# Inhaltsverzeichnis

# 1 Einleitung

## 1.1 Problembeschreibung

Bei synchronen Flow-Shop-Problemen handelt es sich um Produktionsplanungsprobleme, bei denen die zu produzierenden Güter (Jobs) auf einer zyklisch angeordneten Produktionsanlage produziert werden. Die Produktionsanlage besteht aus $m$ Stationen, die sich mit der Anlage drehen. Außen, um die Anlage herum, befinden sich $m$ fortlaufend nummerierte fixierte Maschinen $M_1, \ldots, M_m$, die die einzelnen Produktionsschritte durchführen. Dabei handelt es sich bei Maschine $M_1$ um das Einlegen des Jobs in die Anlage und bei Maschine $M_m$ um die Entnahme des fertigen Produkts. Durch Rotation der Anlage werden die Stationen mit den auf ihnen befindlichen Jobs zur jeweils nächsten Maschine transportiert. Die Reihenfolge, in der alle Jobs die Maschinen durchlaufen müssen ist also fest vorgegeben. Eine Rotation darf immer nur dann stattfinden, wenn alle Maschinen ihren Produktionsschritt an ihrem aktuellen Job durchgeführt haben. Auf diese Weise können die Jobs, im Gegensatz zum klassischen (asynchronen) Flow-Shop, immer nur *synchron* zur nachfolgenden Maschine gelangen, Die Zeit, die zwischen zwei Rotationen vergeht, wird als *Zykluszeit* bezeichnet.

Die zu produzierenden Jobs sind gegeben durch die Menge $J = \{j_1, \ldots, j_n\}$ und die Prozesszeiten von Job $j$ auf Maschine $i$ sind durch $p_{ij}$ gegeben. Ziel ist es, eine Reihenfolge $\pi \in J^n$ der Jobs zu erstellen, die die gesamte Produktionsdauer minimiert. Die Zykluszeiten $c_t$ mit $1 \leq t \leq n + m - 1$ berechnen sich wie folgt:

$$c_t = \max_{i=\max\{1,t-n+1\}}^{\min\{t,m\}} p_{i\pi_{t-i+1}}$$

Die Zielfunktion ist also $C_{\max} = \sum_{t=1}^{n+m-1} c_t$. Andere Zielfunktionen werden in dieser Masterarbeit nicht betrachtet.

Eine Teilmenge $D \subseteq \{M_1, \ldots, M_m\}$ der Maschinen heißt *dominierend*, wenn

$$p_{dj} \geq p_{ej} \quad \forall j \in J, d \in D, e \notin D$$

ist. Die Prozesszeiten aller Jobs auf dominierenden Maschinen sind also immer mindestens so groß wie die Prozesszeiten auf den restlichen Maschinen. Treten dominierende Maschinen auf, müssen für die Berechnung der Zykluszeiten die Prozesszeiten auf den übrigen Maschinen also nicht betrachtet werden.

Zusätzlich benötigen die Jobs Ressourcen aus einer Menge $R$, um in die Stationen eingelegt werden zu können. Diese Ressourcen können erst nach Fertigstellung eines Jobs, also nachdem er an Maschine $M_m$ aus der Anlage genommen wurde, wiederverwendet werden. Sie sind allerdings nur in begrenzter Zahl vorhanden und im Allgemeinen ist nicht jede Ressource für jeden Job geeignet. Für $j \in J$ sei $\rho_j \subseteq R$ die Menge der Ressourcen, die für $j$ geeignet ist. Umgekehrt sei für $r \in R$ mit $\iota_r \subseteq J$ die Menge der Jobs bezeichnet, für die $r$ geeignet ist. An Maschine $M_1$ kann es daher notwendig sein, vor dem Einlegen des nächsten Jobs die Ressource zu wechseln, wenn auf der entsprechenden Station zuvor Job $j \in J$ mit Ressource $r \in \rho_j$ fertiggestellt wurde und nun Job $j' \in J$ eingelegt werden soll, wobei $r \notin \rho_{j'}$ ist.

