einem stochastischen Modell werden die z_i jedoch zu einer Zufallsvariablen und eine k-malige Auswertung von $f(x_i)$ liefert die k Stichprobenwerte z_{ik} der

Grundgesamtheit aller möglichen Ausprägungen, welche z_i annehmen kann. Diese Grundgesamtheit besitzt den Mittelwert $\mathbb{E}[z_i] = \mu_i$ und die Varianz $\mathrm{Var}(z_i) = \sigma_i^2$ (Ross, 2006). Es muss nun, analog zu $z_i > z_j$ im deterministischen Fall, eine Ordnungsrelation für Zufallsvariablen gefunden werden. Bei Optimierungsexperimenten wird hierfür oft die so genannte "Ordnung im Mittel[1]" verwendet. Zum Vergleichen von zwei Auftragsfolgen x_i und x_j werden somit die jeweiligen Mittelwerte μ_i und μ_j herangezogen.

Die unbekannte und wahrscheinlich immense Größe der Grundgesamtheit aller Möglichen Ausprägungen von z_i schliessen jedoch eine Vollerhebung und eine damit verbundene exakte Berechnung von μ_i aus. Aus diesem Grund wird zur Teilerhebung gegriffen. Dies bedeutet, dass anhand einer Stichprobe der Größe n von Ausprägungen des Zielfunktionswertes versucht wird, den Parameter μ_i zu schätzen. Als erwartungstreuer Schätzer für μ_i kann der Stichprobenmittelwert $\bar{Z}_i = \frac{1}{n}\sum_{k=1}^{n} z_{ik}$ heran gezogen werden (Bleymüller u. a., 2000).

Dieser Stichprobenmittelwert $\bar{Z}_i$ ist jedoch selbst wieder eine Zufallsvariable, da er bei verschiedenen Stichprobenziehungen unterschiedliche Werte annimmt, wodurch der einfache Vergleich $\bar{Z}_i > \bar{Z}_j$ wieder nicht sinnvoll ist. Durch den zentralen Grenzwertsatz ist jedoch bekannt, dass das Stichprobenmittel, wenn die Stichprobengröße 30 übersteigt und die zugrunde liegenden Stichprobenwerte als beliebig verteilte unabhängige Zufallsvariablen mit Mittelwert μ und Standardabweichung σ interpretiert werden können, annähernd normalverteilt mit Erwartungswert μ und Standardabweichung $\frac{\sigma}{\sqrt{n}}$ ist (Ross, 2006)[2].

Diese Erkenntnis mit dem zentralen Grenzwertsatz ist wegen 2 Gründen wichtig:

1. Der Mittelwert der Grundgesamtheit des Stichprobenmittels und der Mittelwert der Grundgesamtheit der Stichprobenwerte sind gleich mit Wert μ. Dies ist wichtig da somit eine Aussage, zum Beispiel mit einem statistischen Test, über den Mittelwert der Grundgesamtheit des Stichprobenmittels gleichzeitig eine Aussage über den Mittelwert der Grundgesamtheit der

[1] Stochastische Ordnung, Wikipedia, 2009, http://de.wikipedia.org/w/index.php?title=Stochastische_Ordnung, Online abgerufen am 12. März 2009

[2] siehe außerdem: T-Test, Wikipedia, 2009, http://de.wikipedia.org/w/index.php?title=T-Test, Online abgerufen am 12. März 2009

Stichprobenwerte ist.

2. Das Stichprobenmittel ist normalverteilt, weshalb es möglich ist mit einem Mitglied der Gruppe der t-Tests eine statistische Aussage über den Unterschied zwischen zwei Stichprobenmittelwerten zu machen.

3.4.2 Der stochastische Grösser-Gleich-Operator

Im letzten Abschnitt wurde festgestellt, dass bei ausreichender Stichprobengröße der Mittelwert der Grundgesamtheit des Stichprobenmittels gleich dem Mittelwert der Grundgesamtheit der Stichprobenwerte ist und dass das Stichprobenmittel normalverteilt ist. Dieses Wissen soll nun genutzt werden, um mit zwei Stichproben z_{1k} und z_{2k} mit k = 1, ..., n (wobei n>30) der Zufallsvariablen z_1 und z_2 der Zielfunktion eine Aussage darüber zu machen ob $\mathbb{E}[z_1] = \mu_1 > \mathbb{E}[z_2] = \mu_2$, also ob die Lösung x_1 einen schlechteren mittleren Zielfunktioneswert hat als Lösung x_2. Die Stichprobenmittel für die Stichproben der Zielfunktionswerte von x_1 und x_2 sind $\bar{Z}_1$ bzw. $\bar{Z}_2$.

Zur Anwendung kommt der Zweistichprobentest für die Differenz zweier Mittelwerte bei unbekannter und unterschiedlicher Varianz der Grundgesamtheiten (Bleymüller u. a., 2000). Dieser Test soll die Frage beantworten, ob von der beobachteten Differenz der Stichprobenmittel $c = \bar{Z}_1 - \bar{Z}_2$ auf eine Differenz der Mittelwerte der Grundgesamtheit $\mu_1 - \mu_2$ geschlossen werden kann.

Für diesen Test ist die Prüfgröße T in Formel 3.7 angeführt, wobei $S^2_{Z_i}$ die Stichprobenvarianz passend zum i-ten Stichprobenmittel ist. Wenn die Nullhypothese H_0: $\mu_1 - \mu_2 = c$ ist und die Alternativhypothese H_1: $\mu_1 - \mu_2 > c$ dann wird H_0 abeglehnt, wenn $T > t_{f;1-\alpha}$, wobei f in Formel 3.8 beschrieben ist.

$$T = \frac{\bar{Z}_1 - \bar{Z}_2 - c}{\sqrt{\frac{S^2_{Z_1}}{n} + \frac{S^2_{Z_2}}{n}}} \tag{3.7}$$

$$f \approx \frac{(\frac{S^2_{Z_1}}{n} + \frac{S^2_{Z_2}}{n})^2}{\frac{1}{n-1}(\frac{S^2_{Z_1}}{n})^2 + \frac{1}{n-1}(\frac{S^2_{Z_2}}{n})^2} \tag{3.8}$$

Die konkrete Annahme in dieser Arbeit ist, dass $c = 0$ ist und somit die verwen-

deten Stichproben keinen statistischen Hinweis darauf geben, dass die Mittelwerte der Grundgesamtheit unterschiedlich sind ($H_0 : \mu_1 = \mu_2$). Die Alternativhypothese besagt, dass der Mittelwert der 1. Grundgesamtheit größer ist als jener der 2. Grundgesamtheit ($H_1 : \mu_1 > \mu_2$). Bei einem Signifikanzniveau von $\alpha = 0.05$ bedeutet eine Ablehnung von H_0, dass eine Wahrscheinlichkeit kleiner 5% besteht, dass die Nullhypothese abgelehnt wurde, obwohl sie den richtigen Sachverhalt dargestellt hat und die Abweichung nur dem Zufall zuzuschreiben war.

Gilt es nun, zum Beispiel für VNS (Abbildung 3.4, Zeile 18, $x_2 = x^{'}$, $x_1 = x$), zwei Lösungen miteinander zu vergleichen, dann stellt x_2 eine signifikante Verbesserung gegenüber x_1 dar, wenn der verwendete einseitige t-Test ergibt, dass die Nullhypothese abzulehnen ist. Diese Vorgehensweise wird hier als stochastischer Grösser-Gleich-Operator bezeichnet.

Die Anwendung bei SA funktioniert analog (Abbildung 3.5, Zeilen 8 und 9, $x_2 = y$, $x_1 = x$), es muss jedoch zusätzlich die Akzeptanzfunktion (Abbildung 3.5, Zeile 11) modifiziert werden. Das Δ in Zeile 8 ist im stochastischen Fall gleich $\bar{Z}_2 - \bar{Z}_1$ und die Lösung x_2 wird in Zeile 9 als signifikant besser akzeptiert, wenn der verwendete t-Test ergibt, dass die Nullhypothese abzulehnen ist. Diese Vorgehensweise bringt mit sich, dass Lösungen nicht als Verbesserungen angesehen werden, die im Mittel zwar besser sind, aber keine signifikante Verbesserung versprechen. In diesem Fall ist Δ negativ, wodurch $exp(-\Delta/T)$ in Zeile 11 immer eine Zahl grösser 1 liefert. Dadurch würden alle Lösungen akzeptiert, die im Mittel aber nicht signifikant besser sind. Um dies zu verhindern wurde die Akzeptanzfunktion in $exp(-|\Delta|/T)$ umgeändert, wodurch wieder nur positive Δ auftreten.

Kapitel 4

Umsetzung des Problems als Software

Dieses Kapitel beschreibt die Umsetzung des gegebenen Modells in ein Computerprogramm um Vergleiche zwischen den einzelnen Algorithmen durchführen zu können. Abschnitt 4.1 beschreibt wie das gegebene Maschinenmodell mit Hilfe der Simulationssoftware AnyLogic abgebildet wurde. Abschnitt 4.2 zeigt die Umsetzung der Optimierungsalgorithmen.

4.1 Berechnung des Belegungsplanes

In Abschnitt 2.2.1.4 wurde bereits dargelegt, dass es bei komplizierten Modellen der Reihenfolgeplanung keine einfache Möglichkeit der Berechnung des Belegungsplanes gibt. Aus diesem Grund greift man zu vorgefertigten Simulationsumgebungen wie AnyLogic (Xjtek, 2005b), da solche Umgebungungen viele Komponenten zur Erstellung einer Ablaufsimulation für das gegebene Modell bereits beinhalten.

AnyLogic[1] ist eine Simulationsumgebung um unter anderem diskrete Ablaufsimulationen (eng. discrete event simulations) zu erstellen. Die graphische Oberfläche erlaubt es Modelle intuitiv und schnell zu erstellen. Gleichzeitig ist es möglich komplexe Erweiterungen mit Hilfe der Programmiersprache Java vorzunehmen.

[1] AnyLogic, XJ Technologies Company Ltd., http://www.xjtek.com/

Um dem Leser einen groben Überblick über den Fluss der Aufträge durch das Modell, unter Vernachlässigung von technischen Details, zu geben soll dieser, anhand der graphischen Repräsentation der *Active Object Class (AOC) Main* (Abbildung 4.1) des AnyLogic Modells erläutert werden. Die in dieser Abbildung verwendeten Komponenten lassen sich direkt dem in Abbildung 2.10 vorgestellten Modell zuordnen. Jene Komponenten, welche die erste Stufe darstellen werden durch das Rechteck mit der Beschriftung *Stage1* hervorgehoben und jene der Stufe zwei durch das Rechteck mit der Beschriftung *Stage2*.

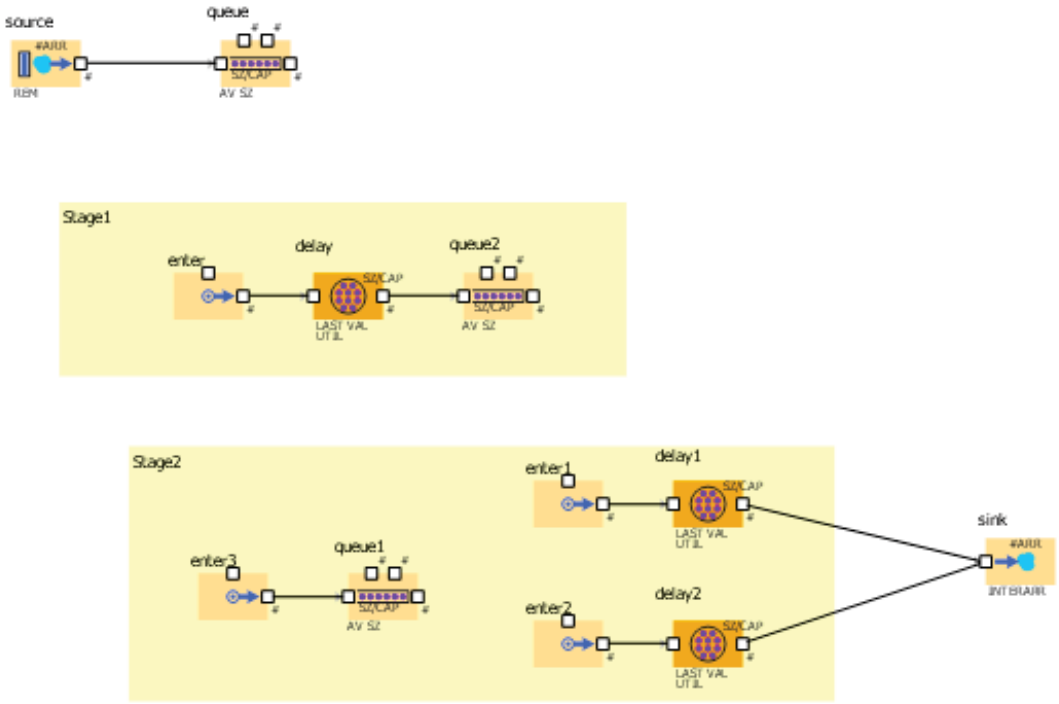

Abbildung 4.1: Active Object Class Main - Graphische Ansicht

Aufträge werden in der Komponente „source“[2]generiert und in „queue“ gelagert. In einer vorgegebenen Reihenfolge werden sie an „enter“ in Stufe 1 (Rechteck mit Markierung *Stage1*) weitergegeben und in „delay”(entspricht der Maschine 1) bearbeitet. Nach Fertigstellung im „delay“ wird der Auftrag in „queue2“ gelagert (während dieser Zeit ist „delay“ blockiert), bis ein Platz in „queue1“ in Stufe 2 (Rechteck mit Markierung *Stage2*) frei ist. „queue1“ repräsentiert den Puffer zwischen den Stufen.

[2]Wörter in Anführungszeichen (z.B. „queue“) sind Namen von Komponenten in Abbildung

In Stufe 2 werden die Aufträge nach dem First-In-First-Out-Verfahren (FIFO) verteilt. Es wird immer jener Auftrag, welcher in „queue1“ an erster Stelle steht der nächsten frei werdenden Maschine (also „delay1“ über „enter1“ (Maschine2) oder „delay2“ über „enter2“ (Maschine3)) zugeordnet.

Sind die Aufträge in der zweiten Stufe fertig, gelangen sie in „sink“. Dort wird ihre Fertigstellungszeit vermerkt und danach werden sie entsorgt[3].

Die folgenden Unterabschnitte beschreiben den zuvor oberflächlich angerissenen Arbeitsablauf im Detail. Abschnitt 4.1.1 beschreibt die Erzeugung der Aufträge, das Vermerken der Zielfunktionswerte und das Entsorgen der Aufträge. Abschnitt 4.1.2 beschreibt die Abläufe in Stufe 1. Abschnitt 4.1.3 beschreibt die Abläufe in Stufe 2.

