JOHANNES KEPLER
UNIVERSITÄT LINZ

Eingereicht von
Manuel Schlenkrich

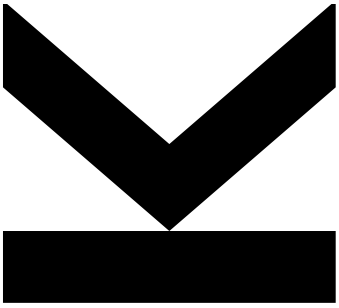
Angefertigt am
Research Institute for Symbolic Computation (RISC)

Betreuer und Erstbeurteiler
Assoc. Univ.-Prof. DI Dr. Wolfgang Windsteiger

Zweitbetreuer
DI Dr. Michael Bögl

September 2018

Das Shifting-Bottleneck Verfahren für das Job-Shop Scheduling-Problem



Bachelorarbeit

zur Erlangung des akademischen Grades

Bachelor of Science

im Bachelorstudium

Technische Mathematik

JOHANNES KEPLER
UNIVERSITÄT LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Inhaltsverzeichnis

Zusammenfassung

Schedulingprobleme treten in unserer Welt in den verschiedensten Bereichen auf. Egal ob in Krankenhäusern bei der Zuteilung von Patienten zu Behandlungsräumen, in Tischlereien und anderen Manufakturen bei der Abfolge von Arbeitsschritten auf Maschinen oder der Nutzung von Schienentrassen im Bahnverkehr. Das Lösen dieser Ablaufplanungsprobleme ist oftmals essentiell, um einen reibungsfreien Arbeitsalltag garantieren zu können oder um Produktionsprozesse zu optimieren.

Eine besonders häufig auftretende Variante dieser Schedulingprobleme ist das sogenannte Job-Shop Ablaufplanungsproblem, bei dem die Aufträge auf allen zur Verfügung stehenden Ressourcen in einer definierten Reihenfolge bearbeitet werden müssen. Für große Problemdaten mit einer hohen Anzahl an Aufträgen und Maschinen ist das Lösen dieser Probleme sehr aufwändig und im mathematisch-wissenschaftlichen Bereich noch nicht zur Gänze erforscht.

Eine Methode zur Lösung dieser Problemklasse ist die *Shifting-Bottleneck Heuristik*. Wie der Name schon verrät, handelt es sich dabei nicht um ein exaktes Verfahren, sondern um eine Heuristik, die zwar eine zulässige Lösung liefert, dies jedoch nicht unbedingt das Optimum sein muss. In der Regel sind die Lösungen dieses Verfahrens jedoch sehr gute Näherungen.

Dieses Verfahren löst mehrere kleinere Hilfsprobleme und gelangt nach m Iterationen zu einem Ergebnis, wobei m die Anzahl der Ressourcen bzw. Maschinen ist. Das Lösen dieser Hilfsprobleme geschieht zwar exakt, was Zeit und Rechenaufwand kostet, diese sind jedoch um einiges einfacher zu lösen als das Gesamtproblem und somit bleibt der Aufwand in einem akzeptablen Rahmen.

Die Basisversion der *Shifting-Bottleneck Heuristik* wurde im Rahmen dieser Arbeit in Python implementiert und mit Benchmarkproblemdaten aus der Literatur getestet.

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Bachelorarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Bachelorarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, 14.09.2018

Manuel Schlenkrich

Einleitung

Die Ablaufplanung ist seit den 1950er Jahren Forschungsgebiet der Mathematik und seither stetig gewachsen. Im englischen als Scheduling bezeichnet, beschäftigt man sich dabei mit der effizienten Nutzung von knappen Ressourcen zur Optimierung von Arbeitsabläufen. Da Arbeitsmittel wie Personal, Arbeitszeit, Maschinen oder Kapital in der Realität nur begrenzt verfügbar sind, stellt sich die Frage, wie man Arbeitsaufträge einplanen sollte, um diese Ressourcen bestmöglichst auszunutzen.

Diese Aufgabe erledigen sogenannte Scheduling-Verfahren oder zu deutsch Algorithmen zur Lösung von Ablaufplanungsproblemen. Solche Verfahren können sich je nach Problemstellung enorm unterscheiden und sind meist stark auf die jeweiligen Gegebenheiten zugeschnitten.

In dieser Bachelorarbeit wird zuerst das Ablaufplanungsproblem in seiner allgemeinen Form vorgestellt und danach werden verschiedene Varianten davon näher erläutert. Es werden auch grundlegende Verfahren zur Lösung dieser Probleme präsentiert um dem Leser einen groben Überblick über die Materie zu vermitteln.

Im vertiefenden Teil dieser Arbeit wird das Job-Shop Ablaufplanungsproblem detailliert untersucht. Dabei wird auf verschiedene Zielfunktionen und Nebenbedingungen eingegangen, wobei der Zusammenhang mit industriellen Aufgabenstellungen hergestellt wird.

Ein oft verwendetes Verfahren zur Lösung solcher Job-Shop Probleme ist das sogenannte *Shifting Bottleneck Verfahren*. Dieser Algorithmus und die dafür verwendeten Hilfsprobleme werden analysiert, nachdem die Problemstellung vollständig vorgestellt wurde. Eine Basisvariante dieses Schedulingverfahrens wird im Zuge dieser Bachelorarbeit in Python implementiert und mit Datensätzen aus Literatur und Praxis (Zuglogistik) getestet.

Einführungsbeispiel

Um sich besser in die Problematik dieses Themengebiets hineindenken zu können, wird an dieser Stelle ein erstes praktisches Beispiel präsentiert, an dem im Laufe dieser Arbeit einige Resultate veranschaulicht werden.

Ein kleines lokales Krankenhaus verfügt über ein Untersuchungszimmer *UZ*, einen Röntgenraum *XR*, einen Raum für Ultraschall Untersuchungen *UL*, einen Operationssaal *OP* und einen Raum zur Nachbesprechung der Untersuchung *NB*. Diese Ressourcen des Krankenhauses sind nur begrenzt verfügbar, da jeweils nur ein Patient pro Raum behandelt werden kann und dies eine gewisse Zeit in Anspruch nimmt. Natürlich ist das Krankenhaus daran interessiert, so vielen Menschen wie möglich helfen zu können - das Ziel ist also die Warte- und Gesamtbehandlungsdauern zu minimieren.

Bedingungen der realen Welt sind dabei, dass ein Patient nicht gleichzeitig in zwei oder mehreren Behandlungszimmern zur gleichen Zeit sein kann. Darüber hinaus erhält jeder Patient eine individuelle Reihenfolge, in der er oder sie die Behandlungszimmer besuchen muss, je nach Symptomen und Beschwerden.

In diesem Beispiel sollen 3 Patienten versorgt werden.

- Patient 0 hat sich beim Sport vermutlich den Arm gebrochen und muss zuerst zum Röntgen, danach in den Untersuchungsraum. Dann wird der Operationssaal zum Gipsen benötigt und es findet eine Nachbesprechung statt. Auch ein Ultraschall wird benötigt.

- Patient 1 erlitt eine schwere Schnittverletzung und benötigt sofort den Operationssaal. Danach muss geklärt werden, ob es Verletzungen am Knochen gab, deshalb wird auch der Röntgenraum gebraucht. Schließlich wird der Patient noch untersucht und zur Nachbesprechung geschickt. Abschließend muss routinemäßig ein Ultraschall gemacht werden.

- Patient 2 klagt über Schmerzen in der Schulter und muss zuerst zum Röntgen, danach zum Ultraschall und nach einer ärztlichen Untersuchung in den Operationssaal. Abschließend findet wieder eine Nachbesprechung statt.

Somit ergeben sich folgende Reihenfolgen:

- Patient 0: (XR, UZ, OP, NB, UL)

- Patient 1: (OP, XR, UZ, NB, UL)

- Patient 2: (XR, UL, UZ, OP, NB)

Gesucht wird nun ein optimaler Ablaufplan, der angibt welcher Patient zu welcher Zeit in welchem Raum behandelt wird. Dabei soll die Gesamtbehandlungszeit minimal sein.

Kapitel 1

Ablaufplanungsprobleme

1.1 Charakterisierung

Problemstellungen in der Ablaufplanung werden üblicherweise durch drei Parameter charakterisiert.

- Maschinenumgebung α
- Nebenbedingungen β
- Zielfunktion γ

Dabei wird in der Literatur die folgende Notation verwendet [vgl. 10].

$$\alpha|\beta|\gamma \tag{1.1}$$

Bezeichnungen

Folgende Liste enthält häufig verwendete Bezeichnungen in der Arbeit.
$p_{i,j}$ ... Bearbeitungszeit von Job j auf Maschine i
$r_{i,j}$ ... Ankunftszeit von Job j auf Maschine i (frühestmöglicher Beginn)
$s_{i,j}$ ... Startzeit von Job j auf Maschine i (fix eingeplant)
C_j ... Fertigstellungszeitpunkt von Job j
d_j ... Liefertermin von Job j
L_j ... Verspätung von Job j ($L_j = C_j - d_j$)
$L(m)$ ... Verspätung die auf Maschine m entsteht

Allgemeine Problemstellung

Für ein Scheduling-Problem benötigt man eine Menge von zu erledigenden Aufgaben, auch Jobs genannt. Diese Menge wird während der weiteren Arbeit als $J = \{1, \ldots, n\}$ bezeichnet. Zusätzlich benötigt man eine Menge an Maschinen (oder allgemeiner Ressourcen) auf denen diese Aufgaben erledigt werden sollen. Die Menge der Maschinen wird $M = \{1, \ldots, m\}$ genannt. Für jeden Job $j \in J$

gibt es eine Menge an Operationen $O_j = \{o_{1,j}, \ldots, o_{m,j}\}$, welche auf den Maschinen aus M ausgeführt werden sollen, um den jeweiligen Job abzuschließen. Die Dauer einer Operation $o_{i,j}$ auf einer Maschine heißt Bearbeitungszeit und wird mit $p_{i,j}$ bezeichnet. Die Lösung des Scheduling-Problems besteht darin, einen Ablaufplan zu erstellen, der unter Einhaltung von Nebenbedingungen die Zielfunktion optimiert. Ablaufplan bedeutet in diesem Zusammenhang die Festlegung eines Startzeitpunktes für jede Operation $o \in O_j$ für alle $j \in J$. Dabei wird der Startzeitpunkt von Operation $o_{i,j}$ mit $s_{i,j}$ bezeichnet. [vgl. 10].

Schedule

Die Lösung eines Ablaufplanungsproblems wird Schedule oder einfach Ablaufplan genannt und lässt sich als $m \times n$ Matrix schreiben. Der Eintrag $s_{i,j}$ gibt dabei den Startzeitpunkt der Operation $o_{i,j}$ an, also den Beginn der Bearbeitung von Job j auf Maschine i.

M \ J	1	2	...	j	...	n
1	$s_{1,1}$	$s_{1,2}$	*	$s_{1,j}$	*	$s_{1,n}$
2	$s_{2,1}$	$s_{2,2}$	*	$s_{2,j}$	*	$s_{2,n}$
$\vdots$	*	*	*	*	*	*
i	$s_{i,1}$	$s_{i,2}$	*	$s_{i,j}$	*	$s_{i,n}$
$\vdots$	*	*	*	*	*	*
m	$s_{m,1}$	$s_{m,2}$	*	$s_{m,j}$	*	$s_{m,n}$

Maschinenumgebung

Die Maschinenumgebung liefert Auskunft über die Anzahl der verfügbaren Maschinen (Ressourcen) und deren besondere Eigenschaften bezüglich Reihenfolge oder Art der Bearbeitung. Dabei kann man unterscheiden, ob ein Auftrag auf jeder Maschine bearbeitet werden muss, also die einzelnen Maschinen verschiedene Aufgaben erledigen. Der zweite Fall wäre, dass diese alle gleichartig funktionieren und es keine Rolle spielt auf welcher Maschine ein Auftrag bearbeitet wird. Der Parameter der Maschinenumgebung enthält auch die Information, ob eine mögliche Reihenfolge der Maschinen für alle Auftrge gleich sein muss oder ob sich diese unterscheiden können [vgl. 10].

Nebenbedingungen

Die Nebenbedingungen konkretisieren das Planungsproblem weiter, indem zahlreiche Bedingungen und Forderungen an die Bearbeitung der Aufträge im Parameter β beschrieben werden. Dabei kann es sich beispielsweise um Vorrang- oder Unterbrechungsbedingungen handeln.

Es kann vorkommen, dass bestimmte Jobs nicht von Beginn an bearbeitet werden können, sondern erst nach einer gewissen Zeit verfügbar sind. Dies wird als Ankunftszeit bezeichnet.