Für die Ressourcen können folgende Situationen auftreten:

- Alle Ressourcen sind für alle Jobs geeignet, also $\rho_j = R$ für alle $j \in J$.
- Die Jobs lassen sich in disjunkte Gruppen unterteilen, so dass für alle Jobs aus einer Gruppe dieselbe Ressourcenmenge geeignet ist. Wenn also $\rho_i \cap \rho_j \neq \emptyset$, dann folgt $\rho_i = \rho_j$.
- Die Ressourcenmengen bilden Hierarchien. D.h., wenn $\iota_q \cap \iota_r \neq \emptyset$, dann folgt $\iota_q \subseteq \iota_r$ oder $\iota_r \subseteq \iota_q$.
- Die $\rho_j$ sind beliebige Teilmengen von $R$.

Neben dem Wechsel von Ressourcen, der zusätzliche Zeit in Anspruch nimmt, können auch andere Formen von *Rüstkosten* auftreten. Z.B. kann es sein, dass an einer Station zunächst einige Umstellungen vorgenommen werden müssen, bevor der neue Job eingelegt werden kann. Die Jobs können in Familien $\mathcal{F}$ eingeteilt werden, so dass beim Übergang zwischen zwei Jobs aus den Familien $f$ und $g$ die Rüstkosten $s_{fg}$ auftreten. Diese Rüstkosten können

- sowohl vom Vorgänger als auch vom Nachfolger abhängig sein ($s_{fg}$),
- nur vom Nachfolger abhängig sein ($s_{fg} = s_g$) oder
- konstant sein ($s_{fg} = s > 0$).

Dabei wird jeweils für $s_{ff} = 0$ angenommen, dass also keine Rüstkosten innerhalb einer Familie auftreten.

Insgesamt gilt es also, neben der Reihenfolge $\pi$ auch ein Mapping $f : J \to R$ zu finden, das jedem Job $j \in J$ eine Ressource $r \in \rho_j$ zuweist und folgenden Ansprüchen genügt:

- $f$ muss zulässig sein in dem Sinne, dass beim Einlegen jedes Jobs $j \in J$ eine Ressource $r \in \rho_j$ verfügbar ist (d.h., dass $r$ sich nicht gerade an anderer Stelle in

der Anlage befindet).

- $f$ und $\pi$ zusammen sollen optimal sein in dem Sinne, dass die Summe aus den durch $\pi$ und $f$ definierten Zykluszeiten und Rüstkosten minimal ist.

## 1.2 Motivation

In [**?**] wurde gezeigt, dass dieses Problem schon $\mathcal{NP}$-schwer ist, wenn alle Ressourcen für alle Jobs geeignet sind und keine Rüstkosten auftreten. Versuche, dieses Problem – oder auch einige Spezialfälle davon – mit linearer Programmierung zu lösen, waren nur für sehr kleine Instanzen mit $n < 30$ erfolgreich, was weit hinter praktischen Anforderungen zurückliegt. Aufgrund der Komplexität des Problems sollen in dieser Arbeit zwei Dekompositionsansätze verfolgt werden:

1. Zunächst wird eine Reihenfolge $\pi$ aufgestellt, ohne Ressourcen und Rüstkosten zu betrachten, und anschließend wird das Mapping $f$ basierend auf $\pi$ erstellt, ohne $\pi$ noch zu verändern.

2. Es wird zuerst das Mapping $f$ erstellt, so dass die Ressourcen zulässig zugewiesen sind und die Rüstkosten minimal sind. Anschließend werden, ohne $f$ zu verändern, die Jobs so angeordnet, dass die Zykluszeiten möglichst minimal sind, und so $\pi$ erstellt.

Beide Ansätze liefern natürlich im Allgemeinen keine optimalen Lösungen. Außerdem sind selbst die aus den Ansätzen resultierenden Teilprobleme teilweise noch $\mathcal{NP}$-schwer. Beispielsweise ist bei Ansatz (1) das Berechnen einer optimalen Reihenfolge $\pi$ (soweit bekannt) nur in dem Spezialfall, dass es genau zwei benachbarte dominierende Maschinen gibt, mit dem Algorithmus von Gilmore und Gomory [**?**] in Polynomialzeit lösbar. Bei der anschließenden Zuweisung von Ressourcen ist noch unbekannt, ob ein polynomieller Algorithmus existiert. Dies herauszufinden ist eines der Ziele dieser Arbeit.