4.1.1 Allgemeine Programmteile

Ein Modell in AnyLogic beginnt in der *AOC Main*. In der Code-Ansicht dieser Klasse können im Abschnitt 'Startup code' Instruktionen angegeben werden, welche zu Beginn des Simulationslaufes ausgeführt werden und notwendige Initialisierungen für das Modell vornehmen. Für das konkrete Modell werden hier die Daten der Aufträge, die Auftragsfolge für die erste Maschine und die Größe des Puffers zwischen den Stufen eingelesen.

Für die Auftragsfolge, mit der die Maschine 1 belegt werden soll, besitzt die *AOC Main* eine Instanzvariable mit dem Namen sequenceM1, in welcher die Daten nach dem Einlesen gespeichert werden.

Erzeugung der Auftragsobjekte

Die Daten der Aufträge (Auftragsnummer, Bearbeitungszeiten auf den Maschinen) werden in der Klasse JobData gespeichert. Deren Aufbau ist in Quellcode 4.1 dargestellt. Die int Variable in Zeile 2 ist die Auftragsnummer des Auftrages (anhand derer der Auftrag identifiziert wird, z.B. wenn es um die Einteilung auf den Maschinen geht) und die Map in Zeile 3 beinhaltet Objekte vom Typ

4.1.

[3]Da wir um einen hohen moralischen und ethischen Standard bemüht sind es es selbstverständlich nicht mehr benötigte Simulationsentitäten mit dem ihnen gebührenden Respekt zu behandeln und sie den Rest ihres digitalen Lebens in würdiger Art und Weise artgerecht ausklingen zu lassen!

Integer als Schlüssel und Objekte vom Typ Double als Werte. Der Schlüssel entspricht einer Maschinennummer und der zugehörige Wert der besten Schätzung der Bearbeitungsdauer auf dieser Maschine (siehe Abschnitt 4.1.2 für Details über die beste Schätzung der Bearbeitungsdauer in Zusammenhang mit stochastischen Modellen und die Verwendung als Bearbeitungszeit).

Quellcode 4.1: Klasse JobData

```
1 public class JobData {
2   public int jobID;
3   public Map processingTimes = new HashMap();
4
5   public String toString() {
6     return "ID: "+jobID+" PTimes:"+processingTimes+"\n";
7   }
8 }
```

Die Befüllung der JobData-Objekte wird in Quellcode 4.2 gezeigt. In Zeile drei wird der Map processingTimes ein Eintrag mit Schlüssel 1 und Wert 1 hinzugefügt. Dies bedeutet, dass dieser Auftrag auf der ersten Maschine eine Bearbeitungszeit von 1 hat. Beim Einlesen bekommt AnyLogic eine Liste von JobData-Objekten und speichert sie in der Instanzvariable jobsData der *AOC Main*.

Quellcode 4.2: Befüllung der Klasse JobData

```
1 JobData j1 = new JobData();
2 j1.jobID = 1;
3 j1.processingTimes.put(new Integer(1), new Double(1));
4 j1.processingTimes.put(new Integer(2), new Double(2));
5 j1.processingTimes.put(new Integer(3), new Double(3));
6 jobsData.add(j1);
```

Am Beginn der Simulation erzeugt die Komponente „source“ so viele Entitäten der Klasse Job (siehe Quellcode 4.3), wie die Instanzvariable jobsData der *AOC Main* vorgibt. Die Entitäten werden danach alle in „queue“ gelagert.

Um Warteschlangen (queues) und Maschinen (delays) darzustellen, wird die AnyLogic Enterprise Library (Xjtek, 2005a) verwendet. Damit eine Klasse dort als Objekt, das durch die Anordnung der Komponenten fließt, verwendet werden kann, muß sie von der Klasse Entity abgeleitet werden, so auch die Klasse Job (Zeile 1).

In den Einstellungen von Quellkomponenten (wie „source“ in Abbildung 4.1) kann mit dem Parameter newEntity angegeben werden, welche Klasse für die Generierung der Objekte verwendet werden soll, mit interarrivalTime wie viel

Zeit zwischen der Generierung zweier Objekte liegen soll und mit arrivalsMax wie viele Objekte generiert werden sollen. In „source“ hat newEntity den Wert Job.class, interarrivalTime der Wert 0 und arrivalsMax die Länge von jobsData als Wert. Zusammengefasst bedeutet dies, dass „source“ arrivalsMax Job-Ojekte zu Beginn auf einmal erzeugt.

Quellcode 4.3: Klasse Job

```
1  public class Job extends Entity {
2    public int jobID;
3    public Map processingTimes = new HashMap();
4
5    public Job() {
6      Main mainClass = (Main) Engine.getRoot();
7
8      int numberOfInstanciatedJobs =
9        mainClass.numberOfInstanciatedJobs;
10
11     JobData jd =
12       (JobData) mainClass.jobsData.get(numberOfInstanciatedJobs);
13
14     mainClass.numberOfInstanciatedJobs++;
15
16     this.jobID           = jd.jobID;
17     this.processingTimes = jd.processingTimes;
18   }
19
20   public double getProcessingTimeForMachine(int machine) {
21     Object pTim = processingTimes.get(new Integer(machine));
22     return ((Double) pTim).doubleValue();
23   }
24
25   public String toString() {
26     return "ID: "+jobID+" PTimes: "+processingTimes+"\n";
27   }
28 }
```

Jedes Mal wenn „source“ ein Job-Objekt erzeugt, wird der Konstruktor der Klasse Job (Quellcode 4.3 Zeilen 5 bis 18) aufgerufen. Um die aktuelle Anzahl schon instanzierter Job-Objekte zu vermerken besitzt die *AOC Main* das Feld numberOfInstanciatedJobs. In Zeile 6 wird eine Referenz auf die *AOC Main* geholt und in Zeile 8 das Attribut numberOfInstanciatedJobs ausgelesen. In Zeile 11 wird aus der jobsData Liste, welche die Ausgangsdaten für die Aufträge enthält, der Auftrag an der numberOfInstanciatedJobs-ten Stelle geholt. In Zeile 14 wird numberOfInstanciatedJobs in der *AOC Main* um 1 erhöht um zu vermerken, dass ein weiterer Auftrag instanziert wurde. In den Zeilen 16 und 17 werden die

Daten aus dem JobData-Objekt in die Felder der Job-Objektes transferiert.

Die Methode getProcessingTimeForMachine in Zeile 20 von Quellcode 4.3 liefert für die übergebene Maschinenkennzahl den korrespondierenden Eintrag aus der processingTimes Map, also die beste Schätzung der Bearbeitungszeit des Auftrages auf dieser Maschine.

Zielfunktionen

Wenn die Aufträge in der 2. Stufe fertig sind, also aus „delay1“ oder „delay2“ kommen, werden sie an „sink“ weitergeleitet. Jedes Mal wenn ein Job in „sink“ ankommt wird die Funktion sinkIt(entity) (Quellcode 4.4) aufgerufen, da diese im Parameter onEnter der Komponente angeführt ist. Der Parameter entity dieser Funktion enthält den soeben angekommenen Auftrag.

Quellcode 4.4: Funktion in sinkIt();

```
1  public void sinkIt(Entity entity) {
2    Integer jobId = new Integer(((Job) entity).jobID);
3    tmpCompletionTimes.put(jobId, new Double(getTime()));
4
5    finishedJobsCounter++;
6
7    if(finishedJobsCounter == numberOfJobs) {
8     saveCompletionTimes(tmpCompletionTimes);
9     saveBufferLevel(queue1.getStatsSize().mean()/bufferSize);
10   }
11 }
```

Ist ein Auftrag in „sink“ angekommen dann ist der Produktionsprozess für ihn abgeschlossen, womit der Zeitpunkt an dem er ankommt der Fertigstellungszeit entspricht. Diese wird in der Variable tmpCompletionTimes (Zeile 3) vermerkt, die eine Map ist, in der als Schlüssel die Auftragsnummer und als Wert dieser Zeitpunkt vermerkt wird.

Der Produktionsprozess ist ganz abgeschlossen wenn der letzte Auftrag in „sink“ angekommen ist. Aus diesem Grund wird bei jedem Eintreffen eines Auftrages die Variable finishedJobsCounter (Zeile 5) um eins erhöht. Ist der Wert in dieser Variable gleich der Anzahl der Aufträge (Zeile 7), dann werden in Zeile 8 und 9 die Fertigstellungszeiten der Aufträge und der durchschnittliche Pufferstand gespeichert.

Der durchschnittliche Pufferstand (Zeile 9) wird berechnet, indem die durchschnittliche Länge von „queue1“ (queue1.getStatsSize().mean()) durch die Größe

des Puffers (bufferSize) dividiert wird. Da das Vermerken und Berechnen der durchschnittlichen Länge von „queue1“ sehr aufwendig sein kann, ist es von Haus aus nicht eingeschaltet. Es wird aktiviert indem man in der queue-Komponente das Feld statsEnabled auf true setzt. Dann ist es möglich mit dem Aufruf von queue1.getStatsSize().mean(); die mittlere Größe der Queue-Komponente auszulesen, was dem mittleren Füllstand des zu minimierenden Puffers entspricht. Durch die Größe des Puffers wird dividiert um immer einen normierten prozentuellen Füllstand zu bekommen.

In den vorangegangen Absätzen wurden die Fertigstellungszeiten und der durchschnittliche Pufferstand für die Berechnung der korrespondierenden Zielfunktionen vermerkt. Die verbleibende dritte Zielfunktion, die Auslastung von Maschine 1, wird an einer anderen Stelle im Programmcode ermittelt. Da die Messung der Auslastung beendet wird, wenn der letzte Auftrag die Stufe 1 verlässt, ist der zuständige Programmcode in der Funktion transfer2Stage2(); (siehe Abschnitt 4.1.2) untergebracht. Auch hier registriert eine Zählervariable jeden passierenden Auftrag und wenn diese Variable der Gesamtanzahl der zu verarbeitenden Aufträge entspricht dann wird mit delay.getStatsUtilization().mean(); die durchschnittliche Auslastung von Maschine 1 vermerkt.

Aus den selben Gründen wie bei der Statistik für die durchschnittliche Länge von „queue1“, ist auch die Statistik für die durchschnittliche Auslastung von „delay“ (entspricht Maschine 1) ausgeschaltet. Um sie einzuschalten, ändert man den Parameter statsEnabled auf true und mit delay.getStatsUtilization().mean(); kann die mittlere Auslastung ausgelesen werden.

Die Auslastung wird ab jenem Zeitpunkt gemessen, an dem der erste Auftrag auf der 1. Maschine begonnen wird, bis zu jenem Zeitpunkt, an dem der letzte Auftrag (der n-te bei n Aufträgen) die 1. Maschine verlässt. Alternativ wäre es möglich als Endzeitpunkt jenen zu nehmen, an dem der letzte Auftrag die 2. Stufe verlässt. Es kommt dann jedoch zu dem Phänomen, dass die 1. Maschine vom 1. bis zum n-ten Auftag durcharbeitet und trotzdem keine Auslastung von 100% hat. Dazu kommt es, da der n-te Auftrag die 1. Maschine verläßt und noch auf der 2. Stufe bearbeitet werden muss und diese Zeit würde dann auf der 1. Maschine als ungenützte Zeit verbucht. Erstere Formulierung wird als sinnvoller angesehen, da es das Ziel ist die Auslastung möglichst effizient (schnell) die Aufräge durch

die 1. Maschine zu bekommen.

Es kann nun der Fall auftreten, dass die Maschine eine Auslastung von 100% hat, es jedoch zu einem durchschnittlichen Pufferstand von mehr als 0% kommt. Diese Situation ergibt sich, wenn der n-te Auftrag auf der 1. Maschine fertig ist und im Puffer warten muss. Es is dann zwar möglich die 1. Maschine optimal (im Sinne von keinem Stillstand auf der 1. Maschine) zu belegen, jedoch wird die Weiterverarbeitung auf der 2. Stufe noch nicht optimal (im Sinne von einem allzeit leeren Puffer) erledigt. Die Kennzahlen spiegeln also in diesem Fall das gewollte Verhalten wieder.

4.1.2 Implemetierung der 1. Stufe

Nachdem die Job Objekte für alle Aufträge zu Beginn der Simulation erzeugt wurden (siehe Abschnitt 4.1.1), befinden sie sich in „queue“. Die vom Benutzer vorgegebene Aufragsfolge für Maschine 1 („delay“) ist in der Liste sequenceM1 gespeichert. Die Aufträge müssen in dieser Reihenfolge an „delay“ weiter geleitet werden.

Dies wird durch die Funktion prioritizeAndSendJob(); und durch die Komponente „enter“ bewerkstelligt. Die Funktion prioritizeAndSendJob(); ist in Quellcode 4.5 angegeben. Sie wird im onEnter von „queue“ und in transfer2Stage2 (); aufgerufen.

Zuerst wird in Zeile 2 überprüft ob noch Job Objekte in „queue“ auf die Bearbeitung warten und ob die Maschine belegt ist. Die Variable indexOfNextJob in Zeile 4 soll später jene Position des nächsten zu bearbeitenden Auftrages beinhalten wo er in „queue“ zu finden ist. Falls dieser noch nicht in „queue“ ist, soll sie -1 sein. In den Zeilen 6 und 7 wird ermittelt welche Nummer der nächste zu bearbeitende Auftrag hat. Darüber, wie viele Aufträge bereits bearbeitet wurden, gibt die Variable numberOfProcessedJobs Auskunft. In den Zeilen 9 bis 13 wird „queue“ Platz für Platz auf den gesuchten Auftrag überprüft und falls dieser gefunden wurde seine Position vermerkt[4]. Falls der gesuchte Auftrag gefunden wurde, wird ab Zeile 15 vermerkt, dass ein weiter Auftrag bearbeitet wurde, dass die Maschine besetzt ist und, dass das Job Objekt mit queue.remove(); aus „queue“ entfernt und mit enter.take(); an „enter“ weitergegeben wird. Grundsätzlich verbinden Enter-

blöcke Teile des Modells welche die Enterprise Library (Xjtek (2005a)) benutzen mit selbst erzeugten Teilen. Außerdem ist es möglich mit enter.take(Entity e); einem Enter-block eine beliebige Entität, hier Aufträge, zu übergeben.