Muss ein Job $i \in J$ zuerst erledigt werden, bevor ein anderer Job $j \in J$ begonnen werden kann, handelt es sich um eine Vorrangbeziehung. Treten Nebenbedingungen dieser Form in der Problemdefinition auf, wird dies mit $\beta = prec$ für *precondition* vermerkt.

Ist es möglich, eine Operation eines Jobs auf einer Maschine zu unterbrechen und zu einem späteren Zeitpunkt ohne Verzögerung fortzusetzen, so spricht man von der Nebenbedindung *preemption*. In der Problemspezifikation wird dies mit $\beta = pmtn$ notiert. In realen Problemstellungen können diese Nebenbedingungen zahlreich auftreten, um die Komplexität der Wirklichkeit bestmöglich auf das Modell zu übertragen [vgl. 10].

Zielfunktion

Mit der Ziefunktion γ wird die Qualität eines berechneten Ablaufplans gemessen. Ziel ist es, den Ablaufplan so zu modifizieren, dass man einen *optimalen* Wert für die Zielfunktion erhält. Meist soll die Gesamtdurchlaufzeit eines Ablaufplans minimiert werden, es kann aber auch versucht werden, Verspätungen zu verhindern. Für besonders wichtige Aufträge kann man eine Zielfunktion auch durch eine Einführung von Gewichten anpassen [vgl. 10].

Definition 1.1 (Makespan, gewichteter Makespan)
Als Makespan oder Durchlaufzeit eines Ablaufplanungsproblems bezeichnet man den Zeitpunkt, an dem der letzte Job fertiggestellt wurde. Der Makespan wird mit C_{max} bezeichnet und es gilt $C_{max} := max\{C_j | j \in J\}$. Der gewichtete Makespan wird mit $C_{max,\omega}$ bezeichnet und es gilt $C_{max,\omega} := max\{\omega_j C_j | j \in J\}$.

Weniger anfällig auf einzelne Ausreißer als der Makespan eines Ablaufplans ist es, die Summe aller Fertigstellungszeitpunkte als Zielfunktion anzusetzen.

Definition 1.2 (Gesamtdurchlaufzeit, gewichtete Gesamtdurchlaufzeit)
$\sum_{j \in J} C_j$ wird als Gesamtdurchlaufzeit bezeichnet. $\sum_{j \in J} \omega_j C_j$ wird als gewichtete Gesamtdurchlaufzeit bezeichnet.

Wie bei der Durchlaufzeit kann man auch bei der Verspätung entweder die maximale Verspätung betrachten oder diese aufsummieren.

Definition 1.3 (Maximale Verspätung, gewichtete maximale Verspätung)
Die maximale Verspätung eines Ablaufplanungsproblems wird mit L_{max} bezeichnet und es gilt $L_{max} := max\{L_j | j \in J\}$. Die gewichtete maximale Verspätung wird mit $L_{max,\omega}$ bezeichnet und es gilt $L_{max,\omega} := max\{\omega_j L_j | j \in J\}$.

Definition 1.4 (Gesamtverspätung, gewichtete Gesamtverspätung)
$\sum_{j \in J} L_j$ wird als Gesamtverspätung bezeichnet. $\sum_{j \in J} \omega_j L_j$ wird als gewichtete Gesamtverspätung bezeichnet.

1.2 Varianten von Ablaufplanungsproblemen

1.2.1 Einmaschinenprobleme

Bei der einfachsten Form eines Ablaufplanungsproblems steht nur eine Maschine zur Verfügung, auf der Aufträge erledigt werden können. Demnach werden Entscheidungen bezüglich der Zuteilung von Ressourcen bei dieser Variante nicht benötigt. Es wird lediglich eine Reihenfolge gesucht, in der die Jobs auf der Maschine erledigt werden sollen [vgl. 10]. Auf das Einführungsbeispiel bezogen, ist ein Einmaschinenproblem ein Krankenhaus, das nur über einen Behandlungsraum verfügt.

Für Einmaschinenprobleme schreiben wir in Notation (1.1)

$$1|\beta|\gamma$$

1.2.2 Parallele Maschinen

Bei dieser Variante eines Ablaufproblems stehen m Maschinen zur Verfügung. Diese unterscheiden sich jedoch nicht in ihrer Funktion. Es macht also keinen Unterschied, welcher Auftrag auf welcher Maschine ausgeführt wird, da diese die selbe Aufgabe erledigen. Ein praktisches Beispiel aus der realen Welt wäre die Zuteilung von Computerprozessen zu Prozessoren. Es stellt sich jedoch heraus, dass Probleme dieser Art im Allgemeinen bereits für die geringe Anzahl von zwei Maschinen nicht in polynomieller Zeit lösbar sind. Man nennt solche Probleme auch NP-schwer. Nur unter speziellen Voraussetzungen an Zielfunktion und Nebenbedingungen ist ein solches Problem in polynomieller Zeit lösbar, also P-schwer [vgl. 10]. Auf das Einführungsbeispiel bezogen, ist ein Ablaufplanungsproblem mit parallelen Maschinen ein Krankenhaus, bei dem m gleichartige Behandlungsräume zur Verfügung stehen. Dies könnten beispielsweise 3 Operationssäle sein.

Für Ablaufplanungsprobleme mit parallelen Maschinen schreiben wir in Notation (1.1)

$$P_m|\beta|\gamma$$

Es hat sich bewährt für diese Variante des Ablaufplanungsproblems Verfahren mit einer sogenannten Prioritätsregel anzuwenden. Diese garantiert zwar keine optimale Lösung des Problems, im Allgemeinen werden jedoch gute Näherungen erzielt. Näher wird darauf im Abschnitt 1.3.1 eingegangen [vgl. 10].

1.2.3 Flow-Shops

Bei einem Flow-Shop Problem stehen ebenfalls m Maschinen zur Verfügung. Es wird aber nun davon ausgegangen, dass jeder Auftrag auf jeder der m Maschinen

ausgeführt werden muss. Die Maschinen unterscheiden sich also in ihrem Aufgabenbereich. Probleme mit dieser Eigenschaft werden als Shop Probleme bezeichnet. Ein weiteres wichtiges Merkmal eines Flow-Shop Planungsproblems ist, dass die Reihenfolge, in welcher die Aufträge die Maschinen besuchen müssen, für alle Aufträge gleich ist. Für den Fall, dass einzelne Aufträge eine Maschine nicht besuchen müssen wird dies mit einer Bearbeitungszeit von 0 modelliert [vgl. 10]. Auf das Einführungsbeispiel bezogen, ist ein Flow-Shop Problem mit m Maschinen ein Krankenhaus, in dem jeder Patient alle m Behandlungsräume in der selben Reihenfolge besuchen muss. Dies trifft etwa auf Routineuntersuchungen zu, die für jeden Patienten gleich sind.

Für Flow-Shop Problem mit m Maschinen schreiben wir in Notation (1.1)

$$F_m|\beta|\gamma$$

1.2.4 Job-Shops

Job-Shops sind eine Verallgemeinerung des Flow-Shop Ablaufplanungsproblems. Auch hier muss jeder Auftrag auf jeder der m Maschinen bearbeitet werden, es handelt sich also wieder um Maschinen, die sich in ihrer Funktion unterscheiden. Die Reihenfolge der Maschinen kann hier allerdings für jeden Auftrag unterschiedlich sein. Diese Reihenfolge ist fix vorgegeben und in der Problemstellung enthalten. Auch hier wird eine Maschine, die für einen Auftrag nicht benötigt wird mit Bearbeitungszeit 0 modelliert [vgl. 10]. Auf das Einführungsbeispiel bezogen ist ein Job-Shop Problem ein Krankenhaus, bei dem jeder Patient alle Behandlungsräume in einer vorgegeben Reihenfolge besuchen muss. Diese Reihenfolge kann für jeden Patienten unterschiedlich sein.

Für ein Job-Shop Problem mit m Maschinen schreiben wir in Notation (1.1)

$$J_m|\beta|\gamma$$

1.2.5 Open-Shops

Die letzte Variante des Ablaufplanungsproblems ist erneut eine Erweiterung eines Shop Problems. Bei Open-Shop Problem müssen wie schon beim Flow-Shop und Job-Shop alle Aufträge auf allen m Maschinen erledigt werden. Im Unterschied zum Job-Shop kann diese Reihenfolge aber für alle Aufträge beliebig gewählt werden und ist nicht durch die Problemstellung festgelegt [vgl. 10]. Auf das Einführungsbeispiel bezogen, ist ein Open-Shop Problem ein Krankenhaus, bei dem alle Patienten alle Behandlungsräume besuchen müssen, aber die Reihenfolge beliebig gewählt werden kann.

Für Open-Shop Probleme mit m Maschinen schreiben wir in Notation (1.1)

$$O_m|\beta|\gamma$$

1.3 Basisverfahren

In diesem Abschnitt werden einige grundlegende Verfahren zur Lösung von Ablaufplanungsproblemen vorgestellt. Zuerst werden einfache Regeln zur Planung von Einmaschinenproblemen und Problemen mit parallelen Maschinen präsentiert, die dann auf Shop Probleme erweitert werden.

1.3.1 Prioritätsregeln

Wie bereits bei den Ablaufplanungsproblemen mit parallelen Maschinen erwähnt sind Prioritätsverfahren einfache Verfahren zur Lösung von Schedulingproblemen. Diese können in der Regel gute Ablaufpläne erzeugen, auch wenn deren Optimalität nicht garantiert werden kann. Es kann keine Aussage über die Güte der Lösung eines solchen Verfahrens getroffen werden. Verfahren dieser Art werden Heuristiken genannt. Mit gewissen Anpassungen lassen sich diese Verfahren auch auf Shop Probleme anwenden.

Definition 1.5 (Prioritätsregel, Prioritätsplan)
Ein Sortierkriterium für Aufträge wird Prioritätsregel genannt. Gelangt man durch Sortieren der Aufträge nach einer Prioritätsregel und schrittweises Einplanen dieser Aufträge aus der geordneten Liste auf der nächsten verfügbaren Maschine, so nennt man den resultierenden Ablaufplan Prioritätsplan.

Hier ein Auszug einiger Prioritätsregeln.

Earliest Deadline First (EDF)
Die Aufträge werden *aufsteigend* nach ihren *Lieferterminen* sortiert. Diese Regel wird deshalb auch *Lieferterminregel* genannt. Der Job, welcher am frühesten fertiggestellt werden muss, ist also der erste in der Liste, der nächstspätere ist der zweite und so weiter. Dies ist eine naheliegende Regel und wird von den meisten "Per-Hand-Planern" verwendet [vgl. 1].

Shortest Job Next (SJN)
Die Aufträge werden *aufsteigend* nach *Bearbeitungszeit* sortiert. Kurz wird dieses als SJN oder auch als SPT für *Shortest Processing Time* bezeichnet. Diese Vorgehensweise kann sinnvoll sein, wenn man die Anzahl der verspäteten Aufträge minimieren möchte, da die Gefahr, dass einer der kürzeren Jobs zu spät kommt, sehr gering ist [vgl. 1].

Longest Job Next (LJN)
Jobs werden *absteigend* nach ihrer *Bearbeitungszeit* geordnet. Dies kann sinnvoll sein, wenn man eine besonders gleichmäßige Verteilung der Aufträge erreichen möchte, da man mit den kürzeren Jobs zum Schluss sehr flexibel planen kann [vgl. 1].

First Come First Served (FCFS)
Bei diesem Verfahren werden die Aufträge *aufsteigend* nach *Ankunftszeiten* sortiert. Der Job, der am frühesten zur Verfügung steht, wird also auch als erstes bearbeitet, ohne dabei den Liefertermin der Aufträge zu betrachten. Dies kann sinnvoll sein, wenn alle Aufträge etwa die gleiche Bearbeitungsdauer haben,

wird jedoch bei Ablaufplänen mit großen Differenzen der Bearbeitungszeiten zu Verspätungen führen [vgl. 1].

1.3.2 Least Laxity First Verfahren

Bisher wurden nur Verfahren betrachtet, bei denen zu Beginn eine Liste nach Prioritäten sortiert wurde, aus der dann Maschinen der Reihe nach zugeordnet wurden. Diese Liste blieb im Verlauf des Verfahrens fix. Beim sogenannten *Least Laxity First Verfahren* ("Verfahren des geringsten Spielraums") wird der nächste einzuplanende Job in jedem Iterationsschritt neu berechnet. Dieses Verfahren ist nur für Einmaschinenprobleme oder Probleme mit parallelen Maschinen geeignet. Dabei wird jener Auftrag j gewählt, dessen Spielraum $l_{1,j}$ bis zum Liefertermin am geringsten ist. Dieser Spielraum wird wie folgt berechnet.

$$l_{1,j} = d_j - r_{1,j} - p_{1.j}$$

$d_j \ldots$ Liefertermin von Job j
$r_{1,j} \ldots$ Ankunftszeit von Job j
$p_{1,j} \ldots$ Bearbeitungszeit von Job j

Aufträge werden immer so spät wie möglich eingeplant, sodass sie kurz vor ihrem Fälligkeitstermin fertiggestellt werden. Im Grunde ähnelt dieses Verfahren der Lieferterminregel, doch liegt der Vorteil darin, dass früher erkannt wird, wann sich ein Auftrag nicht mehr bis zur Deadline ausgehen wird. Nachteil ist jedoch auch erhöhter Rechenaufwand, da bei der Lieferterminregel nur einmal die Liste sortiert werden muss. Man sollte deshalb bei der Wahl des Verfahrens abwiegen, wie schwerwiegend sich eine Verspätung im Ablaufplan auswirkt.