Da die Rüstkosten $s_{fg}$ in dem gegebenen Praxisfall deutlich größer sind als die Prozesszeiten $p_{ij}$ der Jobs auf den Maschinen, ist davon auszugehen, dass Ansatz (2) für diesen Praxisfall die besseren Lösungen erzielen wird. Nichtsdestotrotz ist auch der erste Ansatz interessant, da er in anderen Praxisbeispielen von Bedeutung sein kann.

## 1.3 Bisherige Ansätze

Hier das ursprüngliche „allumfassende“ MIP vorstellen und seine schlechte Laufzeit als Motivation für die Dekompositionsansätze nehmen.

# 2 Der erste Dekompositionsansatz

Beim ersten Dekompositionsansatz wird zunächst eine (möglichst optimale) Jobreihenfolge $\pi$ bestimmt. Dabei werden Ressourcen und Rüstkosten außer Acht gelassen. Anschließend wird das Mapping $f$ aufgestellt, so dass Ressourcen nur dann eingeplant werden, wenn sie auch zur Verfügung stehen, und darüber hinaus möglichst selten ausgetauscht werden müssen, so dass geringe Rüstkosten auftreten.

In Abschnitt 2.1 werden einige exakte und heuristische Verfahren für die Berechnung von $\pi$ vorgestellt, was im Allgemeinen $\mathcal{NP}$-schwer ist. Anschließend werden in Abschnitt 2.2 Verfahren vorgestellt, die die ein möglichst gutes Mapping $f$ erzeugen. Dabei wird speziell darauf eingegangen, ob mit der gegebenen Reihenfolge $\pi$ und den gegebenen Ressourcen überhaupt eine zulässige Lösung möglich ist, und darauf, wie dann ggf. die Zulässigkeit durch nachträgliche Änderungen an $\pi$ erzeugt werden kann.

## 2.1 Berechnen einer Jobreihenfolge

### 2.1.1 Gilmore Gomory

Der Algorithmus von Gilmore und Gomory [**?**] löst in $\mathcal{O}(n \log n)$ einen speziellen Fall des Travelling-Salesman-Problems (TSP), bei dem alle Knoten zwei Koordinaten $x, y$ haben und die Kantenkosten $c_{ij}$ zwischen je zwei Knoten $i$ und $j$ nur von der $x$-Koordinate von $i$ und der $y$-Koordinate von $j$ abhängig sind.

Diese Situation liegt beim synchronen Flow-Shop vor, wenn es nur zwei benachbarte dominierende Maschinen gibt. O.B.d.A seien dies $M_1$ und $M_2$. Die Berechnung der Zykluszeiten vereinfacht sich dann zu $c_t = \max\{p_{2\pi_{t-1}}, p_{1\pi_t}\}$ für $2 \leq t \leq n$. Sie sind also für je zwei Jobs $\pi_{t-1}, \pi_t$ nur noch von der Prozesszeit des vorderen Jobs auf der zweiten Maschine ($p_{2\pi_{t-1}}$) und der Prozesszeit des hinteren Jobs auf der ersten Maschine ($p_{1\pi_t}$) abhängig.

Mit diesem Spezialfall des Problems beschäftigt sich Matthias Kampmeyer in seiner Masterarbeit genauer. Daher wird hier nicht näher auf die Funktionsweise des Algorithmus eingegangen eingegangen.

### 2.1.2 Lineare Programmierung

Zum Finden einer optimalen Lösung dieses Teilproblems wurde folgendes Mixed Integer Program (MIP) aufgestellt. Dabei gilt für die Binärvariablen $x_{jk} = 1$ genau dann, wenn Job $j$ an Position $k$ in der Reihenfolge $\pi$ steht. Es ist außerdem $N = \{1, \ldots, n\}$ und $T = \{1, \ldots, n + m - 1\}$.