Quellcode 4.5: Funktion prioritizeAndSendJob();

```
1  public void prioritizeAndSendJob() {
2   if(queue.size()>0 && !machineOccupied) {
3
4    int indexOfNextJob = -1;
5
6    Object jobId=sequenceM1.get(numberOfProcessedJobs);
7    int numberOfNextJob = ((Integer)jobId).intValue();
8
9    for(int i=0; i<queue.size(); i++) {
10     Job j1 = (Job) queue.get(i);
11     if(j1.jobID == numberOfNextJob)
12           indexOfNextJob = i;
13   }
14
15    if(indexOfNextJob != -1) {
16     numberOfProcessedJobs++;
17     machineOccupied = true;
18     enter.take(queue.remove(indexOfNextJob));
19    }
20   }
21 }
```

Sobald ein Job Objekt ins „enter“ gesetzt wurde, wandert es automatisch ins „delay“ weiter. Wie lange es dort bleibt, wird über den Parameter delayTime der delay-Komponente entschieden, der mit processingTime(Job j1, int mId); gefüllt ist. In der Funktion (siehe Quellcode 4.6) wird die beste Schätzung der Bearbeitungsdauer des übergebenen Auftrages in Zeile 3 ausgelesen und in Zeile 5 nach Formel 2.4 modifiziert.

Sobald der Auftrag in „delay“ abgearbeitet wurde, wird er automatisch an „queue2“ weitergeleitet da AnyLogic vorschreibt, dass ein Objekt nach verstreichen seiner Bearbeitungszeit eine delay-Komponente verlassen muss. Die Maschine bleibt jedoch blockiert, da sie laut dem Modell erst wieder frei ist, wenn der Auftrag im Puffer („queue1“) angekommen ist.

[4]Prinzipiell ist die Funktion zu mächtig für die hier getätigte Annahme, dass alle Aufträge zum Zeitpunkt 0 verfügbar sind. Dies ist jedoch gewollt, da keine Änderungen beim Wegfall dieser Annahme vorzunehmen sind.

Quellcode 4.6: AOC Machine - processingTime(Job j1)

```
1 double processingTime(Job j1, int mId)
2 {
3   double pBestEst = j1.getProcessingTimeForMachine(mId);
4
5   double pActual = (1 + pTimesExperimental * Machine.uniform(-1,1))
        * pBestEst;
6
7   return pActual;
8 }
```

Im Parameter onEnter von „queue2“ ist die Funktion transfer2Stage2(); eingetragen die aufgerufen wird, sobald ein Objekt in der Komponente ankommt. Diese Funktion (siehe Quellcode 4.7) überprüft zuerst, ob der Puffer („queue1“) voll ist und ob überhaupt ein Objekt in „queue2“ vorhanden ist (Zeile 2). Letzteres ist notwendig, da die Funktion auch aufgerufen wird, wenn ein Auftrag den Puffer verlässt (Funktion prioritizeAndSendJobStage2();, siehe Abschnitt 4.1.3), um zu überprüfen, ob ein Auftrag darauf wartet, nachrücken zu können. Bei erfolgreicher Überprüfung wird der Auftrag der in „queue2“ wartet in „enter3“ gesteckt (Zeile 3), die Maschine als verfügbar markiert (Zeile 4) und versucht den nächsten Auftrag an die Maschine 1 („delay“) zu senden (Zeile 5).

Quellcode 4.7: Funktion transfer2Stage2();

```
1  public void transfer2Stage2() {
2   if(queue1.size()<pufferSize && queue2.size()>0) {
3     enter3.take(queue2.remove(0));
4     machineOccupied = false;
5     prioritizeAndSendJob();
6
7     finishedJobsCounterM1++;
8     if(finishedJobsCounterM1==numberOfJobs)
9       saveUtilization(delay.getStatsUtilization().mean());
10  }
11 }
```

Der zweite Teil der Funktion ab Zeile 7 beschäftigt sich mit der schon weiter oben erwähnten Zielfunktion 'Auslastung von Maschine1'. Wie in Abschnitt 4.1.1 beschrieben, soll die Auslastung vom Einlangen des ersten Auftrages auf Maschine 1 bis zur Vollendung des letzten Auftrages gemessen werden. Somit ist der Startzeitpunkt für die Messung bei 0 da vorgegeben ist, dass alle Aufträge zu Beginn vorhanden sind. Der Endzeitpunkt wird ermittelt, indem eine Zählervariable jedes Mal um 1 inkrementiert wird, wenn ein Auftrag „queue2“ verlassen

kann (Zeile 7). Entspricht der Wert der Zählervariablen der Anzahl der Aufträge, dann ist der letzte Auftrag erreicht (Zeile 8) und mit saveUtilization (); wird die durchschnittliche Auslastung von „delay“ (delay. getStatsUtilization (). mean();) gespeichert (Zeile 9).

4.1.3 Implemetierung der 2. Stufe

Im letzten Abschnitt wurden die Aufträge an „enter3“ geschickt, wenn die Kapazitätsgrenze von „queue1“ noch nicht erreicht ist. Die *AOC Main* besitzt die Variable pufferSize, welche bei der Eigenschaft 'capacity' der Komponente „queue1“ verwendet wird und die Größe dieser Warteschlange (dieses Puffers) festlegt. Die in der Sektion 'Startup code' der *AOC Main* eingelesene Größe für den Puffer wird mittels set_pufferSize (bufferSize ()); gesetzt.

Quellcode 4.8: Funktion prioritizeAndSendJobStage2();

```
1  public void prioritizeAndSendJobStage2() {
2   if(queue1.size()>0) {
3     if(!machineTWOOccupied && !machineTHREEOccupied) {
4       Job jcur = ((Job) queue1.get(0));
5       double timeOnM2 = jcur.getProcessingTimeForMachine(2);
6       double timeOnM3 = jcur.getProcessingTimeForMachine(3);
7       if(timeOnM2<timeOnM3) {
8         machineTWOOccupied = true;
9         enter1.take(queue1.remove(0));
10        transfer2Stage2();
11      }
12      else {
13        machineTHREEOccupied = true;
14        enter2.take(queue1.remove(0));
15        transfer2Stage2();
16      }
17    }
18    else if(!machineTWOOccupied) {
19      machineTWOOccupied = true;
20      enter1.take(queue1.remove(0));
21      transfer2Stage2();
22    }
23    else if(!machineTHREEOccupied) {
24      machineTHREEOccupied = true;
25      enter2.take(queue1.remove(0));
26      transfer2Stage2();
27    }
28  }
29 }
```

Erreicht ein Auftrag „queue1“, wird prioritizeAndSendJobStage2(); aufgerufen, da diese im onEnter Parameter der Komponente eingetragen ist. Die Funktion (siehe Quellcode 4.8) teilt den Auftrag nach bestimmten Kriterien (siehe unten) der 2. und 3. Maschine („delay1“ und „delay2“) zu.

In Zeile 2 wird überprüft ob in „queue1“ ein Auftrag liegt. Dies ist notwendig da die Funktion auch aufgerufen wird, wenn ein Auftrag eine der beiden Maschinen verlässt. Gibt es einen Auftrag zum Weiterleiten, kann zwischen 3 Fällen unterschieden werden: i) es sind beide Maschinen frei (Zeile 3), ii) es ist nur die Maschine 2 frei (Zeile 18) oder iii) es ist nur die Maschine 3 frei (Zeile 23).

Im Fall i) bei welchem beide Maschinen frei sind ist die Vorgabe, dass jene Maschine gewählt wird auf welcher der Auftrag die kürzere Bearbeitungszeit hat. Um dies durchzuführen werden die Zeiten auf den beiden Maschinen ermittelt (Zeilen 4 bis 6). Falls die Bearbeitungszeit auf Maschine 2 kürzer ist als jene auf Maschine 3 (Zeile 7) wird Maschine 2 als besetzt markiert (Zeile 8), der Auftrag an „enter1“ weitergeleitet (Zeile 9) und transfer2Stage2() aufgerufen (Zeile 10), um einen in „queue2“ wartenden Auftrag die Möglichkeit zu geben in Stufe 2 zu kommen. Falls die Bearbeitungszeit auf Maschine 2 nicht kürzer ist als jene auf Maschine 3 werden die soeben beschriebenen Schritte für Maschine 3 ausgeführt.

Ist aus Zeile 3 bekannt, dass nicht beide Maschinen frei sind bleiben 2 Möglichkeiten: Maschine 2 ist frei oder Maschine 3 ist frei. In Zeile 18 wird überprüft ob Maschine 2 frei ist (Fall ii) und wenn dies der Fall ist, werden die oben beschriebenen Schritte für diese Maschine ausgeführt.

War auch die Maschine 2 nicht frei, wird in Zeile 23 überprüft ob Maschine 3 frei ist (Fall iii) und in diesem Fall werden die oben beschriebenen Schritte für diese Maschine ausgeführt.

Die Vorgangsweise für „delay1“ und „delay2“ sind im Prinzip ident, weshalb hier nur „delay1“ beschrieben wird. Ist ein Auftrag bei „delay1“ angelangt wird mit der Funktion processingTime(Job j1, int mId); (siehe Quellcode 4.6) seine Verweildauer in der Komponente bestimmt, da diese im Parameter delayTime angegeben ist. Ist die Maschine fertig, wird prioritizeAndSendJobStage2(); aufgerufen, um einen in „queue1“ wartenden Auftrag die Möglichkeit zu geben einer Maschine zugewiesen zu werden. Außerdem wird die Maschine als frei markiert.

4.2 Umsetzung der Optimierungsalgorithmen

Es ist prinzipiell möglich alle benötigten Teile des Programmes innerhalb der AnyLogic-Oberfläche zu programmieren. Dies hat jedoch (zumindest) zwei Nachteile. Die AnyLogic-Programmoberfläche ist keines Falls eine vollwertige Entwicklungsumgebung und AnyLogic arbeitet intern mit einer älteren Java-Version. Es ist daher sinnvoller alle Teile, welche nicht direkt mit dem in AnyLogic umgesetzten Modell zu tun haben, extern zu entwickeln. In diesem Sinn wurden die Optimierungsalgorithmen und einige andere Hilfsklassen in der Entwicklungsumgebung Eclipse (Version 3.4.1) mit der Java Version jdk1.6.0_07 erstellt.

Die Metaheuristiken VNS und SA wurden wie in Abschnitt 3.3 beschrieben umgesetzt. Als Nachbarschaften dienten die in Abschnitt 3.2.2.2 vorgestellten.

4.2.1 Organisatorischer Aufbau

Um die Klassen in organisatorische Einheiten aufzuspalten, wurden diese in 4 Pakete zusammengefasst (siehe Tabelle 4.1). Alle Paketnamen die verwendet werden beginnen mit *at.univie.pom.walz* um damit die Organisation, die Abteilung und das Projekt für das sie verwendet werden zu kennzeichen. Im Paket *...alpkg* sind Klassen enthalten, welche die Umgebung für die Algorithmen bieten. Das Paket *...alpkg.optialgos* beinhaltet Basisklassen aus welchen konkrete Algorithmen zusammengebaut werden. In *...alpkg.optialgos.noiterative* und *...alpkg.optialgos.iterative* sind verschiedene getestete konkrete Optimierungsalgorithmen enthalten. Ersteres beinhaltet Algorithmen die eine Lösung anhand der gegeben Daten berechnen und letzteres Algorithmen die iterativ nach besseren Lösungen suchen.

at.univie.pom.walz.alpkg
at.univie.pom.walz.alpkg.optialgos
at.univie.pom.walz.alpkg.optialgos.noiterative
at.univie.pom.walz.alpkg.optialgos.iterative

Tabelle 4.1: Java Pakete

Die Klassen in den Paketen *...alpkg.optialgos*,*...alpkg.optialgos.noiterative* und *...alpkg.optialgos.iterative* bilden die in Abbildung 4.2 gezeigte Vererbungshierarchie.

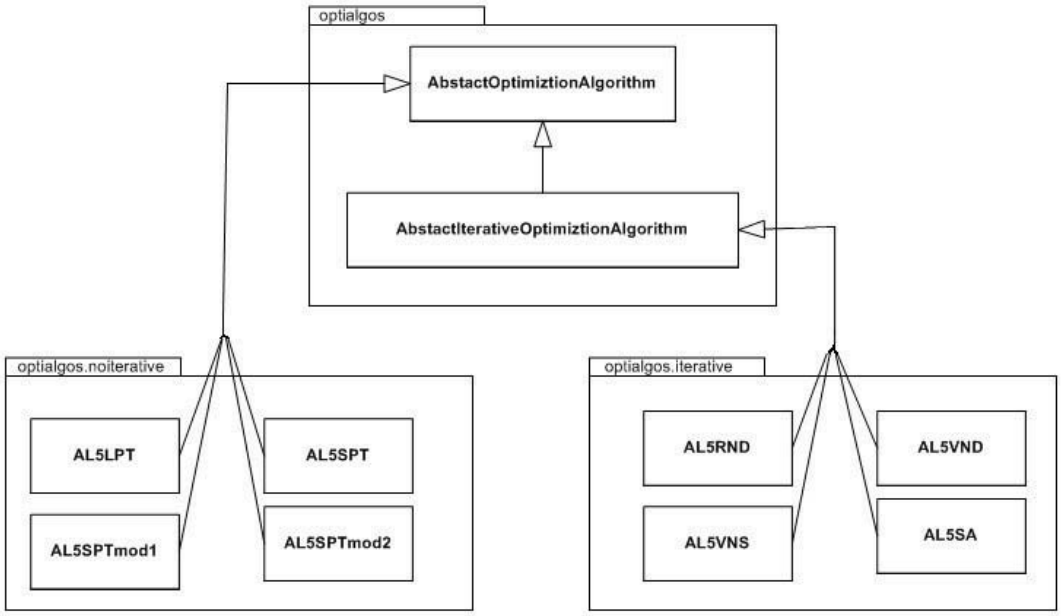

Abbildung 4.2: Vererbungshierarchie der Optimierungsklassen

Die Klasse AbstractOptimizationAlgorithm bildet den Stamm der Vererbungsstruktur. Hier sind Attribute und Methoden untergebracht welche von jedem Optimierungsalgorithmus benötigt werden. Die Klasse AbstractIterativeOptimizationAlgorithm ist von AbstractOptimizationAlgorithm abgeleitet und ergänzt diese um Elemente, welche notwendig sind für iterative Algorithmen.

Algorithmen, welche nach bestimmten Regeln (z.B. Prioritätsregeln) eine bestimmte Auftragsfolge festlegen und nur diese Auswerten, werden von AbstractOptimizationAlgorithm abgeleitet. Konkret wurden 4 bestimmte Algorithmen in 4 Klassen umgesetzt: SPT (Klasse AL5SPT), LPT (Klasse AL5LPT), und 2 Modifikationen von SPT (Klassen AL5SPTmod1 und AL5SPTmod2).

Jene Algorithmen, welche iterativ Lösungen auswerten und den Suchraum nach bestimmten Regeln durchsuchen, werden von AbstractIterativeOptimizationAlgorithm abgeleitet. Es wurden Zufallssuche (Klasse AL5RND), Variable Neighboorhood Descent (VND, Klasse AL5VND), Variable Neighboorhood Seach (VNS, Klasse AL5VNS) und Simulated Annealing (SA, Klasse AL5SA) umgesetzt. Die Verfahren VND, VNS und SA sind in Abschnitt 3.3 beschrieben.