1.3.3 Das Verfahren von Giffler und Thompson

Das Verfahren von Giffler und Thompson ist eine Heuristik zur Lösung von Job-Shop Problemen, die es erlaubt, die in Abschnitt 1.3.1 vorgestellten Prioritätsregeln auf Job-Shops anzuwenden. Dabei wird in jedem Iterationsschritt eine Maschine ausgewählt, auf der eine noch nicht eingeplante Aufgabe frühestmöglich abgeschlossen werden kann.

Dazu sei das Job-Shop Problem definiert über m Maschinen $M = \{1, \ldots, m\}$ und n Jobs $J = \{1, \ldots, n\}$ wobei man die Bearbeitung von Job j auf Maschine i Operation oder Aufgabe nennt und mit $o_{i,j}$ bezeichnet. Es lassen sich dadurch Ankunftszeiten $r_{i,j}$ bestimmen, ab der ein Job j auf Maschine i zur Verfügung steht, bedingt durch vorhergehende Operationen auf anderen Maschinen.

Die Bearbeitungszeit von Job j auf Maschine i wird mit $p_{i,j}$ bezeichnet und gibt die Dauer der Operation $o_{i,j}$ an. Zusätzlich wird eine Variable Z_i eingeführt, die den Zeitpunkt angibt, bis zu dem eine Maschine i belegt ist.

Es wird dann in jedem Iterationsschritt eine Maschine i^* gewählt, auf der eine noch nicht eingeplante Operation beendet werden kann und dann mittels einer Prioritätsregel eine Aufgabe zur Erledigung ausgesucht. Nach Einplanen werden Ankunftszeiten $r_{i,j}$ aktualisiert und die nächste Maschine ausgewählt.

Algorithmus 1.1 (Verfahren von Giffler und Thompson)

Eingabe: *Matrix ($p_{i,j}$) der Bearbeitungszeiten und Matrix ($R_{i,j}$)3 der Bearbeitungsreihenfolgen*

Ausgabe: *Matrix ($s_{i,j}$) der Startzeitpunkte*

1. *Bestimme aus den Bearbeitungsreihenfolgen und den Bearbeitungszeitpunkten die Ankunftszeiten $r_{i,j}$ für alle Jobs j auf allen Maschinen i. $Z_i = 0$.*

2. *Wiederhole folgende Schritte, bis alle Jobs vollständig auf den Maschinen eingeplant sind:*

3. *Bestimme eine Maschine i^* so, dass für ein j, das noch nicht auf i^* geplant ist, gilt*
   $\max\{Z_{i^*}, r_{i^*,j}\} + p_{i^*,j} = C^*$ *wobei*
   $C^* := min\{\max\{Z_i, r_{i,j}\} + p_{i,j} | Job\ j\ noch\ nicht\ auf\ Maschine\ i\ geplant\}$

4. *Wähle eine noch nicht eingeplante verfügbare Operation $o_{i^*,j}$ auf Maschine i^* nach einer Prioritätsregel aus Abschnitt 1.3.1 aus und plane diese ein. Setze also $s_{i^*,k}$ auf* $\max\{Z_i*, r_{i^*,k}\}$.

5. *Aktualisiere die Ankunftszeiten $r_{i,j}$. $Z_{i^*} = max\{Z_{i^*}, r_{i^*,k}\} + p_{i^*,k}$*

Kapitel 2

Das Job-Shop Problem

2.1 Problembeschreibung

Das Job-Shop Ablaufplanungsproblem ist eine Variante des Shop Problems. Für das Problem sind n Aufträge bzw. Jobs $J = \{1, \ldots, n\}$ gegeben, von denen jeder in einer eigenen, fix definierten Reihenfolge auf m Maschinen $M = \{1, \ldots, m\}$ bearbeitet werden muss. Das Bearbeiten eines Auftrags j auf einer Maschine i wird als Operation $o_{i,j}$ bezeichnet. Die Bearbeitungsreihenfolge kann für jeden Auftrag unterschiedlich sein. Aussagen bezüglich der Lösbarkeit und Komplexität für das Job-Shop Problem können auch für das Flow-Shop Problem genutzt werden, da dies nur ein Spezialfall ist, bei dem die Bearbeitungsreihenfolge für jeden Auftrag gleich ist. Dies ist einer der Gründe, warum in dieser Arbeit im besonderen das Job-Shop Problem analysiert wird.

Jeder Job j hat neben der Reihenfolge der Maschinen auch noch fix definierte Bearbeitungszeiten $p_{i,j}$, die auf jeder Maschine i für $i = 1, \ldots, m$ verbracht werden müssen. Demnach ist ein Job-Shop Planungsproblem durch zwei $m \times n$ Matrizen vollständig definiert [vgl. 10].

1. Die Matrix der Bearbeitungszeiten $(p_{i,j})$, wobei $p_{i,j} \in \mathbb{R}^+$ bzw. in den meisten Fällen $p_{i,j} \in \mathbb{N}$. Siehe Tabelle 2.1.

M \ J	1	2	...	j	...	n
1	$p_{1,1}$	$p_{1,2}$	*	$p_{1,j}$	*	$p_{1,n}$
2	$p_{2,1}$	$p_{2,2}$	*	$p_{2,j}$	*	$p_{2,n}$
$\vdots$	*	*	*	* *	*	
i	$p_{i,1}$	$p_{i,2}$	*	$p_{i,j}$	*	$p_{i,n}$
$\vdots$	*	*	*	*	*	*
m	$p_{m,1}$	$p_{m,2}$	*	$p_{m,j}$	*	$p_{m,n}$

Tabelle 2.1: Bearbeitungszeiten

2. Die Matrix der Reihenfolgen $(R_{i,j})$ der Maschinen für jeden Auftrag, wobei $R_{i,j} \in \{x \in \mathbb{N} : 1 \leq x \leq m\}$. Siehe Tabelle 2.2.

M \ J	1	2	...	j	...	n
1	$R_{1,1}$	$R_{1,2}$	$*$	$R_{1,j}$	$*$	$R_{1,n}$
2	$R_{2,1}$	$R_{2,2}$	$*$	$R_{2,j}$	$*$	$R_{2,n}$
$\vdots$	$*$	$*$	$*$	$* *$	$*$	
i	$R_{i,1}$	$R_{i,2}$	$*$	$R_{i,j}$	$*$	$R_{i,n}$
$\vdots$	$*$	$*$	$*$	$*$	$*$	$*$
m	$R_{m,1}$	$R_{m,2}$	$*$	$R_{m,j}$	$*$	$R_{m,n}$

Tabelle 2.2: Maschinenreihenfolgen

Aus diesen Tabellen kann auch eine weitere wichtige Tabelle abgeleitet werden, die in vielen Lösungsverfahren benötigt wird. Die Matrix der Ankunftszeiten $r_{i,j}$ von Job j auf Maschine i für $i = 1, \ldots, m$ und $j = 1, \ldots, n$. Diese Ankunftzeiten bestimmen ab welchem Zeitpunkt ein Auftrag auf einer Maschine zur Verfügung steht und bearbeitet werden kann, da bereits alle vorher notwendigen Maschinen besucht wurden. Dabei wird $r_{i,j}$ wie folgt berechnet. Sei *R(i,j)* der Eintrag der i-ten Zeile und j-ten Spalte der Maschinenreihenfolgematrix. Es sei $M(i,j) = \{k|R(k,j) < R(i,j)\}$ die Menge der Maschinen, welche von Job j vor Maschine i besucht werden müssen.

$$r_{i,j} = \sum_{k \in M(i,j)} p_{k,j} \tag{2.1}$$

Daraus ergibt sich die dritte $m \times n$ Matrix für das Job-Shop Problem in Tabelle 2.3, wobei $r_{i,j} \in \mathbb{R}^+$ bzw. in den meisten Fällen $r_{i,j} \in \mathbb{N}$

M \ J	1	2	...	j	...	n
1	$r_{1,1}$	$r_{1,2}$	$*$	$r_{1,j}$	$*$	$r_{1,n}$
2	$r_{2,1}$	$r_{2,2}$	$*$	$r_{2,j}$	$*$	$r_{2,n}$
$\vdots$	$*$	$*$	$*$	$*$	$*$	$*$
i	$r_{i,1}$	$r_{i,2}$	$*$	$r_{i,j}$	$*$	$r_{i,n}$
$\vdots$	$*$	$*$	$*$	$*$	$*$	$*$
m	$r_{m,1}$	$r_{m,2}$	$*$	$r_{m,j}$	$*$	$r_{m,n}$

Tabelle 2.3: Ankunftszeiten

Nun, da die wichtigsten Bestandteile eines Job-Shop Schedulingproblems definiert wurden, stellt sich natürlich die Frage, wie eine Lösung eines solchen Problems aussieht beziehungsweise was eigentlich gesucht wird.

Ein Schedulingverfahren für ein Job-Shop Problem soll einen Ablaufplan für jede der Maschinen liefern, also eine Reihenfolge der Aufträge mit jeweiligem Startzeitpunkt auf der Maschine. Dabei müssen die einzelnen Ablaufpläne der Maschinen in Summe zulässig sein. Es dürfen also weder zwei oder mehrere Jobs gleichzeitig auf der selben Maschine bearbeitet werden, noch darf ein Job gleichzeitig auf zwei oder mehreren Maschinen bearbeitet werden [vgl. 10].

Definition 2.1 (Job-Shop Schedule, Zulässiger Job-Shop Schedule)
Es sei ein Job-Shop Ablaufplanungsproblem gegeben, wobei $J = \{1, \ldots, n\}$ *die Menge der zu erledigenden Jobs und* $M = \{1, \ldots, m\}$ *die Menge der Maschinen angibt. Weiters sei* $O_j = \{o_{1,j}, \ldots o_{m,j}\}$ *die Menge der nötigen Operationen auf den Maschinen um den Job* $j \in J$ *fertigzustellen und* $p_{i,j}$ *sei die Dauer von Operation* $o_{i,j}$*.*

Eine Matrix $(s_{i,j})$ *wird Job-Shop Schedule genannt, wenn der Eintrag* $s_{i,j}$ *den Startzeitpunkt von Operation* $o_{i,j}$ *angibt und diese Operation für* $p_{i,j}$ *Zeiteinheiten bearbeitet wird.*

M \ J	1	2	...	j	...	n
1	$s_{1,1}$	$s_{1,2}$	*	$s_{1,j}$	*	$s_{1,n}$
2	$s_{2,1}$	$s_{2,2}$	*	$s_{2,j}$	*	$s_{2,n}$
⋮	*	*	*	*	*	*
i	$s_{i,1}$	$s_{i,2}$	*	$s_{i,j}$	*	$s_{i,n}$
⋮	*	*	*	*	*	*
m	$s_{m,1}$	$s_{m,2}$	*	$s_{m,j}$	*	$s_{m,n}$

Ein Job-Shop Schedule heißt zulässig*, wenn dadurch folgende Nebenbedingungen eingehalten werden.*

1. *Zwei oder mehrere Jobs dürfen nicht gleichzeitig auf der selben Maschine bearbeitet werden*
2. *Ein Job darf nicht gleichzeitig auf zwei oder mehreren Maschinen bearbeitet werden*

2.2 Repräsentation als disjunktiver Graph

Nachdem wir das Job-Shop Problem im vorigen Kapitel durch die Matrizen der Bearbeitungszeiten und Maschinenreihenfolgen vollständig definiert haben, stellt sich die Frage, wie ein Problem und dessen mögliche Lösung repräsentiert werden sollen. Das Problem wird durch einen disjunktiven Graph veranschaulicht. Die Lösung wird durch einen konjunktiven Graph repräsentiert, der nur noch gerichtete Kanten enthält [vgl. 3].