$$\min \quad \sum_{t=1}^{n+m-1} c_t \tag{2.1}$$

$$\text{s.t.} \quad \sum_{k=1}^{n} x_{jk} = 1 \qquad j \in N \tag{2.2}$$

$$\sum_{j=1}^{n} x_{jk} = 1 \qquad k \in N \tag{2.3}$$

$$c_t \geq \sum_{j=1}^{n} p_{t-k+1,j} \cdot x_{jk} \quad t \in Tk = \max\{1, t-m+1\}, \ldots, \min\{n, t\} \tag{2.4}$$

$$c_t \geq 0 \qquad t \in T \tag{2.5}$$

$$x_{jk} \in \{0, 1\} \qquad j, k \in N \tag{2.6}$$

Dieses MIP liefert für Instanzen mit $n \leq 30$ eine optimale Lösung in unter einer Stunde. Bei größeren Instanzen ist dieser Zeitaufwand zur Lösung nicht mehr praktikabel. Das gilt insbesondere für das gegebene Praxisbeispielen mit Instanzgrößen von mehreren Tausend Jobs.

### 2.1.3 Heuristische Verfahren

Aufgrund der $\mathcal{NP}$-Schwere des Problems und der schlechten Laufzeit des MIPs aus Abschnitt 2.1.2 bei Instanzen realer Größe werden hier einige heuristische Ansätze zur Lösung des Problems vorgestellt.

#### Non-Full-Schedule-Heuristik

Diese Heuristik ist eine konstruktive Greedy-Heuristik, die Schritt für Schritt einen Job an $\pi$ anhängt, beginnend mit einer leeren Reihenfolge. In jeder Iteration werden alle noch verbleibenden Jobs bewertet und der Job mit der besten Bewertung wird an $\pi$ angehängt. Die Heuristik benötigt also genau $n$ Iterationen. Die Bewertungsfunktion betrachtet die letzten $m - 1$ Zykluszeiten der noch nicht fertigen Reihenfolge, wobei die

Zykluszeiten am Anfang bei einer noch leeren Reihenfolge als 0 angenommen werden. Für jeden Job $j$, der noch nicht in $\pi$ ist, werden diese $m-1$ Zykluszeiten mit den ersten $m-1$ Prozesszeiten von $j$ verglichen. Die Idee ist, dass diese möglichst übereinstimmen sollten. Ist eine Prozesszeit sehr viel größer als die aktuelle Zykluszeit, zu der sie hinzugefügt werden würde, würde die Zykluszeit entsprechend ansteigen. Ist umgekehrt die Zykluszeit sehr viel größer als die Zugehörige Prozesszeit von $j$, dann würde diese kurze Prozesszeit verschenkt werden. Die Berwertungsfunktion berechnet jeweils den Unterschied zwischen Prozesszeit und Zykluszeit und bildet die Summe dieser $m-1$ Werte. Diese Summe ist die Bewertung für einen Job $j$. Der Job mit der kleinsten Bewertung wird an $\pi$ angehängt. Da in jeder Iteration alle Verbleibenden Jobs betrachtet werden und für jeden dieser Jobs $m-1$ Zeiten Verglichen werden, liegt die asymptotische Laufzeit dieser Heuristik in $\mathcal{O}(n^2m)$.

#### Double Ended Non-Full-Schedule-Heuristik

Diese Heuristik basiert auf der Non-Full-Schedule-Heuristik. Der Unterschied besteht darin, dass $\pi$ nicht nur von vorne, sondern gleichzeitig auch von hinten zur Mitte hin aufgebaut wird. Jeder noch nicht in $\pi$ enthaltende Job wird pro Iteration mit beiden Enden der bisherigen Reihenfolge verglichen. Die Bewertungsfunktion für das hintere Ende arbeitet analog. Es wird der beste Job für das vordere Ende und der beste für das hintere Ende gesucht und der mit der besseren Bewertung wird vorne bzw. hinten eingefügt. Die asymptotische Laufzeit liegt hier ebenfalls in $\mathcal{O}(n^2m)$.

#### Gilmore-Gomory-Heuristik

Diese Heuristik wendet den Algorithmus von Gilmore und Gomory auf beliebige Instanzen an. Dazu wird eine Instanz $I$ zunächst auf dominierende Maschinen untersucht. Nun können zwei Fälle eintreten:

1. Unter den dominierenden Maschinen $D$ gibt es zwei, die benachbart sind, also $\exists i \in \{1, \ldots, m\}, M, M' \in D$ mit $M = M_i$ und $M' = M_{i+1}$.