4.2.2 Implementierung der Optimierungsklassen

Um einen Überblick über die Umsetzung der Optimierungsalgorithmen in Java zu geben, werden in den folgenden Abschnitten die Basisklassen und ausgewählte Algorithmen vorgestellt.

Abschnitt 4.2.2.1 beschreibt die Klasse AbstractOptimizationAlgorithm, die Basisklasse für alle Optimierungsalgorithmen, und Abschnitt 4.2.2.2 beschreibt die Klasse AbstractIterativeOptimizationAlgorithm, die Basisklasse für alle iterativen Optimierungsalgorithmen. Abschnitt 4.2.2.3 beschreibt als Beispiel für die Umsetzung einer Prioritätsregel die Klasse AL5SPT und Abschnitt 4.2.2.4 stellt als Beispiel für die Umsetzung eines iterativen Optimierungsalgorithmus die Klasse AL5VNS vor.

4.2.2.1 Klasse AbstractOptimizationAlgorithm

Die Klasse AbstractOptimizationAlgorithm versammelt Attribute und Methoden, welche von allen Optimierungsalgorithmen benötigt werden (Abbildung 4.3).

Sie besitzt 2 Attribute: bestSolution vom Typ AL5Solution und numberOfCheckedSolutions vom Typ int. Diese werden von jeder Art von Optimierungsalgorithmus benötigt, da die beste bisher gefundene Lösung aufbewahrt werden muss und es von Interesse ist, wie viele Lösungen ausgewertet wurden. Sie sind private deklariert, es kann auf sie jedoch mit getter und setter Methoden zugegriffen werden.

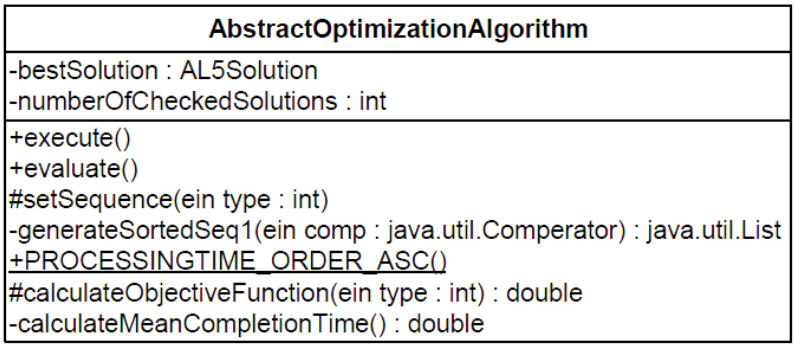

Abbildung 4.3: Detailiertes Klassendiagram von AbstractOptimizationAlgorithm

Die Methode execute ist abstract deklariert und wird von den ableitenden

Klassen mit den für den jeweiligen Algorithmus spezifischen Anweisungen gefüllt.

Die Methode evaluate() wertet eine gesetzte Reihenfolge aus, indem es die Simulation mit den Eingabewerten versorgt und sie dann startet. Diese Funktion ist protected deklariert weil die abgeleiteten Klassen darauf zugreifen können sollen, andere aber nicht. Am Ende der Funktion wird numberOfCheckedSolutions um eins erhöht um anzuzeigen, dass eine weitere Lösung überprüft wurde.

In der Methode setSequence wird nach einem bestimmten Verfahren die nächste zu evaluierende Lösung generiert. Der Parameter type gibt an, welches Verfahren zur Anwendung kommt. Er ist vom Typ int, es sind jedoch nur int-Werte zulässig die in der OptimizationAlgorithmConstants Klasse mit Konstanten definiert sind, sonst wird ein Fehler ausgegeben. Mögliche Werte sind beispielsweise 3 für das SPT-Verfahren oder 14 für die Modifikation 1 des SPT Verfahrens. Wird zum Beispiel 3, also SPT-Verfahren, als Parameter übergeben, dann generiert der Aufruf von generateSortedSeq1(PROCESSINGTIME_ORDER_ASC); eine neue Reihenfolge, welche für die Auswertung vermerkt wird. Alle hier möglichen Generierungsverfahren (SPT, LPT, SPTmod1, SPTmod2) lösen einen Aufruf von generateSortedSeq aus, da sie alle auf dem Sortieren der Reihenfolge nach bestimmten Eigenschaften des Auftrages beruhen (siehe nächster Absatz).

Die Methode generateSortedSeq1 bekommt einen Comperator von JobData Objekten übergeben (z.B. PROCESSINGTIME_ORDER_ASC aus dem vorherigen Absatz) und liefert eine Liste mit den Auftragsnummern welche nach den Vorgaben des Comperator sortiert sind (siehe Quellcode 4.9).

Quellcode 4.9: Funktion generateSortedSeq1();

```
1  private List<Integer> generateSortedSeq1(
2     Comparator<JobData> comperator) {
3   Collections.sort(jobDataList, comperator);
4
5   List tmpSeqM1 = new ArrayList();
6   for(JobData jd1 : jobDataList)
7     tmpSeqM1.add(new Integer(jd1.jobID));
8
9   return tmpSeqM1;
10 }
```

Ein Comperator ist eine Klasse, welche vorgibt, nach welchen Eigenschaften einer Klasse ihre Objekte verglichen werden. Die Methode Collections.sort() in Zeile 3 aus Quellcode 4.9 bekommt eine Liste von Objekten und einen für diese

Objekte passenden Comperator und sortiert die Liste dementsprechend. Quellcode 4.10 zeigt den schon mehrfach erwähnten Comperator PROCESSINGTIME_ORDER_ASC.

Quellcode 4.10: Comperator PROCESSINGTIME_ORDER_ASC

```
1  public static final
2    Comparator<JobData> PROCESSINGTIME_ORDER_ASC
3    = new Comparator<JobData>() {
4        public int compare(JobData jd1, JobData jd2) {
5         double pTime1=jd1.processingTimes.get(new Integer(1));
6         double pTime2=jd2.processingTimes.get(new Integer(1));
7
8         if(pTime1<pTime2)
9          return -1;
10        else if(pTime1>pTime2)
11         return +1;
12        else
13         return 0;
14     }
```

Zuerst wird die Bearbeitungszeit der beiden übergebenen Aufträge ausgelesen. Ist die Zeit des 1. Auftrages kleiner als jene des 2. dann wird -1 zurück gegeben, ist sie grösser wird +1 zurück gegeben und sind sie gleich dann wird 0 zurück gegeben. Will eine Methode wie Collections.sort() beim Sortieren wissen, wie zwei Aufträge zueinander stehen, ruft sie die Methode compare() des Comperators auf und je nachdem ob -1, 1 oder 0 zurück kommt, ist das erste Argument kleiner, grösser oder gleich dem Zweiten.

Quellcode 4.11: Funktion calculateObjectiveFunction();

```
1  protected double calculateObjectiveFunction(int type) {
2   if(type==OptimizationAlgorithmConstants.ZF_MEAN_COMPLETION_TIME) {
3     return calculateMeanCompletionTime();
4   }
5   else if(type==OptimizationAlgorithmConstants.ZF_MAKESPAN) {
6     return calculateMakespan();
7   }
8   else
9     throw new IllegalArgumentException("Argument not supported!");
10 }
```

Die Methode calculateObjectiveFunction() berechnet einen Zielfunktionswert für eine ausgewertete Lösung (siehe Quellcode 4.11). Der Parameter type gibt an, welche Zielfunktion berechnet werden soll. Er ist vom Typ int, es sind jedoch nur int-Werte zulässig die in der OptimizationAlgorithmConstants Klasse mit Konstanten definiert sind, sonst wird ein Fehler ausgegeben. Ein möglicher Wert ist

zum Beispiel 2 für die mittlere durchschnittliche Fertigstellungszeit, wodurch jene Funktion aufgerufen wird, welche diese Zielfunktion berechnen kann, in diesem Fall calculateMeanCompletionTime();.

4.2.2.2 Klasse AbstractIterativeOptimizationAlgorithm

Die Klasse AbstractIterativeOptimizationAlgorithm ist von der Klasse AbstractOptimizationAlgorithm abgeleitet und erweitert diese um Funktionalität, welche von iterativen Algorithmen benötigt wird (Abbildung 4.4).

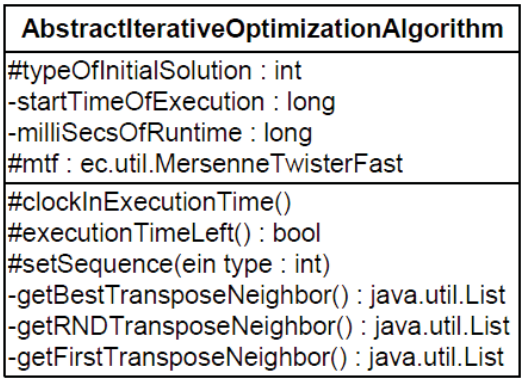

Abbildung 4.4: Klassendiagram von AbstractIterativeOptimizationAlgorithm

Ein iterativer Algorithmus startet stets von einer bestimmten Ausgangslösung aus, weshalb diese im Attribut typeOfInitialSolution angegeben werden muss. Diese hat den Typ int, es sind jedoch nur int-Werte zulässig die in der OptimizationAlgorithmConstants Klasse mit Konstanten definiert sind. Mögliche Werte sind zum Beispiel 1 für eine zufällig generierte Lösung und 3 für eine Lösung die nach dem SPT-Verfahren generiert wird.

In der hier umgesetzten Form der iterativen Algorithmen wird davon ausgegangen, dass das Abbruchkriterium eine gewisse Laufzeitgrenze ist. Um diese Laufzeit zu erfassen und zu überprüfen sind die Variablen startTimeOfExecution und milliSecsOfRuntime vorhanden. In ersterer wird vermerkt wann der Algorithmus zu rechnen begonnen hat und in zweiterer steht wie lange er rechnen darf.

Da bei manchen hier angewendeten iterativen Algorithmen die Entscheidung über die nächste zu überprüfende Lösung von Zufall abhängig gemacht wird, ist

es notwendig (pseudo) Zufallszahlen zu generieren. Als Generator wird eine Instanz der Klasse MersenneTwisterFast (Makoto und Takuji, 1998) verwendet, auf welche eine Referenz in der Variable mtf verweist.

In einem der vorherigen Absätze wurden die Variablen startTimeOfExecution und milliSecsOfRuntime erläutert. Um die Zeit zu messen, welche ein Algorithmus hat um zu arbeiten werden die Methoden clockInExecutionTime() und executionTimeLeft() verwendet. Beim Aufruf von clockInExecutionTime() wird die aktuelle Zeit in der Variable startTimeOfExecution vermerkt. Danach wird immer zu Beginn der nächsten Iteration oder einer zeitaufwendigen Berechnung die Methode executionTimeLeft() aufgerufen, welche überprüft, ob seit dem Aufruf von clockInExecutionTime() mehr als milliSecsOfRuntime Millisekunden vergangen sind und dementsprechend entscheidet ob abgebrochen oder weiter gerechnet wird.

In der Klasse AbstractIterativeOptimizationAlgorithm überlagert die Methode setSequence() der Klasse AbstractOptimizationAlgorithm und es werden weitere Verfahren, um die nächste zu evaluierende Lösung zu generieren, definiert. Der Parameter type gibt an, welches Verfahren zur Anwendung kommt. Wird er in dieser Methode nicht gefunden, wird der Aufruf an die setSequence() von AbstractOptimizationAlgorithm weitergereicht. Er ist vom Typ int, es sind jedoch nur int-Werte zulässig die in der OptimizationAlgorithmConstants Klasse mit Konstanten definiert sind, sonst wird ein Fehler ausgegeben.

Mögliche Werte sind jene, welche schon in der Beschreibung der Klasse AbstractOptimizationAlgorithm genannt wurden und zusätzliche wie zum Beispiel 5 für den besten Nachbarn in der Transpose-Nachbarschaft oder 8 für eine zufällig ausgewählten Nachbarn aus der Transpose-Nachbarschaft. Es werden hier exemplarisch die Varianten für den besten, einen zufälligen und den ersten Transpose-Nachbar vorgestellt.

Wurde setSequence() mit dem Parameter 5 (bester Nachbarn in der Transpose-Nachbarschaft) aufgerufen, reicht diese den Aufruf an getBestTransposeNeighbor() weiter und bekommt jene Auftragsfolge zurück, welche dem besten Nachbarn der aktuellen Auftragsfolge entspricht oder die aktuelle Auftragsfolge, falls kein besserer Nachbar für diese in der Transpose-Nachbarschaft existiert (Quellcode 4.12).

Quellcode 4.12: Funktion getBestTransposeNeighbor();

```
1  private List getBestTransposeNeighbor() {
2   List changedsequenceM1 = getCopyOfList(currentSequence);
3   AL5Solution bestCurTranSol = new AL5Solution();
4
5   for(int i=0;
6    i<changedsequenceM1.size()-1 && executionTimeLeft();
7    i++) {
8
9     Collections.swap(changedsequenceM1, i, i+1);
10
11    evaluate(changedsequenceM1);
12
13    int zfConst= OptimizationAlgorithmConstants.
          ZF_MEAN_COMPLETION_TIME;
14    double newOfVal=calculateObjectiveFunction(zfConst);
15    double bestOfVal=bestCurTranSol.getMeanCompletionTime();
16
17    if(newOfVal<bestOfVal) {
18      bestCurTranSol.setSeqM1(changedsequenceM1);
19      bestCurTranSol.setMeanCompletionTime(newOfVal);
20    }
21
22    changedsequenceM1 = getCopyOfList(currentSequence);
23  }
24
25  if(bestCurTranSol.getSeqM1() == null)
26    return getCopyOfList(currentSequence);
27  else
28    return bestCurTranSol.getSeqM1();
29 }
```

Zu Beginn wird eine Arbeitskopie der aktuellen Auftragsfolge erstellt (Zeile 2) und ein AL5Solution Objekt angelegt, um die aktuell beste gefundene Lösung zu speichern (Zeile 3). Die Transpose Nachbarschaft ist (n-1) Elemente groß (n ist die Anzahl der Aufträge), weshalb in der for-Schleife (Zeilen 5 - 7) so lange iteriert wird, so lange die Zählervariable i kleiner als changedsequenceM1.size()-1 ist. Ausserdem wird nur iteriert, so lange noch Zeit zum Ausführen übrig ist (siehe Beschreibung der Funktion executionTimeLeft()). Der Tausch der Nachbarn wird in Zeile 9 durchgeführt und danach wird in Zeile 10 die veränderte Auftragsfolge ausgewertet. Der Block in den Zeilen 13 bis 15 lässt den Zielfunktionswert der veränderten Folge auswerten und holt zum Vergleichen den Zielfunktionswert der aktuell besten Lösung. Ist in Zeile 17 der neue Zielfunktionswert kleiner als der alte, so wird die veränderte Lösung als die aktuell Beste gespeichert. In Zeile 22 wird die Arbeitskopie wieder auf die aktuelle Ausgangsfolge zurück gesetzt.