Definition 2.2 (Disjunktiver Graph für das Job-Shop Problem)
Ein gerichteter Graph für das Job-Shop Problem ist ein Graph $G = (V, E, \omega)$*,*

wobei V die Menge der Knoten, E die Menge der Kanten und ω die Gewichtung der Knoten ist. Dabei gilt:

$$V = M \times N \cup \{0, *\} \tag{2.2}$$

Dabei steht 0 für einen fiktiven Startknoten (Quelle) und $*$ für einen fiktiven Endknoten (Senke). Jeder Knoten ist gewichtet. Als Gewichtungsfunktion dient $\omega : M \times N \to \mathbb{R}, (i, j) \mapsto p_{i,j}$ und $\omega(0) = \omega(*) = 0$.

$$E = C \cup D \tag{2.3}$$

Wobei

- *C die Menge der gerichteten (konjunktiven) Kanten darstellt. Dies sind Kanten die nur in eine Richtung zeigen. Dies sind jene Kanten, die sich aus der Matrix ($R_{i,j}$) der Bearbeitungsreihenfolge auf den Maschinen ergeben. Besucht Job j Maschine i, nachdem Maschine k besucht wurde, entsteht eine Kante von Knoten (i, j) nach Knoten (k, j). Diese Kanten sind schon durch die Problemstellung festgelegt. Zusätzlich enthält C alle Kanten vom Startknoten 0 in Richtung der Knoten ohne Vorgänger und jene Kanten von den Knoten ohne Nachfolger Richtung Senke $*$.*

- *D die Menge aller nicht-gerichteten (disjunktiven) Kanten darstellt. Diese bestehen zwischen jenen Knoten, die Operationen auf der selben Maschine darstellen. Da die Bearbeitungsreihenfolge auf den Maschinen zu Beginn noch nicht bestimmt ist, ist auch die Richtung der Kanten noch nicht bestimmt.*

Ein solcher Graph könnte nun wie in Abbildung 2.1 aussehen.

In dieser Graphik bedeutet der Knoten $o_{i,j}$, dass Job j auf Maschine i ausgeführt wird. Zusätzlich ist über jedem Knoten dessen Bearbeitungszeit $p_{i,j}$ angegeben.

Während des Scheduling Prozesses sollen nun nach und nach die disjunktiven Kanten zu konjunktiven Kanten gemacht werden, es soll also die Richtung fixiert werden. Dadurch ergibt sich für jede Maschine eine eigene Belegungsreihenfolge.

Wenn alle Kanten zu gerichteten Kanten gemacht wurden, sieht der Graph und somit der fertige Ablaufplan wie in Abbildung 2.2 aus.

2.3 Das Shifting-Bottleneck Verfahren

Das Shifting-Bottleneck Verfahren ist eine Heuristik zur Minimierung der Gesamtbearbeitungszeit eines Job-Shops. Da es sich um eine Heuristik handelt, liefert dieses Verfahren nicht in jedem Fall eine optimale Lösung des Problems, doch sind die erzeugten Lösungen im Allgemeinen "sehr gute" Näherungen.

Wie bereits im vorigen Kapitel erläutert steht man bei Job-Shops vor dem Problem der Zuteilung von Jobs zu Maschinen. Dabei tauchen Engpässe unter den Ressourcen auf, es kommt also zu Konflikten bei der Bearbeitung auf

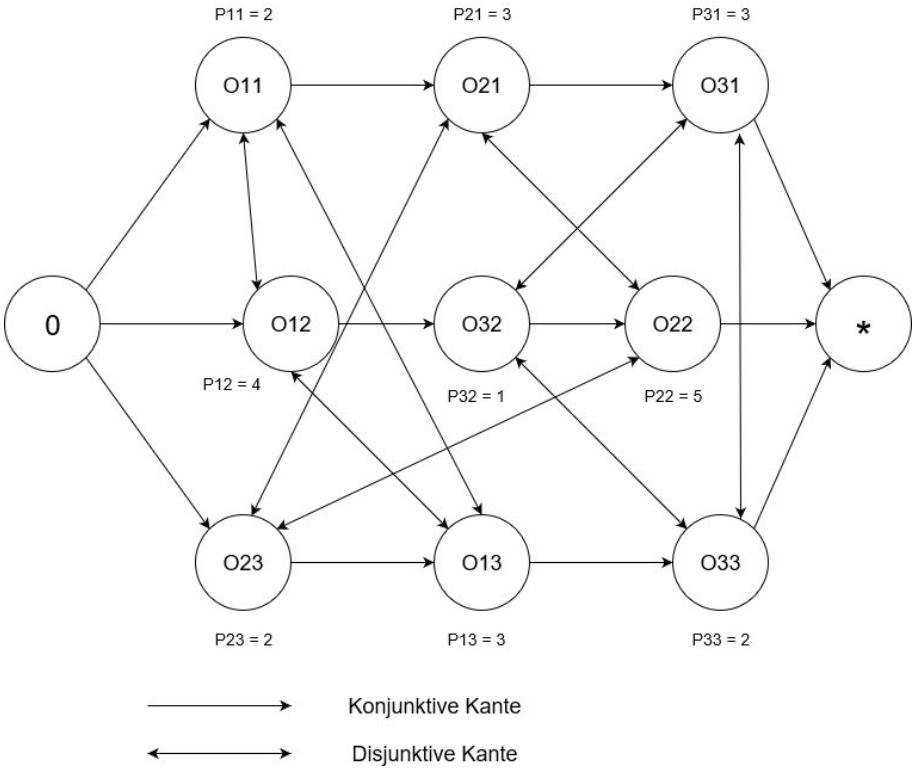


Abbildung 2.1: Disjunktiver Graph zur Veranschaulichung eines Job-Shop Problems

einzelnen Maschinen. In Anlehnung an die Verengung einer Flasche werden diese Engpässe auch als Flaschenhälse bezeichnet. Dies klärt den ersten Begriff "Bottleneck" im Namen des Verfahren.

Im Shifting-Bottleneck Verfahren werden nun diese Engpässe von Iteration zu Iteration für alle Maschine gelöst. In jedem Interationsschritt wird die Maschine gewählt, in der die maximale Verspätung nach aktuellem Stand am größten ist. Dann wird auf dieser Maschine der Engpass gelöst und eine optimale Bearbeitungsreihenfolge der Aufträge bestimmt. Somit verschiebt sich der Flaschenhals in jeder Iteration auf die nächste Maschine, bis nach m Iterationen alle Engpässe gelöst sind, wobei m die Anzahl der Maschinen ist [vgl. 1, 10].

2.3.1 Das 1-Maschinen Hilfsproblem

Eine Iteration des Shifting-Bottleneck Verfahrens besteht daraus, die Engpässe der Jobs für eine ausgewählte Maschine zu lösen. In der ersten Iteration wird die Maschine gewählt, für die die maximale Verspätung am größten ist.

Für die ausgewählte Maschine wird eine Branch-and-Bound Methode ange-

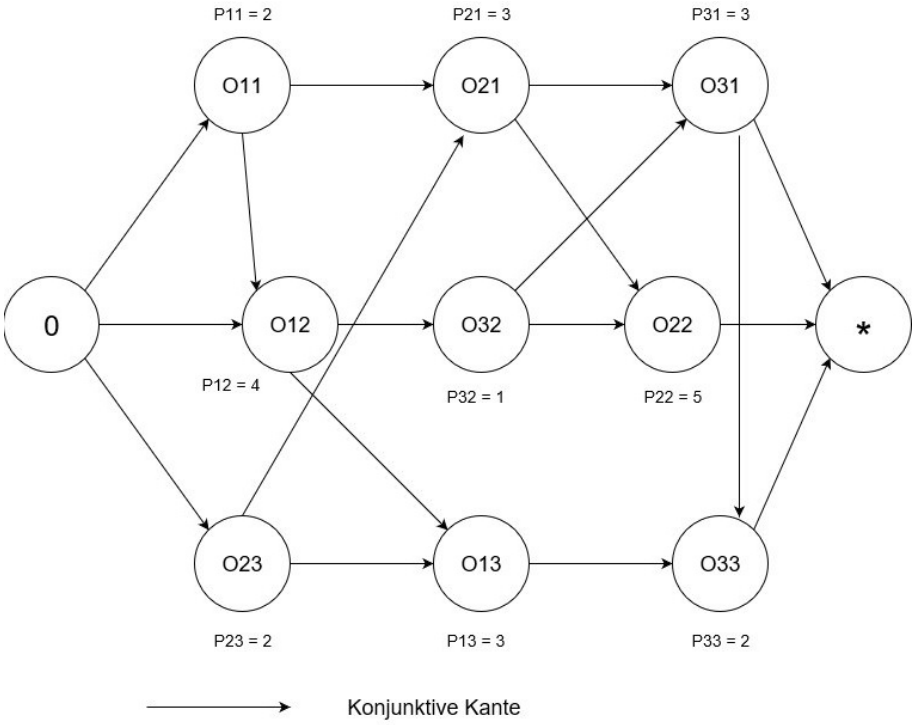


Abbildung 2.2: Konjunktiver Graph zur Veranschaulichung eines Job-Shop Problems

wandt um einen optimalen Ablaufplan für genau diese Maschine zu finden. Da es sich bei der Branch-and-Bound Methode um ein exaktes Verfahren handelt, findet man für dieses Teilproblem tatsächlich eine exakte Lösung.

Da bei diesem Hilfsproblem nur eine Maschine betrachtet wird, handelt es sich bei dieser Maschinenumgebung um eine 1-Maschinen Umgebung. Die Nebenbedingungen sind sogenannte Ankunftszeiten $r_{1,j}$. Diese Ankunftszeiten, also ab welchem Zeitpunkt ein Auftrag bearbeitet werden kann, berechnen sich aus den Bearbeitungszeiten der Jobs auf den vorhergehenden Maschinen. Die Zielfunktion bei diesem Teilproblem ist die maximale Verspätung der Aufträge auf dieser Maschine L_{max}. In der in Kapitel 1 eingeführten Notation handelt es sich also bei dem Hilfsproblem um ein Problem der Form

$$1|r_{i,j}|L_{max} \tag{2.4}$$

Wie wird nun der Engpass dieser Maschine gelöst? Dazu wird ein Branch-and-Bound Verfahren verwendet. Dazu wird zuerst eine obere Schranke der maximalen Verspätung definiert, indem ein Ablaufplan mittels der in Kapitel 1.3.1 definierten Prioritätsregel *First Come First Served* erstellt wird. Die Jobs werden also in der Reihenfolge ihrer Ankunftszeit auf der Maschine eingeordnet.

Für den Fall, dass bei dieser einfach zu berechnenden Lösung die Verspätung bereits 0 betragen würde, bräuchte man nicht mehr weiterrechnen und hätte bereits eine Lösung für das 1-Maschinen Hilfsproblem gefunden. In den meisten Fällen wird dies jedoch nicht der Fall sein, da die *First Come First Served* Prioritätsregel keine Bearbeitungszeiten und Liefertermine bei der Auswahl des nächsten Jobs berücksichtigt.

Aus diesem Grund wird zusätzlich zum Upper Bound auch ein Lower Bound für das Hilfsproblem berechnet. Der Lower Bound gibt eine untere Schranke für die maximale Verspätung des Ablaufplans an. Dieser Wert kann nicht unterschritten werden, da es keine bessere zulässige Lösung gibt. Wenn Upper Bound und Lower Bound den gleichen Wert haben, kann man das Verfahren beenden. Doch woher weiß man, wie eine solche untere Schranke aussieht? Dafür muss man sich noch eines zusätzlichen Hilfsproblems bedienen, welches unter gelockerten Nebenbedingungen gelöst werden soll. Man nennt dieses Problem auch relaxiertes 1-Maschinen Hilfsproblem [vgl. 10].

2.3.2 Das relaxierte 1-Maschinen Hilfsproblem

Das relaxierte Hilfsproblem ist bis auf eine Änderung bei den Nebenbedingungen gleich definiert wie das 1-Maschinen Hilfsproblem aus dem vorhergehenden Abschnitt. Der wesentliche Unterschied besteht darin, dass Aufträge beliebig oft während ihrer Bearbeitung unterbrochen werden dürfen. Dies wird durch das Symbol *pmtn* für *preemption* dargestellt. Es handelt sich also um ein Problem der Form

$$1|pmtn, r_{i,j}|L_{max} \tag{2.5}$$

Eine Lösung des relaxierten Problems im Allgemeinen nicht zulässig für das Hilfsproblem, da dort die Jobs nicht unterbrochen werden dürfen. Jedoch erkennt man, dass jede zulässige Lösung des Hilfsproblems auch eine zulässige Lösung des relaxierten Hilfsproblems ist, da es sich nur um einen Spezialfall handelt, bei dem die Unterbrechungen nicht in Anspruch genommen werden. Das Minimum der maximalen Verspätungen des relaxierten Problems ist demnach niemals größer, als das des allgemeinen Hilfsproblems. Aus diesem Grund kann die Lösung des relaxierten Problems als untere Schranke für den Branch-and-Bound Algorithmus verwendet werden.