2. Es gibt keine benachbarten dominierenden Maschinen.

Wenn Fall (2) eintritt, werden alle $m$ Maschinen als dominierend angesehen, da dies für den Algorithmus keine Einschränkung darstellt. Nun werden zwei benachbarte dominie-

rende Maschinen gewählt. O.B.d.A seien dies die Maschinen $M_1$ und $M_2$. Seien

$$d_{\min} \coloneqq \min_{\substack{i \in \{M_1, M_2\} \\ j \in J}} p_{ij} \tag{2.7}$$

$$e_{\max} \coloneqq \max_{\substack{i \in D \setminus \{M_1, M_2\} \\ j \in J}} p_{ij} \tag{2.8}$$

$$K \coloneqq \frac{e_{\max}}{d_{\min}}. \tag{2.9}$$

Nun wird aus der gegebenen Instanz $I$ eine neue Instanz $I'$ erzeugt, die sich nur dadurch von $I$ unterscheidet, dass die Prozesszeiten auf allen dominierenden Maschinen außer auf $M_1$ und $M_2$ mit $\frac{1}{K}$ skaliert werden, also

$$p'_{ij} \coloneqq \begin{cases} \frac{p_{ij}}{K} & \text{für } i \in D \setminus \{M_1, M_2\} \\ p_{ij} & \text{sonst.} \end{cases} \tag{2.10}$$

Auf diese Weise sind $M_1$ und $M_2$ in $I'$ die einzigen dominierenden Maschinen und folglich kann $I'$ mit dem Algorithmus von Gilmore und Gomory optimal gelöst werden. Die resultierende Reihenfolge $\pi$ wird als heuristische Lösung für die ursprüngliche Instanz $I$ verwendet.

Da die Prozesszeiten der in $I'$ vernachlässigten dominierenden Maschinen sich um den Faktor $K$ von denen in $I$ unterscheiden, können sich die aus $\pi$ ergebenen Zykluszeiten von $I$ und $I'$ auch maximal um den Faktor $K$ unterscheiden:

$$c_t \leq K \cdot c'_t \quad \forall 1 \leq t \leq n + m - 1 \tag{2.11}$$

Da die optimale Lösung von $I'$ eine untere Schranke für die optimale Lösung von $I$ ist, also $C'_{\max} \leq C_{\max}$, folgt für die Lösung $L_{GG}$ der Gilmore-Gomory-Heuristik:

$$L_{GG} \leq K \cdot C'_{\max} \leq K \cdot C_{\max} \tag{2.12}$$

Die Gilmore-Gomory-Heuristik hat also eine relative Gütegarantie von $K$. Zusätzlich kann bei der Wahl der zwei benachbarten dominierenden Maschinen der Parameter $K$ optimiert werden, indem nicht beliebige Maschinen gewählt werden, sondern solche mit optimalen $d_{\min}$- bzw. $e_{\max}$-Werten.

Das finden der geeigneten dominierenden Maschinen kann in $\mathcal{O}(nm)$ durchgeführt werden. Das Anpassen der Prozesszeiten der übrigen dominierenden Maschinen ist nur in der Theorie von Interesse. Der Gilmore-Gomory-Algorithmus kann diese einfach ignorieren. Die Gesamtlaufzeit beträgt daher $\mathcal{O}(nm + n \log n)$.

### Nachbarschaftssuche

Mit einer simplen Nachbarschaftssuche können bereits existierende (nicht optimale) Reihenfolgen verbessert werden. Es wird in $\pi$ nach zwei Jobs gesucht, deren Vertauschung

den Zielfunktionswert verringert, und anschließend vertauscht. Dies geschieht so lange, bis kein solches Jobpaar mehr gefunden wird.

Das Suchen eines solchen Jobpaares kann in $\mathcal{O}(n^2)$ erfolgen. Da der Zielfunktionswert sich mit jeder Iteration verringert und er durch die optimale Lösung beschränkt ist, terminiert die Suche auf jeden Fall.