Es können nun die Fälle auftreten, dass schon beim ersten Überprüfen der Abbruchbedingung der for-Schleife keine Zeit zum Berechnen mehr übrig ist (executionTimeLeft() liefert false) oder, dass nie ein besserer Nachbar gefunden wird. In diesem Fall ist die Auftragsfolge in der aktuell besten Lösung null. Zeile 25 überprüft dies und liefert die aktuelle Ausgangsfolge an getSequence() zurück. Andernfalls wird der aktuell beste gefundene Nachbar zurück geliefert.

Wurde setSequence() mit dem Parameter 8 (zufälliger Transpose-Nachbar) aufgerufen, reicht diese den Aufruf weiter an getRNDTransposeNeighbor() und bekommt einen zufälligen Nachbarn zurück (Quellcode 4.13).

Quellcode 4.13: Funktion getRNDTransposeNeighbor();

```
1 private List getRNDTransposeNeighbor() {
2  List changedsequenceM1 = getCopyOfList(currentSequence);
3
4  int i = mtf.nextInt(changedsequenceM1.size()-1);
5
6  Collections.swap(changedsequenceM1, i, i+1);
7
8  return changedsequenceM1;
9 }
```

In dieser Funktion wird zu Beginn eine Arbeitskopie erstellt (Zeile 2), danach eine ganzzahlige Zufallszahl zwischen 0 und der Anzahl der Aufträge kleiner 1 gezogen (Zeile 4), das Element an i-ter Stelle und sein Nachfolger in der Folge getauscht (Zeile 6) und die veränderte Auftagsfolge zurück geliefert (Zeile 8).

Wurde setSequence() mit dem Parameter 11 (erster besserer Nachbarn in der Transpose-Nachbarschaft) aufgerufen, reicht diese den Aufruf weiter an getFirst-TransposeNeighbor() und bekommt einen zufälligen Nachbarn zurück (Quellcode 4.14).

Die Funktion getFirstTransposeNeighbor() hat bis auf 2 Änderungen den selben Aufbau wie getBestTransposeNeighbor(). Die erste Änderung ist der Block zwischen den Zeilen 4 und 9. Hier wird die aktuelle Lösung ausgewertet und als Vergleichslösung in der aktuell besten gefundenen Nachbarslösung vermerkt. Die zweite Änderung ist in Zeile 25. Da in der aktuell besten gefundenen Nachbarschaftslösung die Ausgangslösung ist, wurde der erste bessere Nachbar gefunden, sobald das erste Mal eine Lösung gefunden wurde bei welcher in Zeile 22 der neue Zielfunktionswert kleiner ist als der bisherige. Dann wird die neue Auftragsfolge und der neue Zielfunktionswert vermerkt, mit break; in Zeile 25 jedoch die

for-Schleife abgebrochen und keine weitere Lösung überprüft.

Quellcode 4.14: Funktion getFirstTransposeNeighbor();

```
1  private List getFirstTransposeNeighbor() {
2   List changedsequenceM1 = getCopyOfList(currentSequence);
3
4   evaluate(currentSequence);
5   AL5Solution bestCurTranSol = new AL5Solution();
6   int zfConst= OptimizationAlgorithmConstants.ZF_MEAN_COMPLETION_TIME
        ;
7   double ofVal= calculateObjectiveFunction(zfConst);
8   bestCurTransposeSolution.setSeqM1(currentSequence);
9   bestCurTransposeSolution.setMeanCompletionTime(ofVal);
10
11  for(int i=0;
12   i<changedsequenceM1.size()-1 && executionTimeLeft();
13   i++) {
14
15     Collections.swap(changedsequenceM1, i, i+1);
16
17     evaluate(changedsequenceM1);
18
19     double newOfVal=calculateObjectiveFunction(zfConst);
20     double bestOfVal=bestCurTranSol.getMeanCompletionTime();
21
22     if(newOfVal<bestOfVal) {
23       bestCurTranSol.setSeqM1(changedsequenceM1);
24       bestCurTranSol.setMeanCompletionTime(newOfVal);
25       break;
26     }
27
28     changedsequenceM1 = getCopyOfList(currentSequence);
29  }
30
31  if(bestCurTranSol.getSeqM1() == null)
32    return getCopyOfList(currentSequence);
33  else
34    return bestCurTranSol.getSeqM1();
35 }
```

4.2.2.3 Klasse AL5SPT

Die in Abschnitt 3.2.1 vorgestellten Prioritätsregeln berechnen anhand der Daten der Aufträge analytisch einmal eine beste Auftragsfolge. Sie sind daher nicht iterativ und werden von der Klasse AbstractOptimizationAlgorithm abgeleitet. Sie wurden in den Klassen AL5SPT, AL5SPTmod1, AL5SPTmod2 und

AL5LPT umgesetzt. Ihr Aufbau ist sehr ähnlich, weshalb hier exemplarisch nur AL5SPT beschrieben wird.

Quellcode 4.15: Klasse AL5SPT

```
1  public class AL5SPT extends AbstractOptimizationAlgorithm {
2   public void execute() {
3     setSequence(OptimizationAlgorithmConstants.SPT_SOLUTION);
4
5     evaluate();
6
7     int zfConst= OptimizationAlgorithmConstants.
          ZF_MEAN_COMPLETION_TIME;
8     double newOfVal=calculateObjectiveFunction(zfConst);
9
10    bestSolution.setSeqM1(changedsequenceM1);
11    bestSolution.setMeanCompletionTime(newOfVal);
12  }
13 }
```

Die Klasse AL5SPT (siehe Quellcode 4.15) erweitert AbstractOptimizationAlgorithm (Zeile 1) und implementiert die abstrakte Methode execute() ihrer Superklasse (Zeile 2). Zu Beginn wird in Zeile 3 die setSequence Methode der Superklasse mit dem Parameter für das SPT-Verfahren aufgerufen, um eine Lösung zu setzen, welche nach diesem Verfahren erzeugt wurde. Diese Zeile ist die einzige, welche sich bei den Klassen, welche Prioritätsregeln umsetzen, unterscheidet. Danach wird in Zeile 5 die gesetzte Lösung ausgewertet, in Zeile 8 die Zielfunktion ausgewertet und in den Zeilen 10 und 11 die Lösung in der Instanzvariable bestSolution (aus der Superklasse geerbt) vermerkt.

4.2.2.4 Klasse AL5VNS

Die in Abschnitt 3.3 vorgestellten Metaheuristiken Variable Neighborhood Search und Simulated Annealing sind iterative Alorithmen. Sie wurden in den Klassen AL5VNS und AL5SA umgesetzt. Exemplarisch soll hier die Umsetzung von AL5VNS vorgestellt werden.

Wie im letzten Absatz beschrieben ist VNS ein iterativer Algorithmus weswegen die Klasse AL5VNS (siehe Quellcode 4.16) von AbstractIterativeOptimizationAlgorithm abgeleitet wird. Sie besitzt das Instanzattribut bestImprovment und überlagert die abstrakte Methode execute() von AbstractOptimizationAlgorithm. Das Attribut bestImprovment gibt an, ob im Schritt “lokale Optimierung”

der beste Nachbar der jeweiligen Nachbarschaft gesucht wird, oder ob nach dem auffinden des ersten besseren Nachbarn abgebrochen werden soll.

In der Methode execute() ist das VNS-Verfahren umgesetzt. Zu Beginn wird in Zeile 5 mit clockInExecutionTime() für die spätere Laufzeitberechnung vermerkt, wann der Algorithmus gestartet wurde. Danach wird in den Zeilen 7 bis 12 die Ausgangslösung berechnet und vermerkt. Die Variable typeOfInitialSolution erbt die Klasse von AbstractIterativeOptimizationAlgorithm und sie kann die Werte 1 für eine zufällige Ausgangslösung und 3 für eine Ausgangslösung, die nach dem SPT-Verfahren berechntet wird, annehmen. In Zeile 14 wird die Anzahl der Nachbarschaften kMax für den Shaking-Schritt mit 3 Mal der Anzahl der Aufträge angesetzt. Zeile 15 gibt an, dass das komplette Verbesserungsverfahren so lange wiederholt werden soll wie noch Zeit übrig ist.

Das Verbesserungsverfahren startet in Zeile 16, wo festgelegt wird, dass mit der ersten Nachbarschaft begonnen werden soll. Danach wird in Zeile 17 angegeben, dass so lange die Shaking-Nachbarschaften durchsucht werden sollen, so lange die aktuelle Nachbarschft k nicht größer als Anzahl der Nachbarschaten kMax ist und noch Zeit übrig ist. In Zeile 18 wird als derzeit betrachtete Lösung die derzeit beste vermerkt.

Der Block von Zeile 19 bis Zeile 27 beruht auf folgender Idee für die Shakingphase. Die Nachbarschaften sollen abwechselnd aufgerufen werden und je höher k ist desto weiter soll vom Ausgangspunkt weggegangen werden. Ersteres wird erreicht indem jede 3. Iteration beginnend bei k=1 Transpose-Shakingschritte gemacht werden, jede 3. Iteration beginnend bei k=2 Interchange-Shakingschritte und jede 3. Iteration beginnend bei k=3 Insert-Shakingschritte. Zweiteres wird erreicht indem die Anzahl der Aufrufe, welchen einen neuen Nachbarn bestimmen, (k-1 div 3)+1 ist, also zum Beispiel 1 Transposeschritt bei k=1, 2 Transposeschritte bei k=4, 3 Transposeschritte bei k=7, usw. Angenommen das aktuelle k ist 7, dann wird Zeile 19 aktiv und es wird insgesamt drei Mal setSequence(OptiConstants.RANDOM_TRANSPOSE); aufgerufen. Es ist hier zu bemerken, dass die aktuelle Lösungsfolge in curSeq mitgespeichert wird und dass der 2. Aufruf von setSequence einen zufälligen Nachbarn von jener Lösung generiert, welche beim 1. Aufruf generiert wurde.

Nachdem im Shaking ein neuer Ausgangspunkt für die Suche erzeugt wurde,

wird nun in den Zeilen 29 bis 34 von diesem Punkt weg lokale Verbesserung mit VND, umgesetzt in der Klasse AL5VND, durchgeführt. Der Klasse werden zu diesem Zweck folgende Dinge übergeben: der aktuelle Ausgangspunkt (Zeile 30), eine gewisse Zeit welche sie zur Verbessrung hat (Zeile 31) und ob jeweils der beste oder der erste Nachbar genommen werden soll (Zeile 32). In Zeile 34 wird die Ergbnisfolge aus der lokalen Optimierung in curSeq eingelesen und ausgewertet.

Falls der gefundene Zielfunktionswert besser ist als der bisherige beste (Zeile 40), werden in der Variable für die beste gefundene Lösung die gefundene Auftragsfolge und der gefundene Zielfunktionswert vermerkt. Wie vom Verfahren vorgegeben, wird außerdem in Zeile 43 die aktuelle Nachbarschaft, gespeichert in der Variable k, wieder auf 1 gesetzt. Ist der gefundene Zielfunktionswert nicht besser als der beste bisher gefundene, dann wird in Zeile 46 in die nächst höhere Nachbarschaft gesprungen.

Quellcode 4.16: Klasse AL5VNS

```
1  public class AL5VNS  extends AbstractIterativeOptimizationAlgorithm
       {
2  public boolean bestImprovment;
3
4  public void execute() {
5   clockInExecutionTime();
6
7   setSequence(typeOfInitialSolution);
8   evaluate();
9   bestSolution.setSeqM1(curSeq);
10  int zfConst= OptimizationAlgorithmConstants.ZF_MEAN_COMPLETION_TIME
        ;
11  double ofVal= calculateObjectiveFunction(zfConst);
12  bestSolution.setMeanCompletionTime(ofVal);
13
14  int kMax = curSeq.size()*3;
15  while(executionTimeLeft()) {
16    int k = 1;
17    while(k<=kMax && executionTimeLeft()) {
18      curSeq = bestSolution.getSeqM1();
19      if(k%3==1)
20        for(int i=0; i<((k-1)/3)+1; i++)
21          setSequence(OptiConstants.RANDOM_TRANSPOSE);
22      else if(k%3==2)
23        for(int i=0; i<((k-1)/3)+1; i++)
24          setSequence(OptiConstants.RANDOM_INTERCHANGE);
25      else if(k%3==0)
26        for(int i=0; i<((k-1)/3)+1; i++)
27          setSequence(OptiConstants.RANDOM_INSERT);
28
29      AL5VND realVND1stSeq = new AL5VND();
30      realVND1stSeq.setInititalSequenceM1(curSeq.getSeqM1());
31      realVND1stSeq.setMilliSecsOfRuntime(getMilliSecsOfRuntime()
          /100);
32      realVND1stSeq.setBestImprove(bestImprovment);
33      realVND1stSeq.execute();
34      curSeq = realVND1stSeq.getBestSolution().getSeqM1());
35
36      evaluate();
37
38      double newOfVal=calculateObjectiveFunction(zfConst);
39      double bestOfVal=bestSolution.getMeanCompletionTime();
40      if(newOfVal<bestOfVal) {
41        bestSolution.setSeqM1(curSeq.getSeqM1());
42        bestSolution.setMeanCompletionTime(newOfVal);
43        k=1;
44      }
45      else
46        k++;
47    } } }
```

Kapitel 5

Experimente und Resultate

Für heuristische Verfahren ist es im Allgemeinen nicht möglich mathematisch exakte Aussagen über die Qualität der von ihnen berechneten Lösungen zu machen. Aus diesem Grund müssen empirische Experimente angewendet werden, um Einsicht in die Güte von vorgeschlagenen Verfahren zu bekommen.

Abschnitt 5.1 beschreibt die möglichen Parameter des gestellten Problems, die darauf angewendeten Algorithmen und die Durchführung der Versuche. Abschnitt 5.2 stellt die Erkenntnisse aus den Versuchen dar.