Sobald man Upper und Lower Bound bestimmt hat, kann der Branch-and-Bound Algorithmus mit der Suche nach der optimalen Lösung des 1-Maschinen Problems beginnen. Zunächst wird der erste Auftrag als fix eingeplant, dieser soll also zur Gänze ohne Unterbrechnung bearbeitet werden. Für die restlichen Aufträge wird dann das relaxierte Problem betrachtet. Man versucht also die anderen Jobs bestmöglich einzuplanen, wobei Unterbrechungen erlaubt sind. Erhält man dafür einen Wert, der über dem Upper Bound liegt, kann der Branch, an dem der erste Auftrag zuerst eingeplant wird, verworfen werden. Erhält man einen niedrigeren Wert muss man den nächsten Job als fix einplanen und erneut das relaxierte Problem betrachten. Dies macht man, bis ein

zulässiger Ablaufplan entsteht, dessen maximale Verspätung unter dem Upper Bound liegt. Dieser Wert wird dann als neue obere Schranke gewählt. Wird kein solcher zulässiger Plan gefunden, kann der untersuchte Branch verworfen werden. Durch die ständige Anpassung des Upper Bounds werden Branches immer schneller verworfen und man erhält schlussendlich eine optimale Lösung für das 1-Maschinen Hilfsproblem. Findet man während der Suche einen zulässigen Ablaufplan ohne Unterbrechnungen mit einer maximalen Verspätung des Lower Bounds kann man den Algorithmus natürlich sofort abbrechen, da man ab diesem Zeitpunkt keine bessere Lösung mehr finden kann [vgl. 10].

2.3.3 Rescheduling

Durch das Lösen des Hilfsproblems auf nur einer Maschine ohne Betrachtung des gesamten Problems können durch das Neu-Einplanen einer Maschine Konflikte mit den bereits eingeplanten Maschinen entstehen. Diese Konflikte können entweder Überschneidungen sein, aber auch Reihenfolgekonflikte. Ist dies der Fall, so müssen bereits eingeplante Maschinen mit den durch die gerade eingeplante Maschine erzeugten aktualisierten Daten noch einmal eingeplant werden. Dafür wird erneut das 1-Maschinen Hilfsproblem mit aktualisierten Ankunftszeiten berechnet. Dieser Vorgang wird als Rescheduling bezeichnet [vgl. 1, 10]

2.3.4 Vorrangbeziehungen

Es ist zwischen zwei Arten von Vorrangbeziehungen zu unterscheiden.

1. $(i,k) \rightarrow (j,k)$ für $k \in \{1,\ldots,n\}$ und $i,j \in \{1,\ldots,m\}$ mit $i \neq j$

   Es bedeutet, dass der Job k zuerst auf Maschine i bearbeitet werden muss, bevor er für Maschine j zur Verfügung steht. Diese Art von Vorrangbeziehung ist bereits in der Definition des Job-Shop Problems enthalten, da die Maschinenreihenfolge für jeden Auftrag gegeben ist.

2. $(k,i) \rightarrow (k,j)$ für $i,j \in \{1,\ldots,n\}$ mit $i \neq j$ und $k \in \{1,\ldots,m\}$

   Es soll für jede Maschine k eine optimale Reihenfolge der Aufträge gefunden werden. Diese Art von Vorrangbeziehung ist in der Problemstellung noch nicht enthalten, sondern soll über das Lösungsverfahren bestimmt werden.

Da nach jedem Iterationsschritt des Shifting-Bottleneck Verfahrens der Engpass einer Maschine gelöst wurde, können auf dieser Lösung basierende Vorrangbeziehungen für die jeweilige Maschine definiert werden. Somit entstehen nach m Iterationen, also wenn die Engpässe für alle Maschinen gelöst wurden, $m*(n-1)$ neue Vorrangbeziehungen des zweiten Typs [vgl. 10]. Somit erhält man nach m Iterationen eine Menge von $n*(m-1)$ Vorrangbeziehungen des ersten Typs und $m*(n-1)$ des zweiten Typs, insgesamt also $2mn-m-n$ Vorrangbeziehungen, aus denen sich ein vollständiger Ablaufpan konstruieren lässt.

2.3.5 Formale Definition

Algorithmus 2.1 (Shifting-Bottleneck Heuristik)

Eingabe: *Matrix $(p_{i,j})$ der Bearbeitungszeiten und die Matrix $(R_{i,j})$ der Maschinenreihenfolgen.*

Ausgabe: $m * (n-1)$ *Vorrangbeziehungen der Form* $(k,i) \rightarrow (k,j)$

1. ***Initialisierung:*** *Sei M die Menge aller Ressourcen. Setze $S = \emptyset$ die Menge aller Maschinen, die bereits eingeplant wurde. Definiere die Menge der Vorrangbeziehungen, die durch die Aufträge in der Problemstellung bestimmt wurden. $Prec = (k,j) \rightarrow (l,j)$, so dass $R_{k,j} = x$ und $R_{l,j} = x+1$ für alle $x \in \{1,\ldots,m-1\}$ für alle Jobs $j \in J$. Setze darüber hinaus die aktuelle Gesamtbearbeitungszeit C_{akt} auf das Maximum der Gesamtbearbeitungszeiten der Jobs in J.*
   $C_{akt} = max_{j \in J} \sum_{i=1}^{m} p_{i,j}$

2. ***Iteration:*** *Wiederhole folgenden Schritt bis $S = M$.*

3. ***Einplanen der Engpassmaschine:*** *Berechne für alle Maschinen $m \in M \setminus S$, die Verspätung $L(m)$ mittels des in Abschnitt 2.3.1 vorgestellten 1-Maschinen Hilfsproblems. Wähle die Maschine m^*, deren Verspätung $L(m^*)$ am größten ist und bestimme die ideale Reihenfolge $(j_k)_{k \in J}$ der Aufträge auf m^*. Füge Vorrangbeziehungen der idealen Bearbeitungsreihenfolge der Form $(m^*, j_k) \rightarrow (m^*, j_{k+1})$ zur Menge Prec hinzu.*
   Setze $S = S \cup \{m^*\}$ und berechne neue Bearbeitungszeit $C_{akt} = max_{j \in J}(max_{i \in M}(r_{i,j} + p_{i,j}))$ mit aktualisierten Ankunftszeiten $r_{i,j}$, die sich durch Prec ergeben.

4. ***Rescheduling:*** *Sollten durch die gerade eingeplante Maschine m^* Reihenfolge- oder Überschneidungskonflikte mit bereits geplanten Maschinen $m \in M$ entstanden sein, löse für diese Maschinen m das Hilfsproblem mit den durch m^* erzeugten aktualisierten Ankunftszeiten neu, plane diese Maschinen erneut ein und aktualisiere die Vorrangbeziehungen in Prec [vgl. 10].*

2.3.6 Anwendung am Einführungsbeispiel

Zu Beginn dieser Arbeit wurde ein anschauliches Beispiel präsentiert, indem ein Ablaufplan für die Behandlung von Patienten in verschiedenen Krankenhausräumlichkeiten erstellt werden sollte.

Mit dem bereits erworbenen Wissen kann man diese Problemstellung näher einordnen. Jeder der Patienten kann in allgemeiner Betrachtungsweise als Job oder Auftrag gesehen werden. Die Behandlungsräume stellen die Ressourcen bzw. Maschinen dar. In der Problemstellung müssen alle Patienten jeden Raum genau einmal besuchen. Man kann dieses Problem also bereits in die Klasse der *Shop Probleme* einordnen.

Darüber hinaus sind die Reihenfolgen der zu besuchenden Räume für jeden Patienten unterschiedlich und fix gegeben. Es handelt sich demnach nicht um einen *Flow-Shop Problem*, sondern um ein *Job-Shop Problem*.

Das Einführungsbeispiel kann also wie in Abschnitt 2.1 durch die beiden Matrizen der Bearbeitungszeiten und Maschinenreihenfolge vollständig definiert werden.

$p_{i,j}$	$Patient_0$	$Patient_1$	$Patient_2$
UZ	15	60	5
XR	30	25	20
UL	30	45	5
OP	40	30	40
NB	15	20	20

Tabelle 2.4: Behandlungszeiten

$R_{i,j}$	$Patient_0$	$Patient_1$	$Patient_2$
UZ	2	3	3
XR	1	2	1
UL	5	5	2
OP	3	1	4
NB	4	4	5

Tabelle 2.5: Behandlungsreihenfolge

Daraus kann man leicht die Matrix der Ankuftszeiten ableiten.

$r_{i,j}$	$Patient_0$	$Patient_1$	$Patient_2$
UZ	30	55	25
XR	0	30	0
UL	100	135	20
OP	45	0	30
NB	85	115	70

Tabelle 2.6: Ankunftszeiten

Nun wenden wir das Shifting-Bottleneck Verfahren an, um einen Ablaufplan für dieses Problem zu erstellen. Sei dazu $S = \emptyset$ die Menge der Ressourcen (Behandlungsräume), die bereits vollständig eingeplant wurden.

Es wird zunächst die erste Engpassressource bestimmt, indem für jede Ressource das 1-Maschinen Hilfsproblem gelöst wird und dabei die maximale Verspätung berechnet wird.

Um die maximale Verspätung pro Behandlungsraum berechnen zu können, wird zunächst die längste Behandlungszeit der Patienten C_{akt} benötigt. Diese wird für $Patient_1$ gebraucht und beträgt 180. Ausgehend davon kann man die Liefertermine für die Jobs auf den einzelnen Ressourcen berechnen, zu denen sie

im jeweiligen Raum fertiggestellt sein sollten, um die aktuelle Bestzeit C_{akt} von 180 nicht zu verspäten. Man stellt sich also für jedes der Behandlungszimmer (Ressource) eine Matrix mit den Kennzahlen Bearbeitungszeit p_j, Ankunftszeit $r_{i,j}$ (basierend auf vorhergehenden Behandlungen in anderen Räumen) und Lieferzeitpunkt d_j zu dem spätestens fertiggestellt werden sollte, um keine Verspätung zu erzeugen. Exemplarisch werden diese Matrizen hier für den OP und das Untersuchungszimmer UZ aufgestellt.

	OP		
	$Patient_0$	$Patient_1$	$Patient_2$
$p_{OP,j}$	40	30	40
$r_{OP,j}$	45	0	30
$d_{OP,j}$	135	30	160

Optimale Reihenfolge für den OP: (P_1, P_2, P_0)
$L(OP) = 0$

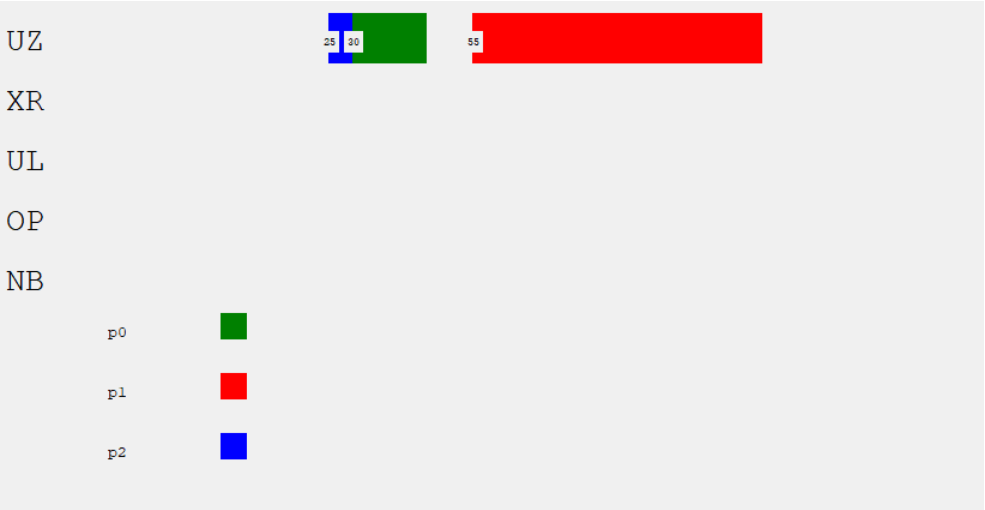
	UZ		
	$Patient_0$	$Patient_1$	$Patient_2$
$p_{UZ,j}$	15	60	5
$r_{UZ,j}$	30	55	25
$d_{UZ,j}$	95	115	120

Optimale Reihenfolge für das UZ: (P_2, P_0, P_1)
$L(UZ) = 0$

Auch bei den anderen Behandlungsräumen kommt es zu keiner Verspätung, also kann eine Ressource zufällig ausgewählt werden. $m^* = UZ$.

Die Patienten werden in der optimalen Reihenfolge (P_2, P_0, P_1) auf der Ressource UZ eingeplant und Vorrangbeziehungen $(UZ, P_2) \rightarrow (UZ, P_0)$, $(UZ, P_0) \rightarrow (UZ, P_1)$ werden gespeichert. Startzeiten auf der Ressource sind entweder wenn möglich die jeweilige Ankunftszeit $r_{i,j}$ oder wenn zu diesem Zeitpunkt ein Vorgänger noch nicht beendet wurde, der nächste frei Startpunkt nach dem Abschluss des Vorgängers.