## 2.2 Zuweisen von Ressourcen

### 2.2.1 Zulässigkeit der Zuweisung

*Dieser Unterabschnitt muss auch nur noch geschrieben werden.*

Zulässigkeit mit Netzflussalgorithmus nachweisen. Darauf eingehen, dass bei disjunkten Jobgruppen keine Optimierungsmöglichkeiten bestehen. Im Gegensatz dazu s. 2.2.2.

### 2.2.2 Optimierung der Ressourcenzuweisung

*Soll dieser Abschnitt noch Bestandteil der Arbeit sein? Wenn ja, dann werde ich mich da zwischendurch immer mal wieder dransetzen, bis eine Lösung gefunden ist. Ein echter Zeitplan ist hierfür schwierig.*

Bei hierarchischen oder beliebigen Jobgruppen eine optimale Zuweisung mit möglichst wenig Rüstkosten finden. Dazu (falls möglich) einen polynomiellen Algorithmus angeben bzw. einen Beseis zur $\mathcal{NP}$-Vollständigkeit.

## 2.3 Unzulässige Reihenfolgen

### 2.3.1 Nachbarschaftssuche

*Anfang August hiermit anfangen.*

Nachbarschaftenssuchen formulieren, die eine unzulässige Reihenfolgen reparieren, so dass sie dann zulässig ist. Dabei soll die Reihenfolge möglichst wenig von ihrer Optimalität einbüßen.

### 2.3.2 Andere Ansätze

Platzhalter, falls noch andere Ideen aufkommen.

# 3 2. Dekompositionsansatz

## 3.1 Ressourcenzuweisung

*Dieser Abschnitt muss auch nur noch ausformuliert werden.*

Generell auf bekannte Laufzeitschranken und Verfahren für die Ressourcenzuweisung kurz eingehen. Davon speziell erklären, wie mit Binpacking eine minimale Anzahl an Rüstkosten erreicht werden kann für $s_{fg} = s$.

## 3.2 Zulässigkeit der Zuweisung

*Bis Mitte Juli sollte die Lösung hier erarbeitet sein.*

Folgende Fragen diskutieren und idealer Weise beweisen: Gibt es unzulässige Lösungen beim Binpacking? Wenn ja, wie lassen sie sich reparieren?

## 3.3 Anordnung der Jobgruppen

### 3.3.1 MIP mit fixierten Bins

Vorstellung des MIPs mit fixierten Bins mit $s_{fg} = s$. Vorstellung einiger Laufzeiten. *MIP steht. Evtl müssen noch einige Laufzeiten gemessen werden.*

Außerdem die erweiterte Formulierung auf allgemeine $s_{fg}$ und Diskussion. *MIP sollte bis Ende Juli formuliert und Laufzeiten gemessen sein.*

### 3.3.2 MIP mit freien Bins

*Muss nur noch aufgeschrieben werden. Evtl. noch ein paar Laufzeiten messen.*

Vorstellung des MIPs mit freien Bins. Die Laufzeit ist sehr viel schlechter, die primale Lösung ist aber meist schon nach kurzer Zeit besser als beim MIP mit fixierten Bins.

### 3.3.3 Mehrere fixierte Reihenfolgen

*Jetzt damit anfangen. Wahrscheinlich bis Mitte August.*

Basierend auf den Erkenntnissen aus 3.3.2 das MIP mit fixierten Bins für mehrere Binreihenfolgen laufen lassen. Um nicht sämtliche Binreihenfolgen durchprobieren zu müssen, ggf. mit bipartitem gewichteten Matching ein lokales Optimum zwischen zwei Bins suchen und daraus eine „gute“ Binreihenfolge erstellen.

# 4 Messergebnisse und Vergleiche

*Anfang September anfangen, soweit Messergebnisse vorliegen.*

Obwohl in den vorherigen Kapiteln schon einige Laufzeiten vorgestellt werden, hier nochmal eine Zusammenfassung und insbesondere ein Vergleich zwischen den beiden Dekompositionsansätzen. Sowohl echte Instanzen als auch generierte. Dabei ggf. auf Methoden zur Instanzengenerierung eingehen und diese bewerten?