5.1 Versuchsanordnung

Rardin und Uzsoy (2001) definieren ein *Problem* als eine generische Form wie zum Beispiel flexibles Flow-Shop-Problem (FFc) und eine *Instanz* als eine bestimmte nummerische Ausprägung der Problemparameter (z.B. zwei Stufen und 50 Aufträge im FFc). Eine *Replikation* ist eine bestimmte Ausprägung der Instanz, z.B. spezifische Werte für die besten Schätzungen der Bearbeitungszeiten der 50 Aufträge. Da es sich bei dem in dieser Arbeit behandelten Modell um ein stochastisches handelt treten bei jedem Simulationslauf (in der Simulationsliteratur ebenfalls Replikation genannt) bestimmte und wahrscheinlich voneinander unterschiedliche tatsächliche Bearbeitungszeitenen auf. Um diesen Namenskonflikt zu entschärfen wird letzteres in dieser Arbeit *Stichprobenreplikation* genannt.

Ziel der Versuchsanordung ist es, so viele Replikationen wie möglich von un-

terschiedlichen Instanzen des Problems zu testen, um eine allgemeine Aussage über die Lösungsqualität der Algorithmen zu gewinnen.

Der folgende Abschnitte 5.1.1 stellt die Problemparameter des hier behandelten Problems vor. Abschnitt 5.1.2 führt die verwendeten Algorithmen an und Abschnitt 5.1.3 erklärt das Schema der Versuchsdurchführung.

5.1.1 Problemparameter

Die Problemparameter des hier behandelten Problems sind:

- Anzahl der Stufen
- Anzahl der Maschinen in Stufe s
- Größe des Puffers vor Stufe s
- Stochastik-Parameter
- Zielfunktion
- Anzahl der Aufträge
- Generierungsvorschrift der Auftragsdaten

Die Anzahl der Stufen (2), die Anzahl der Maschinen pro Stufe (Stufe 1: 1, Stufe 2: 2) und die Größe des Puffers vor Stufe 1 (unendlich) sind vorgegeben. Die anderen Charakteristika werden variiert. Werden diesen Problemparametern bestimmte Zahlenwerte zugewiesen, erhält man eine Probleminstanz. In den folgenden Absätzen werden die möglichen Werte beschrieben und am Ende des Abschnittes in einer Tabelle zusammengefasst.

Der Parameter **Größe des Puffers vor Stufe s** ist nach der Anzahl der Stufen und der Anzahl der Maschinen pro Stufe die dritte zentrale Kenngröße eines flexiblen Flow-Shop-Problems. Im letzten Absatz wurde bereits beschrieben, dass der Puffer vor Stufe 1 mit unendlicher Größe fixiert ist. Die Größe des Puffers vor der 2. Stufe soll variiert werden um die Auswirkungen dieses Parameters auf die Leistungsfähigkeit der Algorithmen zu testen. Drei Testwerte für die Anzahl der Aufträge, welche der Puffer beinhalten können soll, werden getestet: 20% der

Gesamtanzahl der Aufträge, 10% der Gesamtanzahl der Aufträge und genau 1 Auftrag.

Die **Stochastik-Parameter** beinhalten Kenngrößen, wie viel Unsicherheit dem Modell inne wohnt. Die Möglichkeiten stochastisches Verhalten im Modell zu berücksichtigen sind in dieser Arbeit Bearbeitungszeitvariation und Maschinenausfälle (siehe Abschnitt 2.4). Für den Parameter V der Bearbeitungszeitvariation werden die Werte V=0 (keine stochastischen Einflüsse) und V=0,2 (Bearbeitungszeiten können zwischen 80% und 120% der besten Schätzung liegen) getestet. Für Maschinenausfälle werden die beiden Möglichkeiten i) $\epsilon = 100\%$ und $d_{avg} = 0$, also kein Ausfall, und ii) $\epsilon = 90\%$ und $d_{avg} = p_{avg}$, wobei p_{avg} die gesamte durchschnittliche Bearbeitungszeit ist, überprüft.

Da die Kombination ohne Bearbeitungszeitvariation mit Maschinenausfällen als nicht sinnvoll erachtet wird, ergeben sich somit 3 Kombinationsmöglichkeiten:

1. V=0 mit $\epsilon = 100\%$ und $d_{avg} = 0$ (Bearbeitungszeit deterministisch, keine Maschinenausfälle)

2. V=0,2 mit $\epsilon = 100\%$ und $d_{avg} = 0$ (Bearbeitungszeiten stochastisch, keine Maschinenausfälle)

3. V=0,2 mit $\epsilon = 90\%$ und $d_{avg} = p_{avg}$ (Bearbeitungszeit stochastisch, Maschinenausfälle)

In Abschnitt 2.4 wurden die drei zu untersuchenden **Zielfunktion** vorgestellt. Es gilt zu ergründen, wie die Leistung der Algorithmen auf diese unterschiedlichen Bewertungmaße reagiert und ob sich "Spezialisten" für bestimmte Maße ausmachen lassen. 4 Zielfunktionen werden untersucht:

1. die Auslastung der Maschine 1,

2. der Pufferfüllstand des Puffers vor Stufe 2,

3. die durchschnittliche Fertigstellungszeit und

4. eine Kombination der zuvor genannten 3 Varianten.

Mathematisch lassen sich alle 4 Varianten wie in Formel 5.1 darstellen. Die vier Varianten werden durch festlegen der Variablen α, β und γ fixiert. Variante 1 entspricht $\alpha = 1$, Variante 2 entspricht $\beta = 1$, Variante 3 entspricht $\gamma = 1$ und Variante 4 entspricht α=0,5, β=0,25 und γ=0,25.

$$\begin{aligned} &\alpha \cdot \text{ (-Auslastung)} + \beta \cdot \text{ Pufferstand} + \\ &\gamma \cdot \text{ Durchschn. Fertigstellungszeiten} \rightarrow \text{MIN} \\ &\alpha + \beta + \gamma = 1 \end{aligned} \tag{5.1}$$

Mit dem Parameter **Anzahl der Aufträge** lässt sich die Größe der Probleminstanz steuern. Es soll untersucht werden, wie die unterschiedlichen Algorithmen auf kleine bis große Instanzen reagieren. Es werden Instanzen mit 10, 50 und 250 Aufträgen getestet.

Die **Generierungsvorschrift der Auftragsdaten** gibt an, nach welchen Muster die Daten der Aufträge, in diesem Fall die Bearbeitungszeiten, generiert werden. Es gilt hierbei die Ausführungen in Abschnitt 4.5. von Rardin und Uzsoy (2001) zu beachten. Dort wird diskutiert, auf welche Fallen bei der künstlichen Generierung von Replikationsdaten zu achten ist. Es sollten verschiedene Generierungsvorschriften untersucht werden um Aussagen über die Leistungsfähigkeit von Algorithmen in unterschiedlichen realen Umgebungen zu untersuchen.

In dieser Arbeit werden zwei Möglichkeiten untersucht. In der *ersten Variante* werden die besten Schätzungen der Bearbeiungszeit auf Maschine 1 aus der Gleichverteilung im Intervall [50;150] und auf den Maschinen 2 und 3 aus der Gleichverteilung im Intervall [100;300] gezogen. In der *zweiten Variante* werden für jede Maschine jeweils 2 Grundprozesszeiten (M1:40,60;M2:100,160;M3:120,200) festgelegt. Für jeden Auftrag wird für jede Maschine zufällig eine der Grundprozesszeiten gewählt. Die beste Schätzung der Bearbeitungszeit wird aus der Gleichverteilung im Intervall [Grundprozesszeit-10; Grundprozesszeit+10] gezogen.

Variante 1 erzeugt gleichmässig verteilte Daten, was zum schnellen Testen durchaus ausreichend ist. Rardin und Uzsoy (2001) führen jedoch an, dass dies zum Einen selten der Realität entspricht, und zum Anderen, dass dies beeinflussen kann wie schwierig die Instanz zu lösen ist. Aus diesem Grund wurde Variante 2

ersonnen, um realistischere und korrelierte Daten zu generieren. Die Idee ist, dass die Aufträge bestimmten Typen entsprechen, welche eine gewisse Grundprozesszeit haben. Innerhalb eines Auftragstyps kann es jedoch zu leichten Variationen auf Grund von unterschiedlichen Bearbeitungsanforderungen kommen. Man beachte, dass die Grundprozesszeiten der 3. Maschine im Durchschnitt höher liegen als jene der 2. Maschine. Diese Eigenschaft soll in das Modell einbringen, dass die 3. Maschine ein älteres Fabrikat ist und dadurch langsamer.

Alle beschriebenen Instanzparameter welche variiert werden sollen und alle ihre zu testenden Ausprägungen sind in Tabelle 5.1 zusammengefasst.

Grösse des Puffers vor Stufe 2	1) 20% der Anzahl der Aufträge, 2) 10% der Anzahl der Aufträge, 3) 1 Auftrag
Stochastik-Parameter	1) V=0 mit $\epsilon = 100\%$ und $d_{avg} = 0$ 2) V=0,2 mit $\epsilon = 100\%$ und $d_{avg} = 0$ 3) V=0,2 mit $\epsilon = 90\%$ und $d_{avg} = p_{avg}$
Zielfunktion	1) α=1,00; β=0,00; γ=0,00; 2) α=0,00; β=1,00; γ=0,00; 3) α=0,00; β=0,00; γ=1,00; 4) α=0,50; β=0,25; γ=0,25;
Anzahl der Aufträge	1) 10 2) 50 3) 250
Generierungsvorschrift der Auftragsdaten	1) M1: U[50;150]; M2, M3: U[100;300] 2) Grundprozesszeiten +- 10

Tabelle 5.1: Namen und Ebenen der Instanzparameter

5.1.2 Algorithmen

Ziel dieser Untersuchung ist es, die beiden in Abschnitt 3 vorgestellten Metaheuristiken VNS und SA zu vergleichen und bestmögliche Ergebnisse in den jeweiligen Zielfunktionen mit ihnen zu erzielen. Es ist jedoch bekannt, dass die Verwendung einer guten Ausgangslösung eine wichtige Grundlage für Metaheuristiken, die qualitativ hochwertige Lösungen liefern, ist.

Zwar ist es immer möglich eine zufällig generierte Lösung als Ausgangslösung zu verwenden, Voruntersuchungen haben jedoch gezeigt, dass sich durch die Prio-

ritätsregel SPT generierte Ausgangslösungen vorteilhaft auf die Ergebnisse der Metaheuristiken auswirken. Inspiriert wurde die Idee SPT als Ausgangslösung zu verwenden durch die Tatsache, dass für Maschinenbelegungsprobleme mit einer Maschine analytisch gezeigt werden kann, dass SPT zur minimalen gewichteten Fertigstellungszeit ($\sum w_j C_j$) führt.

Die klassische SPT Prioritätsregel verwendet jedoch nur die Bearbeitungszeiten einer Maschine, in dieser Arbeit der Maschine der ersten Stufe. Durch den Puffer begrenzter Größe zwischen den beiden Stufen liegt jedoch die Vermutung nahe, dass die Bearbeitungszeiten der zweiten Stufe ebenfalls eine Auswirkung auf die Lösungsqualität haben. Aus diesem Grund werden zusätzliche zur klassischen **SPT** Prioritätsregel zwei Abwandlungen umgesetzt. Der **SPTmod1** genannte Algorithmus verwendet die Bearbeitungszeiten der ersten Stufe plus den Durchschnitt der Bearbeitungszeiten der zweiten Stufe als Kennwert. Der **SPTmod2** genannte Algorithmus verwendet das Minimum der Bearbeitungszeiten der zweiten Stufe als Kennwert. Als Ausgangslösung für die Metaheuristiken wird die SPT Variante verwendet, welche die besten Ergebnisse liefert.

Für die Metaheuristik VNS werden zwei Varianten überprüft. In der ersten Variante (VNS1) verwendet die lokale Suche der VNS, hier das VND Verfahren, die beste gefundene Verbesserung einer Nachbarschaft als Ausgangspunkt für die nächste Iteration. In der zweiten Variante (VNS2) verwendet VND die erste gefundene Verbesserung als Ausgangspunkt. In beiden Varianten wird nach erreichen der maximalen Laufzeit abgebrochen und das beste bisher gefundene Ergebnis als Ausgangspunkt verwendet.

Die erste Variante ergründet sich aus der Vermutung, dass bessere Ergebnisse erzielt werden können, wenn immer der besten Verbesserungsmöglichkeit gefolgt wird. Der Vorteil der zweiten Variante ist es, dass in vielen Fällen nicht die gesamte Nachbarschaft überprüft werden muss, wodurch es dem Verfahren möglich ist mehr Lösungen zu untersuchen.

Die Metaheuristik SA verwendet die Insert-Nachbarschaft, um sich durch den Lösungsraum zu bewegen und dient als Vergleichsverfahren für die VNS Varianten. Alle betrachteten Optimierungsalgorithmen sind in Tabelle 5.2 angeführt.

Algorithmen	1) SPT 2) SPTmod1 (p1+(p2+p3)/2) 3) SPTmod2 (min(p2,p3)) 4) VNS1 (mit bester Verbesserung) 5) VNS2 (mit erster Verbesserung) 6) SA (mit zufälligem Insert-Nachbarn)

Tabelle 5.2: Algorithmen

5.1.3 Versuchsdurchführung

Die Versuche wurden mit AnyLogic 5.5, Java JDK 1.6.0_04, Eclipse 3.4.0 und dem freien open-source Statistikpaket R auf der Softwareseite und einem Intel Core 2 Duo 2,4 GHZ, 4GB RAM auf der Hardwareseite durchgeführt.

Um einen Einblick über die Qualität der Algorithmen bei unterschiedlichen Ausprägungen der Instanzdaten zu bekommen, wurden für jede Parameterkombination aus Tabelle 5.1 25 Replikationen generiert. Jedem Algorithmus wurden pro Replikation 60 Sekunden Laufzeit zum Lösen zur Verfügung gestellt. Jede Auftragsfolge wird 15 Mal ausgewertet (Stichprobenreplikation) und das Verfahren aus Abschnitt 3.4 angewendet, um zu bestimmen ob eine Auftragsfolge eine Verbesserung darstellt. Um gleiche Bedingungen zu gewährleisten, werden auch bei den deterministischen Instanzen die Auftragsfolgen 15 Mal ausgewertet.

Der erste Teil der Untersuchung versucht abzuklären, welche der drei SPT Varianten (SPT, SPTmod1 und SPTmod2) die besten Ergebnisse liefert. Die beste Variante dient im darauf folgenden Vergleich der Metaheuristiken als deren Ausgangslösung. Im zweiten Teil der Untersuchung werden die beiden VNS Varianten und SA verglichen. Letztendlich werden im dritten Teil der Untersuchung die beste SPT Variante und die beste Metaheuristik verglichen, um zu ermitteln in welchem Ausmaß sich durch das aufwendigere metaheuristische Verfahren Verbesserungen erzielen lassen.