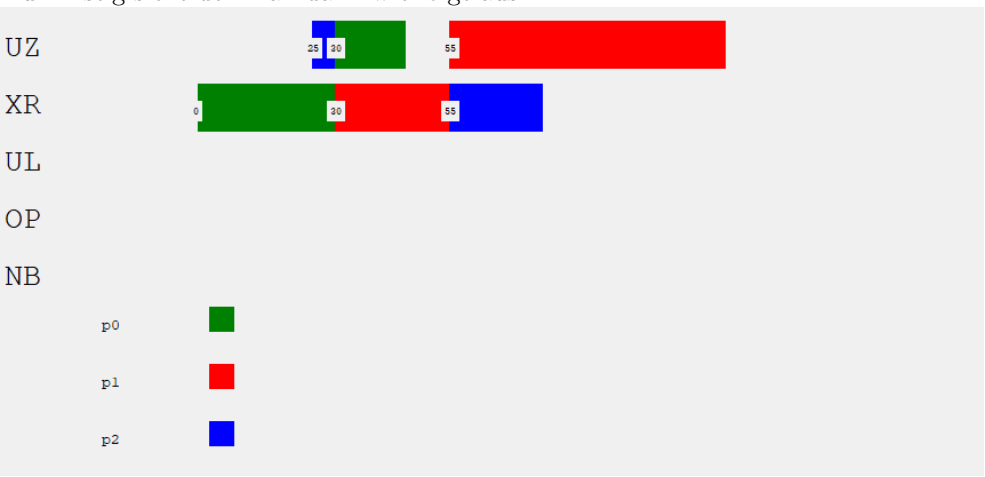
Somit sieht der Ablaufplan nach der ersten geplanten Ressource wie folgt aus:



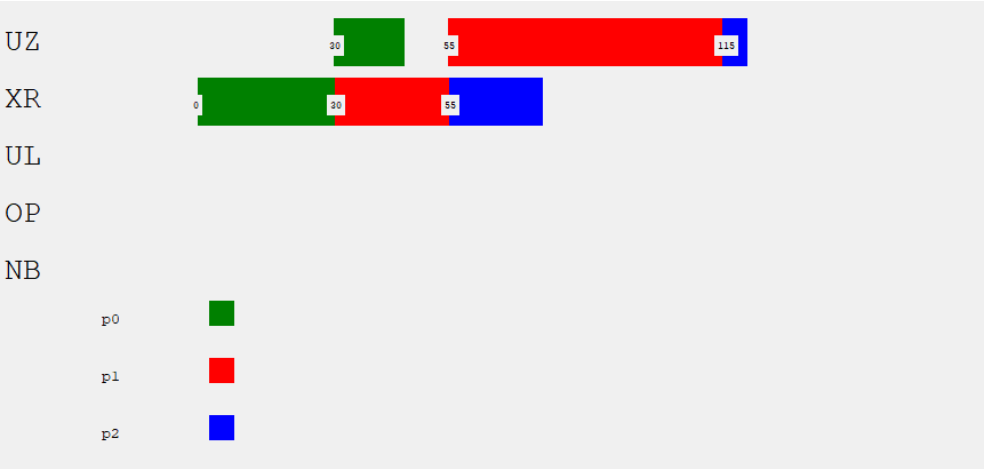
Setze nun die Menge der bereits geplanten Räume $S = S \cup \{UZ\} = \{UZ\}$ und bestimme die nächste Engpassressource.

Als zweite Ressource wird der Röntgenraum XR bestimmt und die optimale Reihenfolge von (P_0, P_1, P_2) bestimmt. Setze $S = S \cup \{XR\} = \{UZ, XR\}$.

Kurzfristig sieht der Plan dann wie folgt aus:



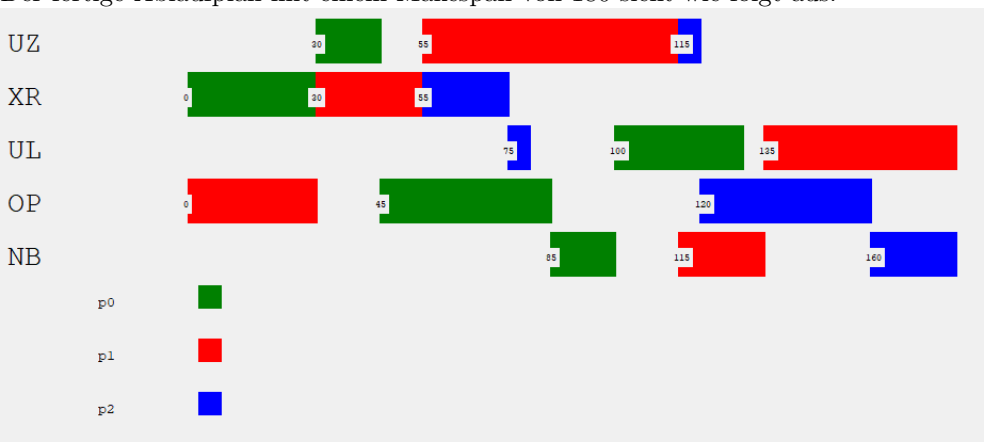
Da allerdings für $Patient_1$ die Reihenfolge der Behandlungsräume nicht korrekt ist, da zuerst das Untersuchungszimmer und dann erst der Röntgenraum besucht wird, muss im Rescheduling das bereits eingeplante Untersuchungszimmer mit neuen Ankunftszeiten erneut eingeplant werden. Diese lauten $(30, 55, 75)$.



Nun ist der Ablaufplan zulässig und die nächste Engpassressource wird ausgewählt. Es wird wieder die optimale Reihenfolge bestimmt und eingeplant. Es werden Vorrangbeziehungen definiert und die Startzeitpunkte wie oben beschrieben festgelegt. Nach dem Einplanen einer neuen Ressource wird kontrolliert, ob es Konflikte mit den bereits geplanten Räumen gibt und diese werden bei Bedarf erneut eingeplant.

Dieser Vorgang wird solange wiederholt, bis alle Behandlungsräume durchgeplant sind.

Der fertige Ablaufplan mit einem Makespan von 180 sieht wie folgt aus:



Der kritische Patient ist P_1 mit einer Behandlungszeit von 180. Seine Behandlungen werden durch die roten Balken gekennzeichnet und er wird durchgehend ohne Wartezeit untersucht. Patient P_0 hat eine Durchlaufzeit von 130 und somit ebenfalls keine Wartezeit. Der Patient P_2 hat mit einer Durchlaufzeit von 180 eine Wartezeit von 90. Die individuelle Reihenfolge für jeden Patienten wird eingehalten und der Makespan entspricht der kritischen Behandlunsgzeit von 180, demnach ist der Ablaufplaufplan nicht nur zulässig, sondern auch optimal.

2.3.7 Benchmark Testbeispiele

Um Aussagen über die Qualität des Verfahrens machen zu können, wurde das Shifting-Bottleneck Verfahren im Rahmen dieser Arbeit in Python implementiert. Es wurde mit ausgewählten Benchmark Testbeispielen verschiedenster Größen von *http://jobshop.jjvh.nl/* auf Lösungsqualität und Laufzeit getestet.

1. Benchmark Instanz von **Lawrence** (la01):
   Problemgröße: 10 Jobs auf 5 Maschinen
   Optimaler Makespan: 666
   Shifting-Bottleneck Verfahren:
   Erreichter Makespan: 671
   Abweichung vom Optimum: 0,75 %
   Berechnungsdauer: 0,062 Sekunden

2. Benchmark Instanz von **Adams, Balas und Zawack** (abz5):
   Problemgröße: 10 Jobs auf 10 Maschinen
   Optimaler Makespan: 1234
   Shifting-Bottleneck Verfahren:
   Erreichter Makespan: 1334
   Abweichung vom Optimum: 8,1 %
   Berechnungsdauer: 0,266 Sekunden

3. Benchmark Instanz von **Adams, Balas und Zawack** (abz8):
   Problemgröße: 20 Jobs auf 15 Maschinen
   Optimaler Makespan: 648
   Shifting-Bottleneck Verfahren:
   Erreichter Makespan: 798
   Abweichung vom Optimum: 23,15 %
   Berechnungsdauer: 1,625 Sekunden

4. Benchmark Instanz von **Taillard** (ta80):
   Problemgröße: 100 Jobs auf 20 Maschinen
   Optimaler Makespan: 5183
   Shifting-Bottleneck Verfahren:
   Erreichter Makespan: 5432
   Abweichung vom Optimum: 4,8 %
   Berechnungsdauer: 106,16 Sekunden

Selbst bei großen Problemdaten mit 2000 Operationen (100 Jobs auf 20 Maschinen) liefert die Shifting-Bottleneck Heuristik akzeptable Näherungen zum Optimum (Abweichung < 5 %) in kurzer Zeit (< 2 Minuten).

2.4 Nachbarschaftsbasierte Verfahren

Um ein Job-Shop Scheduling Problem zu lösen, gibt es neben dem Shifting-Bottleneck Verfahren noch zahlreiche weitere Möglichkeiten. Da die Art des Problems allerdings sehr komplex ist, sind die meisten dieser Methoden keine exakten Verfahren sondern Heuristiken, wie auch das Shifting-Bottleneck Verfahren. Die Lösung ist also nicht garantiert optimal.

Besonders etabliert haben sich unter den Lösungsmethoden für Job-Shop Probleme sogenannte *Local Search Algorithmen*. Dabei wird in einer lokalen Umgebung einer aktuellen Lösung, auch *Nachbarschaft* genannt, nach einer noch besseren gesucht. Um auf diese Verfahren näher eingehen zu können, muss zuerst der Begriff der Startlösung erklärt werden [vgl. 16].

2.4.1 Startlösungen

Um einen Local Search Algorithmus anwenden zu können, muss zuerst eine Startlösung existieren, auf deren Basis dann nach und nach bessere Lösungen gesucht werden.

Definition 2.3 (Startlösung)
Es sei ein Scheduling Problem P gegeben. Eine Startlösung zur Lösung dieses Problems mittels lokaler Suche ist ein zulässiger Ablaufplan des Problems P. Grundsätzlich gibt es bis auf Zulässigkeit keine Einschränkung an die Startlösung, doch im Allgemeinen gilt, je besser diese gewählt wird, desto geringer ist der Rechenaufwand für nachfolgende Local Search Algorithmen.

Da jede zulässige Lösung verwendet werden kann, stellt sich die Frage, wie man mit möglichst wenig Aufwand zu einer Startlösung gelangt, die auch zugleich ein guter Startpunkt für den daurauf folgenden Local Search Algorithmus ist.

Dafür eignen sich die in Kapitel 1.3.1 vorgestellten Prioritätsregeln, angepasst an das Job-Shop Problem. Das Prinzip war, die Jobs nach einem bestimmten Kriterium zu sortieren und diese dann nach und nach auf die Maschinen zu verteilen. Da sich beim Job-Shop Problem die Maschinen hinsichtlich ihrer Funktionsweise unterscheiden (anders als bei den parallelen Maschinen) kann nicht die nächste freie Ressource gewählt werden. Es wird aufsteigend der Sortierung nach jeweils die Maschine zugeteilt die als nächstes an der Reihe ist ab dem Zeitpunkt, ab der sie wieder frei ist. Bereits zugeteilte Jobs an Maschinen werden nicht mehr verändert. Dadurch wird die Lösung natürlich nicht optimal sein. Das ist aber auch nicht die Aufgabe der Startlösung, denn der Local Search Algorithmus wird im weiteren Verlauf Schwachstellen des Ablaufplans erkennen und diese ausbessern [vgl. 16].

Algorithmus 2.2 (Generieren einer Startlösung)

Eingabe: *Matrix $(p_{i,j})$ der Bearbeitungszeiten und die Matrix $(R_{i,j})$ der Maschinenreihenfolgen.*

Ausgabe: *Matrix $(s_{i,j})$ der Startzeiten*

1. *Sortiere alle Operationen $o_{i,j}$ aller Jobs $j \in J$ nach einem Kriterium der in Abschnitt 1.3.1 vorgestellten Prioritätsregeln.*

2. *Sei $S = \{1, \ldots, n * m\}$ die sortierte Liste der Operationen aller Jobs. Wähle die erste Operation $o_{i,j} \in S$, die bereits verfügbar ist. Sei Z_i die momentane Belegzeit von Maschine i, also jener Zeitpunkt, bis zu dem der späteste Auftrag auf dieser Maschine ausgeführt wird. Sei $r_{i,j}$ die Ankunftszeit von Job j auf Maschine i, die sich durch die Bearbeitungszeiten und Reihenfolgen ergibt. Plane sodann $o_{i,j}$ auf Maschine i ein, setze also $s_{i,j} = \max\{Z_i, r_{i,j}\}$.*

3. *Setze $S = S \setminus \{o_{i,j}\}$. Wenn S nicht leer ist gehe zu Schritt 2, andernfalls ist die Startlösung fertig.*

2.4.2 Lokale Suche

Bevor Verfahren der lokalen Suche näher erläutert werden können, wird zunächst die Definition der Nachbarschaft gebraucht.

Definition 2.4 (Nachbarschaft)
Sei L die Menge aller zulässigen Lösungen eines Scheduling Problems P. Eine Abbildung N, die jeder Lösung $a \in L$ eine Menge $N(a) \subseteq L$ zuordnet, nennt sich Nachbarschaftsfunktion. Elemente $b \in N(a)$ werden Nachbarn von a genannt.

Die Nachbarschaftsfunktion N ist ein wesentlicher Teil eines lokalen Suchalgorithmus, da durch sie bestimmt wird, welche anderen Lösungen untersucht werden.

Definition 2.5 (Modifikation)
Sei L die Menge aller zulässigen Lösungen eines Scheduling Problems P und N eine Nachbarschaftsfunktion. Das Erzeugen einer neuen Lösung $b \in N(a)$ wird als Modifikation von a zu b bezeichnet.

Interessant ist dabei, ob man durch Modifikation von einer zulässigen Lösung a zu einer anderen b den Zielfunktionswert verbessert [vgl. 16].

Definition 2.6
Sei L die Menge aller zulässigen Lösungen eines Scheduling Problems P. Sei γ die zu minimierende Zielfunktion von P. Eine Modifikation von $a \in L$ zu $b \in L$ heißt positive Modifikation genau dann, wenn $\gamma(b) < \gamma(a)$.

Sie wird negative Modifikation genannt wenn gilt $\gamma(b) > \gamma(a)$. Im Falle der Gleichheit spricht man von einer neutralen Modifikation.

Ziel eines lokalen Suchalgorithmus ist es nun, iterativ positive Modifikation ausgehend von einer Startlösung durchzuführen. Diese Algorithmen können sich einerseits dadurch unterscheiden, welche anderen Lösungen betrachtet und

verglichen werden — also durch die Definition der Nachbarschaftsfunktion. Andererseits auch dadurch, ab wann eine Modifikation akzeptiert und als neue Lösung übernommen wird.

Die Grobstruktur eines Local Search Algorithmus lässt sich formal wie folgt beschreiben.

Algorithmus 2.3 (Iterative Lokale Suche)

Eingabe: *Matrix $(p_{i,j})$ der Bearbeitungszeiten und die Matrix $(R_{i,j})$ der Maschinenreihenfolgen.*

Ausgabe: *Matrix $(s_{i,j})$ der Startzeiten*

1. *Generiere eine Startlösung a mit Algorithmus 2.2*
2. *Wiederhole:*
   solange $\exists b \in N(a) : \gamma(b) < \gamma(a)$
   b^* sei so ein $b \in N(a)$ mit $\gamma(b^*) < \gamma(a)$
   Setze $a \leftarrow b^*$
3. *Setze $(s_{i,j}) \leftarrow a$*

Je nach Definition der Nachbarschaftsfunktion N kann die Menge $N(a)$ allerdings sehr groß werden und es kann äußerst aufwändig werden, alle Lösungen dieser Menge auszuwerten und zu vergleichen. Deshalb gibt es lokale Suchalgorithmen die nur eine bestimmte Teilmenge von $N(a)$ betrachten oder die Suche früher abbrechen weil sie schon eine akzeptable Verbesserung gefunden haben.

Ein naheliegendes Verfahren ist die sogenannte *Best Improvement Local Search* Methode. Gibt es in einer Nachbarschaft mehrere bessere Lösungen, so möchte man nicht irgendeine davon wählen, sondern jene mit dem niedrigsten Zielfunktionswert. Man möchte also in jedem Iteratiosschritt die aktuelle Lösung zum lokalen Optimum der Nachbarschaft modifizieren. Dies hat zwar einerseits den Nachteil, dass man jede einzelne zulässige Lösung überprüfen und vergleichen muss, andererseits aber auch den Vorteil von großen Verbesserungssprüngen [vgl. 16]. Formal kann man dieses Verfahren wie folgt definieren.

Algorithmus 2.4 (Best Improvement Local Search)

Eingabe: *Matrix $(p_{i,j})$ der Bearbeitungszeiten und die Matrix $(R_{i,j})$ der Maschinenreihenfolgen.*

Ausgabe: *Matrix $(s_{i,j})$ der Startzeiten*

1. *Generiere eine Startlösung a mit Algorithmus 2.2*
2. *Wiederhole:*
   solange $\exists b \in N(a) : \gamma(b) < \gamma(a)$
   wähle $b^* \in N(a) : \gamma(b^*) = min_{b \in N(a)} \gamma(b)$
   und setze $a \leftarrow b^*$

3. *Setze* $(s_{i,j}) \leftarrow a$

Da für Probleme, bei denen die Nachbarschaftsmengen sehr groß werden das Best Improvement Verfahren durch den Rechenaufwand sehr langsam werden kann, gibt es ein abgeändertes Verfahren, welches sich bereits mit einer einfachen Verbesserung der Lösung zufrieden gibt. Es wird nicht unbedingt das lokale Optimum gesucht. Dies bringt zwar Einbußen bei der Größe der Verbesserungsschritte, kann aber enorme Vorteile beim Rechenaufwand und somit bei der Rechenzeit bringen.

Algorithmus 2.5 (First Improvement Local Search)

Eingabe: *Matrix* $(p_{i,j})$ *der Bearbeitungszeiten und die Matrix* $(R_{i,j})$ *der Maschinenreihenfolgen.*

Ausgabe: *Matrix* $(s_{i,j})$ *der Startzeiten*

1. *Generiere eine Startlösung* a *mit Algorithmus 2.2*
2. $T = \emptyset$
3. *Solange* $\exists b \in N(a) \setminus T$ *wiederhole:*
   wähle $b \in N(a) \setminus T$
   setze $T = T \cup \{b\}$
   wenn $\gamma(b) < \gamma(a)$*:*
   setze $a \leftarrow b$ *und setze* $T \leftarrow \emptyset$
4. *Setze* $(s_{i,j}) \leftarrow a$

Ein schwerwiegender Nachteil der lokalen Suche ist die Anfälligkeit in lokalen Optima hängen zu bleiben und globale Optima zu übersehen. Es kann nämlich leicht passieren, dass sich nach einer bestimmten Anzahl an Iterationen immer wieder das selbe Muster an gewählten Nachbarschaften und lokalen Optima bildet. In solch einem Fall wird es unmöglich, noch zum Gesamtoptimum zu gelangen [vgl. 16].

2.4.3 Tabusuche

Eine Gruppe von lokalen Suchalgorithmen, die versucht, genau dieses Problem zu beseitigen, nennt sich *Tabusuche* und wird in diesem Abschnitt vorgestellt. Die Tabusuche ist ein lokales Suchverfahren, welches versucht, kürzlich getätigte Modifikation zu vermeiden, um nicht in lokalen Optima hängen zu bleiben. Dies passiert, indem eine Liste mit den letzten Modifikationen ständig mitgeführt und aktualisiert wird — die sogenannte Tabu Liste. Meist ist diese Liste kurz.

Im Folgenden wird ein allgemeiner Tabusuche Algorithmus definiert.

Algorithmus 2.6 (Tabusuche)

Eingabe: *Matrix* $(p_{i,j})$ *der Bearbeitungszeiten, die Matrix* $(R_{i,j})$ *der Maschinenreihenfolgen, Länge der Tabuliste* t*, Abbruchwert* q *und maximale Iterationsanzahl* maxiter

Ausgabe: *Matrix* $(s_{i,j})$ *der Startzeiten*

1. *Generiere eine Startlösung a mit Algorithmus 2.2*
2. *Setze Lösung* $\leftarrow a$
   Setze besteLösung $\leftarrow$ *Lösung*
   Erstelle TabuListe $= \emptyset$
3. *Wiederhole Schritte 4-7 bis* $\gamma(besteLösung) < q$ *oder* maxiter *erreicht:*
4. *Setze* $M \leftarrow N(Lösung) \setminus TabuListe$
   Wenn $M \neq \emptyset$*:*
   wähle $b^* \in M : \gamma(b^*) = \min_{b \in M} \gamma(b)$
5. *Setze* $TabuListe = TabuListe \cup \{b^*\}$
   Wenn Länge($TabuListe$*)* $> t$*, entferne ältesten Eintrag, also*
   $TabuListe = TabuListe \setminus \{TabuListe(t)\}$
6. *Wenn* $\gamma(b^*) < \gamma(besteLösung)$*:*
   setze besteLösung $\leftarrow b^*$
7. *Setze Lösung* $\leftarrow b^*$
8. *Gib* $(s_{i,j}) \leftarrow$ *besteLösung zurück*

In Schritt 3 erkennt man Teile des *Best Improvement Local Search* Algorithmus wieder. Allerdings wird eben hier nicht in der gesamten Nachbarschaft der aktuellen Lösung gesucht, sondern nur unter jenen, die nicht vor kurzem gewählt wurden und somit auf der *TabuListe* stehen.

Es seien beispielsweise l_1, l_2, l_3 zulässige Lösungen eines Scheduling Problems P und ein Lokaler Suchalgorithmus bleibe in der Schleife $l_1 \rightarrow l_2 \rightarrow l_3 \rightarrow l_1$ hängen.

Ein Verfahren der Tabusuche mit einer Tabuliste der Länge 5 würde nach der 3. Iteration die Lösungen l_1, l_2 und l_3 in der Tabuliste gespeichert haben und somit nach l_3 nicht zurück zu l_1 gelangen können, da dies unzulässig wäre. Somit wäre der Kreisschluss verhindert und es könnte weiter in Richtung des globalen Optimums gesucht werden [vgl. 13].

2.4.4 Simulated Annealing

Eine letzte Art der hier vorgestellten lokalen Suche nennt sich *Simulated Annealing* oder *simulierte Abkühlung*. Der Name stammt aus der Natur und lehnt sich an den Prozess des Gefrieren einer Flüssigkeit oder der Abkühlung von Metallen an. Die Idee dahinter ist, dass zu Beginn des Lösungsverfahren die imaginäre Temperatur der Lösung noch hoch ist und man schnell in einem lokalen Optimum hängen bleiben kann. Je länger der Algorithmus allerdings sucht, desto unwahrscheinlicher wird es, sich gerade bei einem rein lokalen Optimum zu befinden — man sagt die Temperatur nimmt ab beziehungsweise es kühlt

ab. Während die Temperatur des Verfahrens noch hoch ist, soll der Algorithmus auch teilweise schlechtere Lösungen akzeptieren um aus lokalen Minima herauszukommen. Dieses Akzeptieren von qualitativ schlechteren Lösungen soll durch eine Zufallsstruktur bestimmt werden. Je weiter das Verfahren allerdings abkühlt, desto seltener sollen schlechtere Ergebnisse akzeptiert werden, um sich nicht vom tatsächlichen globalen Optimum wieder zu entfernen.

Eine Basisvariante der Simulierten Abkühlung sieht wie folgt aus.

Algorithmus 2.7 (Simulated Annealing)

Eingabe: *Matrix $(p_{i,j})$ der Bearbeitungszeiten, die Matrix $(R_{i,j})$ der Maschinenreihenfolgen, Abbruchwert q und maximale Iterationsanzahl* maxiter

Ausgabe: *Matrix $(s_{i,j})$ der Startzeiten*

1. *Generiere eine Startlösung a_0 mit Algorithmus 2.2*
2. *setze $a \leftarrow a_0$ und $a_{best} \leftarrow a$, setze $t = 0$ und $i = 1$, bestimmte eine monoton fallende Temperaturfolge T_t und eine Temperaturschrittweite N_t.*
3. *Wiederhole Schritte 4-7 bis $\gamma(a_{best}) < q$ oder* maxiter *erreicht:*
4. *Wähle zufällig eine Lösung b aus der Nachbarschaft $N(a)$.*
5. *Wenn $\gamma(b) < \gamma(a)$:*
   setze $a \leftarrow b$
   Andernfalls:
   mit Wahrscheinlichkeit $\exp\left(-\frac{\gamma(b)-\gamma(a)}{T_t}\right)$ *setze dennoch $a \leftarrow b$*
6. *Wenn $\gamma(a) < \gamma(a_{best})$:*
   setze $a_{best} \leftarrow a$
7. *Wenn $i < N_t$:*
   setze $i = i + 1$
   Sonst:
   setze $t = t + 1$ und $i = 1$
8. *Gib $(s_{i,j}) \leftarrow a_{best}$ zurück*

In Schritt 5 geschieht der wesentliche Teil des Algorithmus. Obwohl der gewählte Nachbar einen schlechteren Zielfunktionswert liefert, kann es sein, dass er akzeptiert wird und von dieser Lösung aus weiter gesucht wird. Die Wahrscheinlichkeit $\exp\left(-\frac{\gamma(b)-\gamma(a)}{T_t}\right)$ wird kleiner, je schlechter der Zielfunktionswert von b im Vergleich zur aktuellen Lösung a ist. Darüber hinaus wird die Wahrscheinlichkeit, dass die schlechtere Lösung akzeptiert wird mit fortlaufendem Algorithmus und somit sinkender Temperatur immer geringer. Die Wahrscheinlichkeit geht mit $T_t \to 0$ gegen 0 und wird somit immer mehr zu einem gewöhnlichen lokalen Suchalgorithmus, der nur bessere Lösungen akzeptiert und von diesen weiter sucht.