5.2 Ergebnisse und Interpretation

In Rardin und Uzsoy (2001) wird der Unterschied zwischen statistischer und praktischer Signifikanz diskutiert. Damit ist gemeint, dass ein Unterschied von

einem Promille zwar statistisch signifikant sein kann, dies jedoch keinen praktischen Unterschied macht. Ein Verfahren ist somit nicht praktisch besser, nur weil es statistisch besser ist. Es ist jedoch sinnvoll einen Unterschied, bei dem man entschieden hat ihn für praktisch relevant zu halten, darauf hin zu überprüfen, ob er auch einen statistisch signifikanten Unterschied darstellt.

Wurden Ergebnisse erzielt, welche als praktisch signifikant identifiziert worden sind, muss ein geeignetes Verfahren gewählt werden, um die statistische Signifikanz zu überprüfen. In Rardin und Uzsoy (2001) wird zu diesem Zweck ANOVA vorgestellt. Diese Methode setzt jedoch voraus, dass die untersuchte Stichprobe normalverteilt ist. Nach bisherigen Voruntersuchungen kann nicht davon ausgegangen werden, dass die berechneten Zielfunktionswerte dieser Annahme gerecht werden, wodurch eine andere Methode gesucht werden muss.

In diesem Fall können nicht parametrische Tests wie der Wilcoxon-Test für **nicht verbundene** Stichproben (= Wilcoxon Rangsummen-Test = U-Test = Mann-Whitney-Test) und der Wilcoxon-Test für **verbundene** Stichproben (Wilcoxon Vorzeichen-Rang-Test) verwendet werden. Da die Versuchsanordung immer alle Algorithmen auf die selbe Replikation anwendet, sind die Ergebnisse für eine Replikation eine verbundene Stichprobe. Daraus folgt, dass der Wilcoxon Vorzeichen-Rang-Test angewendet werden muss.

5.2.1 Vergleich der Prioritätsregeln

25 Replikationen jeder Instanz (Parameterkombination aus Tabelle 5.1) wurden von den drei SPT Varianten SPT, SPTmod1 und SPTmod2 gelöst. Für jedes Ergebnis pro Algorithmus und Replikation wurde die prozentuelle Abweichung vom besten gefundenen Ergebnis aller Algorithmen in der Replikation berechnet.

Tabelle 5.3 zeigt für die drei SPT-Varianten die durchschnittliche Abweichung vom besten Replikationsergebnis aggregiert über alle Parameterkombinationen und Replikationen. Der beste Algorithmus ist fett hervorgehoben.

	SPT	SPTmod1	SPTmod2
Gesamt	7,21%	**1,70%**	3,60%

Tabelle 5.3: Durchschn. prozentuelle Abweichung SPT-Varianten Gesamt

Um den Einfluss der verschiedenen Problemparameter auf die Lösungsqualität zu untersuchen, zeigt Tabelle 5.4 die durchschnittliche Abweichung vom besten Replikationsergebnis aggregiert über die einzelnen Stufen der Problemparameter. Der beste Algorithmus pro Aggregationsstufe ist fett hervorgehoben.

	SPT	SPTmod1	SPTmod2
Puffergrösse			
20%	5,03%	**1,59%**	3,54%
10%	8,23%	**1,69%**	3,68%
1 Auftrag	8,38%	**1,84%**	3,59%
Anzahl der Aufträge			
10	4,82%	**2,98%**	5,03%
50	7,38%	**1,09%**	3,14%
250	9,44%	**1,03%**	2,64%
Stochastik-Parameter			
V=0;$\epsilon = 1$	7,78%	**1,73%**	4,00%
V=0,2;$\epsilon = 1$	7,30%	**1,50%**	3,53%
V=0,2;ϵ=0,9	6,56%	**1,88%**	3,28%
Zielfunktion			
α=1,00; β=0,00; γ=0,00;	3,41%	**0,88%**	1,00%
α=0,00; β=1,00; γ=0,00;	13,30%	2,83%	**1,65%**
α=0,00; β=0,00; γ=1,00;	6,08%	**1,55%**	5,90%
α=0,50; β=0,25; γ=0,25;	6,06%	**1,56%**	5,86%
Generierungsvorschrift			
U[50;150]/U[100;300]	7,05%	**1,69%**	5,29%
Grundprozesszeiten +- 10	7,37%	**1,72%**	1,92%

Tabelle 5.4: Durchschn. prozentuelle Abweichung SPT-Varianten Parameterstufen

Über alle Replikationen aggregiert ist SPTmod1 um ca. 2% besser als SPTmod2 und ca. 5,5% besser als SPT (Tabelle 5.3). Auch gruppiert über die Stufen der Problemparameter (Tabelle 5.4) ist bis auf eine Ausnahme SPTmod1 stets durchschnittlich am wenigsten weit vom besten gefunden Replikationsergebnis entfernt. Die erzielte Lösungqualität von SPTmod1 wird als praktisch relevant besser als jene der anderen beiden SPT-Varianten angesehen und auch der Vergleich aller Replikationsdaten mittels Wilcoxon Vorzeichen-Rang-Test von SPTmod1-SPT und SPTmod1-SPTmod2 hat ergeben, dass die Unterschiede statistisch signifikant sind. Aus diesem Grund wird beim Vergleich der Metaheuristiken SPTmod1 als Ausgangslösung verwendet.

5.2.2 Vergleich der Metaheuristiken

25 Replikationen jeder Instanz (Parameterkombination aus Tabelle 5.1) wurden von den Metaheuristiken VNS mit bester Verbesserung (VNS1), VNS mit erster Verbesserung (VNS2) und SA mit zufälligem Insert-Nachbarn gelöst. Als Ausgangslösung wurde, als Konsequenz der Erkenntnisse des letzten Abschnittes, die von SPTmod1 für die jeweilige Instanz generierte Lösung von den Metaheuristiken verwendet. Für jedes Ergebnis pro Algorithmus und Replikation wurde die prozentuelle Abweichung vom besten gefundenen Ergebnis aller Algorithmen in der Replikation berechnet.

Tabelle 5.5 zeigt für die drei Metaheuristiken die durchschnittliche Abweichung vom besten Replikationsergebnis aggregiert über alle Parameterkombinationen und Replikationen. Der beste Algorithmus ist fett hervorgehoben.

	VNS1	VNS2	SA
Gesamt	0,71%	**0,64%**	5,09%

Tabelle 5.5: Durchschn. prozentuelle Abweichung Metaheuristiken Gesamt

Um den Einfluss der verschiedenen Problemparameter auf die Lösungsqualität zu untersuchen zeigt Tabelle 5.6 die durchschnittliche Abweichung vom besten Replikationsergebnis aggregiert über die einzelnen Stufen der Problemparameter. Der beste Algorithmus pro Aggregationsstufe ist fett hervorgehoben.

Das Ergebnis, dass der Unterschied zwischen den beiden VNS Varianten im Gesamtdurchschnitt (Tabelle 5.5) weniger als 0,1% beträgt, gibt Grund zur Annahme, dass keines der beiden Verfahren besser ist. Diese Annahme wird auch durch die Beobachtung unterstüzt, dass die Erebnisse der Aggregation über die Stufen der Problemparameter (Tabelle 5.6) sehr eng beieinander liegen und in 20% der Problemparameterstufen die eine und in 80% der Problemparameterstufen die andere VNS Variante die besten Ergebnisse liefert. Bestätigt wird die Annahme letztendlich durch das Ergebnis des Wilcoxon Vorzeichen-Rang-Test. Dieser meldet für den Vergleich aller Replikationsdaten der beiden Varianten, dass kein signifikanter Unterschied in den Verteilungen der beiden Datensätze angenommen werden kann und somit, dass die beobachteten Unterschiede sehr wahrscheinlich zufällig sind.

Anders als beim Unterschied unter den VNS Varianten ist der Unterschied

	VNS1	VNS2	SA
Puffergrösse			
20%	0,74%	**0,67%**	5,05%
10%	0,75%	**0,66%**	5,20%
1 Auftrag	0,64%	**0,60%**	5,02%
Anzahl der Aufträge			
10	**0,90%**	0,94%	7,18%
50	0,86%	**0,66%**	6,56%
250	0,38%	**0,33%**	1,53%
Stochastik-Parameter			
V=0;$\epsilon = 1$	0,44%	**0,26%**	7,65%
V=0,2;$\epsilon = 1$	0,50%	**0,46%**	4,63%
V=0,2;ϵ=0,9	**1,19%**	1,20%	2,99%
Zielfunktion			
α=1,00; β=0,00; γ=0,00;	**0,34%**	0,36%	1,04%
α=0,00; β=1,00; γ=0,00;	0,93%	**0,77%**	5,68%
α=0,00; β=0,00; γ=1,00;	0,83%	**0,71%**	6,29%
α=0,50; β=0,25; γ=0,25;	0,75%	**0,74%**	7,36%
Generierungsvorschrift			
U[50;150]/U[100;300]	0,62%	**0,62%**	4,22%
Grundprozesszeiten +- 10	0,80%	**0,67%**	5,97%

Tabelle 5.6: Durchschn. prozentuelle Abweichung Metaheuristiken Parameterstufen

zwischen VNS und SA relativ groß. Beide VNS Varianten sind im Gesamtdurchschnitt (Tabelle 5.5) ca. 4,5% besser als SA und sowohl für den Vergleich VNS mit bester Lösung gegen SA als auch für den Vergleich VNS mit erster Lösung gegen SA ergibt der Wilcoxon Vorzeichen-Rang-Test, dass SA signifikant schlechter ist.

5.2.3 Verbesserungspotential der Metaheuristiken

Abschließend wurden 25 Replikationen jeder Instanz (Parameterkombination aus Tabelle 5.1) von allen 6 in den vorherigen beiden Abschnitten behandelten und in Tabelle 5.2 angeführten Algorithmen gelöst, um zu ergründen, wie viel Verbesserung der Lösungsqualität durch den Mehraufwand von Metaheuristiken gegenüber Prioritätsregeln erzielbar ist. Als Ausgangslösung wurde wieder die von SPTmod1 für die jeweilige Instanz generierte Lösung von den Metaheuristiken verwendet. Für jedes Ergebnis pro Algorithmus und Replikation wurde die

prozentuelle Abweichung vom besten gefundenen Ergebnis aller Algorithmen in der Replikation berechnet.

Tabelle 5.7 zeigt für alle 6 Algorithmen die durchschnittliche Abweichung vom besten Replikationsergebnis aggregiert über alle Parameterkombinationen und Replikationen. Der beste Algorithmus ist fett hervorgehoben.

	SPT	SPTmod1	SPTmod2	VNS1	VNS2	SA
Gesamt	11,79%	6,08%	8,06%	1,07%	**1,00%**	5,20%

Tabelle 5.7: Durchschn. prozentuelle Abweichung Alle Gesamt

Um den Einfluss der verschiedenen Problemparameter auf die Lösungsqualität zu untersuchen zeigt Tabelle 5.8 die durchschnittliche Abweichung vom besten Replikationsergebnis aggregiert über die einzelnen Stufen der Problemparameter. Der beste Algorithmus pro Aggregationsstufe ist fett hervorgehoben.

Beide VNS Varianten sind im Durchschnitt über alle Replikationen nur 1% vom jeweiligen besten gefundenen Repliaktionsergebnis entfernt. Die beiden Verfahren finden also nicht immer die beste Lösung, sind jedoch im Durchschnitt sehr nahe dran. Der Wert der besten Prioritätsregel SPTmod1 ist um 5% schlechter was zeigt, dass durchaus erhebliches Verbesserungspotential gegenüber der Lösungen der Prioritätsregeln gegeben ist.

	SPT	SPTmod1	SPTmod2	VNS1	VNS2	SA
Puffergrösse						
20%	12,31%	5,56%	7,39%	1,08%	**1,03%**	5,48%
10%	13,08%	6,28%	8,35%	1,09%	**1,00%**	5,56%
1 Auftrag	9,99%	6,41%	8,44%	1,04%	**0,97%**	5,38%
Anzahl der Aufträge						
10	12,97%	10,96%	13,11%	**1,00%**	1,03%	7,28%
50	11,85%	5,26%	7,41%	1,39%	**1,19%**	5,22%
250	10,56%	2,03%	3,67%	0,83%	**0,78%**	1,99%
Stochastik-Parameter						
V=0;$\epsilon = 1$	14,29%	7,91%	10,28%	0,87%	**0,68%**	7,85%
V=0,2;$\epsilon = 1$	10,94%	5,00%	7,08%	0,79%	**0,76%**	4,93%
V=0,2;ϵ=0,9	10,14%	5,35%	6,82%	**1,55%**	1,56%	3,36%
Zielfunktion						
α=1,00; β=0,00; γ=0,00;	5,44%	2,87%	3,00%	**0,55%**	0,57%	1,25%
α=0,00; β=1,00; γ=0,00;	19,89%	8,89%	7,68%	1,55%	**1,38%**	6,33%
α=0,00; β=0,00; γ=1,00;	10,90%	6,25%	10,77%	1,10%	**0,98%**	6,15%
α=0,50; β=0,25; γ=0,25;	10,94%	6,33%	10,79%	1,08%	**1,07%**	6,25%
Generierungsvorschrift						
U[50;150]/U[100;300]	11,20%	5,63%	9,32%	**0,79%**	**0,79%**	4,40%
Grundprozesszeiten +- 10	12,39%	6,54%	6,80%	1,35%	**1,21%**	6,53%

Tabelle 5.8: Durchschn. prozentuelle Abweichung Alle Parameterstufen

Kapitel 6

Zusammenfassung

Dieses Kapitel fasst die Erkenntnisse dieser Arbeit zusammen und versucht einen Ausblick auf mögliche zukünftige Weiterentwicklungen zu geben.

Am Beginn der Arbeit wird in Abschnitt 2 die Reihenfolgeplanung im Kontext der Planung im Unternehmen vorgestellt und es werden Standardprobleme dieses Gebietes aus der Literatur angeführt. Dies ermöglicht es am Ende des Abschnittes das in dieser Arbeit konkret untersuchte stochastische flexible Flow-Shop-Problem zu beschreiben.

Abschnitt 3 erläutert die verschiedenen Möglichkeiten, Probleme der Reihenfolgeplanung zu lösen. Da es sich bei dem hier behandelten Problem um ein schweres kombinatorisches Optimierungsproblem handelt, muss zur Lösung auf heuristische Verfahren zurückgegriffen werden. Es werden einerseits Prioritätsregeln vorgestellt, welche die Aufträge anhand ihrer Basisdaten reihen und andererseits Metaheuristiken, welche generische Verfahren zum Lösen von kombinatorischen Optimierungsproblemen sind. Die hier verwendeten Metaheuristiken VNS und SA bauen beide auf der Idee der lokalen Suche auf. Aus diesem Grund werden in Abschnitt 3.2.2.2 Nachbarschaften für Reihenfolgeplanungsprobleme vorgestellt.