Simulated Annealing kombiniert die Vorteile der einfachen Lokalen Suche mit den Vorteilen der Tabusuche und ist eine sehr empfehlenswerte Methode um Job-Shop Scheduling Probleme zu lösen [vgl. 17].

Kapitel 3

Industrielle Aufgabenstellungen

In den bisherigen Kapiteln war das betrachtete Problem für Zwecke der Anschaulichkeit und zum leichteren Verständnis eher klein und übersichtlich gehalten. Leider ist dies in der Realität selten der Fall und Problemstellungen aus der Industrie enthalten eine Vielzahl an zusätzlichen Anforderungen und Nebenbedingungen.

3.1 Ressourcetypen

In den bisher behandelten Kapiteln handelte es sich bei den Ressourcen ausschließlich um Maschinenressourcen, die zu der Gruppe der erneuerbaren Ressourcen gehören. Sie stehen während jeder Zeitperiode mit ihrer vollen Kapazität zur Verfügung und können nur belegt, nicht aber verbraucht werden. Ist eine Maschine also im Zeitintervall T belegt, so steht sie im Intervall $T + 1$ wieder voll zur Verfügung.

In vielen realen Problemstellung kommen allerdings auch Ressourcen vor, die sich nicht wie Maschinenkapazitäten verhalten [vgl. 16]. Alternative Ressourcentypen sind:

- **Nicht-erneuerbare Ressourcen:** wenn eine Ressource über den gesamten Planungszeitraum nur einmal mit einer bestimmten Kapazität zur Verfügung steht nennt man sie nicht-erneuerbar. Dies könnte beispielsweise ein Rohstoff sein, der nur begrenzt zur Verfügung steht und der nach dem Verbrauch nicht mehr genutzt werden kann. Ebenfalls typisch für eine nicht-erneuerbare Ressource in einem Planungsproblem ist ein vorgegebenes Budget.
- **Teilweise-erneuerbare Ressourcen:** sind eine Verallgemeinerung von erneuerbaren und nicht-erneuerbaren Ressourcen. Es kann ein bestimmtes Intervall bestimmt werden, in dem die Ressource jeweils wieder zur

Verfügung steht. Erneuerbare Ressourcen sind ein Spezialfall der teilweise-erneuerbaren Ressourcen bei dem die Länge des Erneuerungs-Intervalls genau einen Zeitschritt beträgt. Nicht-erneuerbare Ressourcen sind ebenfalls ein Spezialfall, bei dem das Intervall die Länge der gesamten Planungsperiode besitzt.

- **Kumulative Ressourcen:** sind ähnlich definiert wie die teilweise-erneuerbaren Ressourcen, die nach einem bestimmten Intervall wieder zur Verfügung stehen. Der Unterschied bei kumulativen Ressourcen ist, dass nicht-verbrauchte Kapazitäten in das nächste Zeitintervall transferiert werden können. Wird also in einer Periode weniger verbraucht als zur Verfügung steht, kann in der nächsten Periode um diese Menge mehr verbraucht werden. Es können allerdings nicht im Vorhinein Kapazitäten aus dem nächsten Intervall konsumiert werden.

- **Doppelt beschränkte Ressourcen:** sind sowohl über die gesamte Planungsperiode, als auch über kleine Teilintervalle beschränkt.

3.2 Rüstzeiten

Bisher wurden Scheduling Probleme betrachtet, bei denen Aufträge auf Maschinen abgearbeitet wurden, ohne dass dazwischen Wartezeiten oder sonstige Puffer entstanden. In der Praxis kann es notwendig sein, dass bei verschiedenen Aufträgen Maschinen gereinigt, neu kalibriert oder umgerüstet werden müssen [vgl. 7]. Man unterscheidet zwischen:

- Jobabhängigen Rüstzeiten
- Maschinenabhängigen Rüstzeiten
- Maschinen- und Jobabhängigen Rüstzeiten
- Unabhängigen Rüstzeiten

Die jeweilige Dauer dieser Vorgänge ist nicht selten auch von der Reihenfolge abhängig, in der die einzelnen Jobs abgearbeitet werden. Dabei gilt es wieder zu unterscheiden:

- Ein-Vorgänger abhängig, also ob es nur relevant ist, welcher Auftrag unmittelbar davor durchgeführt wurde
- Mehr-Vorgänger abhängig, also ob auch die Aufträge davor eine Rolle spielen

Eine Rüstzeit wird symmetrisch genannt, wenn sie bei Vertauschen der Reihenfolge der Aufträge gleich bleibt. Ansonsten heißt sie asymmetrisch. Asymmetrische Rüstzeiten kommen beispielsweise in der Ölindustrie vor, bei der Reinigung von Transportwagons. Wird ein höherwertiges Produkt, wie zum Beispiel

Diesel, nach einem niederwertigen Produkt wie Heizöl transportiert, fällt eine Zeit zur Reinigung als Rüstzeit an. Bei umgekehrter Reihenfolge beträgt diese Rüstzeit allerdings 0, da keine Reinigung notwendig ist.

Ein Beispiel für eine symmetrische Rüstzeit ist die Textilindustrie, bei der das Umstecken der Garne für zwei Farben die gleiche Rüstzeit in Anspruch nimmt, unabhängig davon, welche Farbe zuerst gesponnen wurde [vgl. 7].

Implementiert können diese Rüstzeiten durch Einführung von m Rüstzeitmatrizen mit Dimension $(n+1) \times n$ werden. Dabei ist m die Anzahl der Maschinen und n die Anzahl der Jobs. Der Eintrag (i, j) einer Rüstzeitmatrix zu Maschine k gibt dann an, wie Lange mit der Bearbeitung von Job j nach der Bearbeitung von Job i auf Maschine k gewartet werden muss. Es wird eine Zeile mehr benötigt, um auch die Rüstzeit für die Bearbeitung des ersten Jobs zu modellieren, der noch keinen Vorgänger hat.

3.3 Multiprozessor Aufgaben

P sei ein Scheduling-Problem mit den Jobs $J = \{1, \ldots, n\}$ und den Operationen O_j für Job j. Bisher wurde die Annahme getroffen, dass für jede Operation $o_{i,j} \in O_j$ genau eine Ressource (Maschine) benötigt wird. Dies trifft zwar für Probleme in der Maschinenbelegungungsplanung zu, in etlichen anderen Ablaufplanungsproblemen ist diese Annahme allerdings zu speziell. Wenn man Ressourcen nämlich nicht nur auf Maschinen beschränkt, sondern auch Personal, Hilfsgeräte und Arbeitsplätze als solche modelliert kann es durchaus vorkommen, dass für eine Operation $o_{i,j} \in O_j$ zwei oder noch mehr Ressourcen benötigt werden.

Am Beispiel eines Verladebahnhofs von Zugwagons werden beispielsweise für die Operation Zug-Beladen die Ressourcen Beladungsteam, Bahngleise und Beladestation benötigt. Alle drei müssen während der Operation zur Verfügung stehen [vgl. 4].

Fazit

Scheduling-Probleme treten in unserer Welt in den verschiedensten Formen auf und sind in den meisten Fällen schwer zu lösen. Eine der interessantesten Varianten dieser Probleme ist das Job-Shop-Ablaufplanungsproblem bei dem Jobs in einer bestimmten Reihenfolge auf Ressourcen (Maschinen) abgearbeitet werden müssen. Da diese Art des Problems nicht-polynomiell lösbar ist, werden anstatt exakten Verfahren Heuristiken zur Lösung verwendet.

Ein häufig eingesetztes Verfahren zur Lösung von Job-Shop-Scheduling Problemen ist die *Shifting-Bottleneck Heuristik*. Dabei wird schrittweise eine Engpassressource gewählt und darauf ein einfacheres 1-Maschinenhilfsproblem exakt gelöst. Für ein Problem mit m Ressourcen hat die Shifting-Bottleneck Heuristik nach m Iterationen alle Jobs eingeplant und erreicht in der Regel eine sehr gute Näherung an das Optimum.

Die Basisvariante dieses Algorithmus ist nur für relativ einfach gestaltete Scheduling-Probleme ausgelegt und müsste um die in Kapitel 5 vorgestellten Anforderungen erweitert werden, um für industrielle Aufgabenstellungen angewandt werden zu können.

Sehr etabliert sind auf diesem Gebiet *Local Search Verfahren*, die ausgehend von einer Startlösung durch schrittweise Modifikation zu einer besseren Lösung gelangen, die zwar im Allgemeinen nicht das Optimum ist, doch in der Regel eine sehr gute Näherung dazu ist.

Literaturverzeichnis

[1] Adams, J., Balas, E. und Zawack, D. *The shifitng bottleneck procedure for job shop scheduling.* Management Science, 34(3): 391-401, 1988.

[2] Braune, Roland und Wagner, Stefan et al. *Optimization Methods for Large-scale Production Scheduling Problems.* Upper Austria University of Applied Sciences

[3] Bräsel, Heidemarie und Frank, Werner. *Einführung in die Scheduling-Theorie.* Otto-von-Guericke Universität Magdeburg.

[4] Brucker, Peter. *Scheduling Algorithms.* Springer Verlag, Berlin, Heidelberg, New York, 2003.

[5] Habrock, Julian. *Einführung in Production Scheduling - Übersicht über Scheduling-Probleme sowie Lösungs- und Optimierungsmethoden.* Seminararbeit Fachhochschule Aachen, 2012.

[6] Henning, Andre. *Dissertation: Praktische Job-Shop Scheduling-Probleme.* Friedrich Schiller Universität Jena, 2002.

[7] Heuer, Jörg. *Das Multiprozessor Scheduling-Problem mit Reihenfolge-abhängigen Rüstzeiten.* Deutscher Universitẗs Verlag, Wiesbaden, 2004.

[8] Hübner, Felix und Schellenbaum, Uli et. al. *Evaluation von Schedulingproblemen für die Projektplanung von Großprojekten am Beispiel des kerntechnischen Rückbaus.* Working Paper Series in Production and Energy, Karlsruher Institut für Technologie, 2017.

[9] Ivens, Philip und Lambrecht, Marc. *Extending the shifting bottleneck procedure to real-life applications.* European Journal of Operational Research 90, 252-268, 1996.

[10] Jaehn, Florian und Pesch, Erwin. *Ablaufplanung - Einführung in Scheduling.* Springer Gabler Verlag, Berlin-Heidelberg, 2014.

[11] Liu, Shi Qiang und Kozan, Erhan. *A hybrid shifting bottleneck procedure algorithm for the parallel-machine job-shop scheduling problem.* School of Mathematical Sciences, Queensland University of Technology, Brisbane, 2012.

[12] Mönch, Lars und Rose, Oliver. *Shifting-Bottleneck-Heuristik für komplexe Produktionssysteme: softwaretechnische Realisierung und Leistungsbewertung.* Institut für Wirtschaftsinformatik, Technische Universität Ilmenau

[13] Schmidt, Keith. *Using Tabu Search to Solve the Job Shop Scheduling Problem with Sequence Dependent Setup Times.* 2001.

[14] Schuster, Christoph Jörg. *Dissertation: No-wait Job-Shop-Scheduling: Komplexität und Local Search.* Universität Duisburg-Essen, 2003.

[15] Siedentopf, Jukka. *Anwendung und Beurteilung heuristischer Verbesserungsverfahren für die Maschinenbelegungsplanung.* Universität Leipzig, 1994.

[16] Siedentopf, Jukka. *Job Shop Scheduling - Planung durch probabilistische lokale Suchverfahren.* Deutscher Universitätsverlag, Wiesbaden, 2002.

[17] Yazdani, M. und Gholami, M. et.al. *A Simulated Annealing Algorithm for Flexible Job-Shop Scheduling Problem.* Journal of Applied Science, Volume 9(4): 662 - 670, 2009.