Konkret werden drei Variationen der Prioritätsregel SPT, zwei Variationen der Metaheuristik VNS und die Metaheuristik SA, umgesetzt (Abschnitt 4.2) und in Abschnitt 5 deren Lösungsqualität untersucht.

Im ersten Versuch (Abschnitt 5.2.1) werden die drei Variationen der Prioritätsregel SPT miteinander verglichen. Dies führt zur Erkenntnis, dass die erste

Modifikation von SPT, SPTmod1 genannt, unter den untersuchten Prioritätsregeln die besten Lösungen liefert. Allgemein wurde die Erkenntnis gewonnen, dass mit begrenztem Puffer zwischen den Stufen und Rückstau aus der zweiten Stufe Prioritätsregeln, die die Verarbeitungszeiten der zweite Stufe miteinbeziehen, qualitativ höherwertige Lösungen erzielen.

Im zweiten Versuch (Abschnitt 5.2.2) werden die beiden Varianten der Metaheuristik VNS mit der Metaheuristik SA verglichen. Keine signifikanten Unterschiede konnten zwischen den umgesetzten VNS Varianten festgestellt werden, beide waren jedoch erheblich und signifikant besser als SA.

Der letzte vorgestellte Versuch (Abschnitt 5.2.3) zeigt, dass die durch die Prioritätsregel SPTmod1 erzielten Lösungen durch VNS weiter verbessert werden können. Es muss jedoch von Anwendungsfall zu Anwendungsfall entschieden werden, ob die erzielte Verbesserung in akzeptabler Relation zu dem durch die Metaheuristik auftretenden Mehraufwand steht.

Diese Arbeit kann als Basis für zukünftige Entwicklungen dienen, da das in dieser Arbeit vorgestellte Modell und die damit durchgeführten Experimente viele interessante Anknüpfpunkte für Weiterentwicklungen bieten. Folgende Liste gibt einen Überblick über einige der denkbaren Erweiterungsansätze:

- Anstatt des in Abschnitt 2.4 vorgestellten speziellen flexiblen Flow-Shop-Modell könnte die Untersuchung auf ein allgemeines, flexibles Flow-Shop-Modell ausgedehnt werden, in dem auch die Anzahl der Stufen und die Anzahl der Maschinen pro Stufe als Problemparameter verwendet werden.

- Die Aufnahme weiterer Zielfunktionen in die Versuchsanordnung würde es ermöglichen, Aussagen über die Qualität der Lösungsalgorithmen in Bezug auf diese machen zu können.

- In der Literatur werden viele weitere Prioritätsregeln und Metaheuristiken beschrieben. Die Weiterführung der Experimente dieser Arbeit mit diesen Algorithmen würde eine bessere Einschätzung der Güte der Lösungsqualität der hier verwendeten Algorithmen ermöglichen.

- In dieser Arbeit wurden die unterschiedlichen Einstellungsmöglichkeiten innerhalb der Algorithmen nur sehr begrenzt untersucht. Eine detailierte Un-

tersuchung dieses Gebietes könnte weitere Verbesserungen in der Lösungsqualität und Erkenntnisse über die Zusammenhänge zwischen Problem- und Algorithmenparameter mit sich bringen.

Literaturverzeichnis

[Allaoui und Artiba 2006] Allaoui, A. ; Artiba, A.: Scheduling two-stage hybrid flow shop with availability constraints. In: *Comput. Oper. Res.* 33 (2006), Nr. 5, S. 1399–1419

[den Besten und Stützle 2001] Besten, M. den ; Stützle, Thomas: Neighborhoods revisited: An Experimental Investigation into the Effectiveness of Variable Neighborhood Descent for Scheduling. In: *In Proceedings of MIC 2001*, 2001, S. 545–550

[Bleymüller u. a. 2000] Bleymüller, J. ; Gehlert, G. ; Gülicher, H.: *Statistik für Wirtschaftswissenschaftler*. 12. Aufl. München : Verlag Vahlen, 2000

[Domschke u. a. 1993] Domschke, W. ; Scholl, A. ; Voss, S.: *Produktionsplanung : ablauforganisatorische Aspekte*. Berlin Heidelberg New York : Springer, 1993

[Dorigo und Stützle 2004] Dorigo, M. ; Stützle, T.: *Ant Colony Optimization*. The MIT Press, 2004

[Dueck und Scheuer 1990] Dueck, G. ; Scheuer, T.: Threshold accepting: a general purpose optimization algorithm appearing superior to simulated annealing. In: *J. Comput. Phys.* 90 (1990), Nr. 1, S. 161–175

[Eglese 1990] Eglese, R. W.: Simulated annealing: A tool for operational research. In: *European Journal of Operational Research* 46 (1990), Nr. 3, S. 271–281

[Glover 1986] Glover, F.: Future paths for integer programming and links to artificial intelligence. In: *Comput. Oper. Res.* 13 (1986), Nr. 5, S. 533–549

[Glover und Kochenberger 2003] Glover, F. ; Kochenberger, G. A.: *Handbook of Metaheuristics (International Series in Operations Research & Management Science)*. New York, Boston, Dordrecht, London, Moscow : Kluwer ACcademic Publishers, 2003

[Hansen und Mladenovic 2003] Hansen, P. ; Mladenovic, N.: A tutorial on variable neighborhood search / Les Cahiers du GERAD, HEC Montreal and GERAD. 2003. – Forschungsbericht

[Hartl und Dörner 2006] Hartl, R.F. ; Dörner, K.: *Skriptum: Operations Management*. Wien : Produktion und Logisitk VIS, 2006

[Hillier und Lieberman 2002] Hillier, F. ; Lieberman, G.: *Introduction to Operations Research*. 7. Aufl. New York : McGraw-Hill, 2002

[Jensen 2003] Jensen, M. T.: Generating robust and flexible job shop schedules using genetic algorithms. In: *Evolutionary Computation, IEEE Transactions on* 7 (2003), Nr. 3, S. 275–288

[Jungwattanakit u. a. 2009] Jungwattanakit, J. ; Reodecha, M. ; Chaovalitwongse, P. ; Werner, F.: A comparison of scheduling algorithms for flexible flow shop problems with unrelated parallel machines, setup times, and dual criteria. In: *Comput. Oper. Res.* 36 (2009), Nr. 2, S. 358–378. – ISSN 0305-0548

[Kirkpatrick u. a. 1983] Kirkpatrick, S. ; Gelatt, C. D. ; Vecchi, M. P.: Optimization by simulated annealing. In: *Science* 220 (1983), S. 671–680

[Kutanoglu und Sabuncuoglu 2001] Kutanoglu, E. ; Sabuncuoglu, I.: Experimental investigation of iterative simulation-based scheduling in a dynamic and stochastic job shop. In: *Journal of Manufacturing Systems* 20 (2001), Nr. 4, S. 264–279

[Law 1990] Law, A.: Models of random machine downtimes for simulation. In: *WSC' 90: Proceedings of the 22nd conference on Winter simulation*, IEEE Press, 1990, S. 314–316

[Law und Kelton 1999] Law, A. ; Kelton, W.: *Simulation Modeling and Analysis*. New York : McGraw-Hill, 1999

[Makoto und Takuji 1998] Makoto, M. ; Takuji, N.: Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. In: *ACM Trans. Model. Comput. Simul.* 8 (1998), Nr. 1, S. 3–30

[Mladenović und Hansen 1997] Mladenović, N. ; Hansen, P.: Variable neighborhood search. In: *Comput. Oper. Res.* 24 (1997), Nr. 11, S. 1097–1100

[Nahmias 2000] Nahmias, S.: *Production and Operations Analysis*. 4. Edition. McGraw-Hill/Irwin, 2000

[Panwalkar und Iskander 1977] Panwalkar, S. S. ; Iskander, W.: A Survey of Scheduling Rules. In: *Operations Research* 25 (1977), Nr. 1, S. 45–61

[Pinedo 2007] Pinedo, M.: *Planning and Scheduling in Manufacturing and Services*. 2007

[Pinedo 2008] Pinedo, M.: *Scheduling: Theory, Algorithms, and Systems*. 3. Edition. Wien-New York : Springer Verlag, 2008

[Rardin und Uzsoy 2001] Rardin, R. ; Uzsoy, R.: Experimental Evaluation of Heuristic Optimization Algorithms: A Tutorial. In: *J. Heuristics* 7 (2001), Nr. 3, S. 261–304

[Ross 2006] Ross, M. S.: *Simulation*. 4th ed. Academic Press, 2006

[Santos u. a. 1996] Santos, D.L. ; Hunsucker, J.L. ; Deal, D.E.: An evaluation of sequencing heuristics in flow shops with multiple processors. In: *Computers and Industrial Engineering* 30 (1996), S. 681–691

[Sawik 1993] Sawik, T. J.: A Scheduling Algorithm for Flexible Flow Lines with Limited Intermediate Buffers. In: *Applied Stochastic Models and Data Analysis* 9 (1993), Nr. 3, S. 127–138

[Schuh 2006] Schuh, G.: *Produktionsplanung und -steuerung: Grundlagen, Gestaltung und Konzepte*. 3rd ed. Berlin Heidelberg New York : Springer, 2006

[Tavakkoli-Moghaddam und Safaei 2007] Tavakkoli-Moghaddam, R. ; Safaei, N.: A New Mathematical Model for Flexible Flow Lines with Blocking Processor and Sequence-Dependent Setup Time. In: Levner, Eugene (Hrsg.):

Multiprocessor Scheduling, Theory and Applications, I-Tech Education and Publishing, Vienna, Austria, 2007, S. 266–283. – URL `http://intechweb.org/book.php?id=21`. – Online abgerufen am 15.11.2008

[Tavakkoli-Moghaddam u. a. 2009] Tavakkoli-Moghaddam, R. ; Safaei, N. ; Sassani, F.: A memetic algorithm for the flexible flow line scheduling problem with processor blocking. In: *Comput. Oper. Res.* 36 (2009), Nr. 2

[Uetz 2002] Uetz, M.: *Algorithms for Deterministic and Stochastic Scheduling*. Göttingen : Cuvillier Verlag, 2002

[Vaessens u. a. 1996] Vaessens, R. J. M. ; Aarts, E. H. L. ; Lenstra, J.K.: Job Shop Scheduling by Local Search. In: *INFORMS Journal on Computing* 8 (1996), Nr. 3, S. 302–317

[Vieira u. a. 2003] Vieira, G. E. ; Herrmann, J. W. ; Lin, E.: Rescheduling Manufacturing Systems: A Framework of Strategies, Policies, and Methods. In: *Journal of Scheduling* 6 (2003), Nr. 1, S. 39–62

[Weng 2000] Weng, M. X.: Scheduling flow-shops with limited buffer spaces. In: *Winter Simulation Conference Proceedings* Bd. Volume 2, 2000, S. 1359–1363

[Xjtek 2005a] Xjtek, XJ Technologies Company Ltd. ..: *AnyLogic Enterprise Library Reference Guide*. 2005. – URL `http://www.xjtek.com/file/94`. – [Online; retrieved on 07.12.2007]

[Xjtek 2005b] Xjtek, XJ Technologies Company Ltd. ..: *AnyLogic User's Manual*. 2005. – URL `http://www.xjtek.com/file/102`. – [Online; retrieved on 07.12.2007]

Anhang

Kurzzusammenfassung

Die vorliegende Diplomarbeit untersucht das stochastische fexible Flow-Shop-Problem mit begrenztem Puffer zwischen zwei vorhandenen Stufen. Maschinenausfälle und Variation der Bearbeitungszeit werden als stochastische Einflüsse berücksichtigt. Als Zielfunktionen für die Untersuchungen dienen Maschinenauslastung, Pufferstand und durchschnittliche Fertigstellungszeit. Das Modell wird für die Auswertung mit Hilfe der Simulationssoftware AnyLogic implementiert. Anschließend werden möglichst gute Auftragsfolgen der Maschinen für konkrete Probleminstanzen mit Hilfe von Prioritätsregeln und den Metaheuristiken Variable Neighborhood Search (VNS) und Simulated Annealing (SA) bestimmt. Die Arbeit endet mit Vergleichen zwischen den angewendeten Optimierungsverfahren und Bewertungen derselben. Die Resultate zeigen, dass die Lösungen der Prioritätsregeln mit Metaheuristiken erheblich verbessert werden können.

Abstract

The present thesis examines the stochastic flexible flow shop problem with a limited buffer between two defined stages. Stochastic influences occurring through machine breakdowns and variation of processing time are considered. Objective functions for the evaluations are machine utilization, buffer level and total completion time. A simulation for this model utilizing AnyLogic is described. Subsequently job sequences were calculated by dispatching rules and improved by Variable Neighborhood Search (VNS) und Simulated Annealing (SA). The thesis concludes with comparisons and ratings of the methods used. The results indicate that it is possible to improve the solutions generated by dispatching rules significantly by the use of metaheuristics.

Lebenslauf

ANGABEN ZUR PERSON

Name	Stefan Katzensteiner, Bakk. rer. soc. oec.
Adresse	Rieplstraße 9/1 A-1100 Wien
E-Mail	stefan.katzensteiner@fsmat.at
Staatsangehörigkeit	Österreich
Geburtsdatum	21.01.1981

AUSBILDUNG

10/2005	Abschluss als Bakkalaureus der Sozial- und Wirtschaftswissenschaften (Bakk.rer.soc.oec.)
10/2000 - 10/2005	Bakkalaureatsstudium Wirtschaftsinformatik an der Universität Wien und der Technischen Universität Wien
09/1991 - 06/1999	Realgymnasium in Mödling, Österreich

BERUFSERFAHRUNG

09/2008 - jetzt	Wissenschaftlicher Mitarbeiter im Projekt: VPR - Vorausschauende Produktionsregelung in Walzwerken Tätigkeit: Erforschung und Entwicklung von Optimierungsalgorithmen für Flexible Flow Shop Scheduling in Java Institut für Betriebswirtschaftslehre, Universität Wien, Österreich
09/2008 - jetzt	Technischer Mitarbeiter im Projekt: MCCE - Multi-Channel Consumer Experience Tätigkeit: Erforschung und Entwicklung von Agenten-basierten Simulationsmodellen Institut für Betriebswirtschaftslehre, Universität Wien, Österreich
07/2008 - jetzt	Selbstständiger Gewerbetreibender auf den Gebieten Informationstechnologie & -dienste
04/2006 - 03/2007	Wissenschaftlicher Mitarbeiter in Ausbildung im Projekt: CDPPA - Competence-Driven Project Portfolio Selection Tätigkeit: Erforschung und Entwicklung von Optimierungsalgorithmen für F&E Projektportfolioauswahl in C++ Institut für Statistik und Decision Support Systems, Universität Wien, Österreich
01/2000 - 08/2000	Wehrdienst mit Ausbildung zum Sanitätsgehilfen