Inhaltsverzeichnis

1 Einleitung

1.1 Problembeschreibung

Bei synchronen Flow-Shop-Problemen handelt es sich um Produktionsplanungsprobleme, bei denen die zu produzierenden Güter (Jobs) z.B. 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. In Abbildung 1.1 ist eine solche zyklische Anlage dargestellt. 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 M_i sind durch p_{ij} gegeben. Eine Beispielinstanz mit $n = 5$ und $m = 3$ ist in Tabelle 1.2 zu sehen. Ziel ist es, eine Permutation π der Jobs zu erstellen, die die gesamte Produktionsdauer minimiert. Diese Zielfunktion wird mit $C_{\max}$ bezeichnet. Die Beispielinstanz 1.2 ist in Abbildung **??** als Gantt-Diagramm aufgetragen. Oben sind die Jobs in der initialen Reihenfolge und unten in einer bezüglich $C_{\max}$ optimalen Reihenfolge. 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 lässt sich also durch $C_{\max} = \sum_{t=1}^{n+m-1} c_t$ berechnen. Andere Zielfunktionen werden in dieser Masterarbeit nicht betrachtet.

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

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

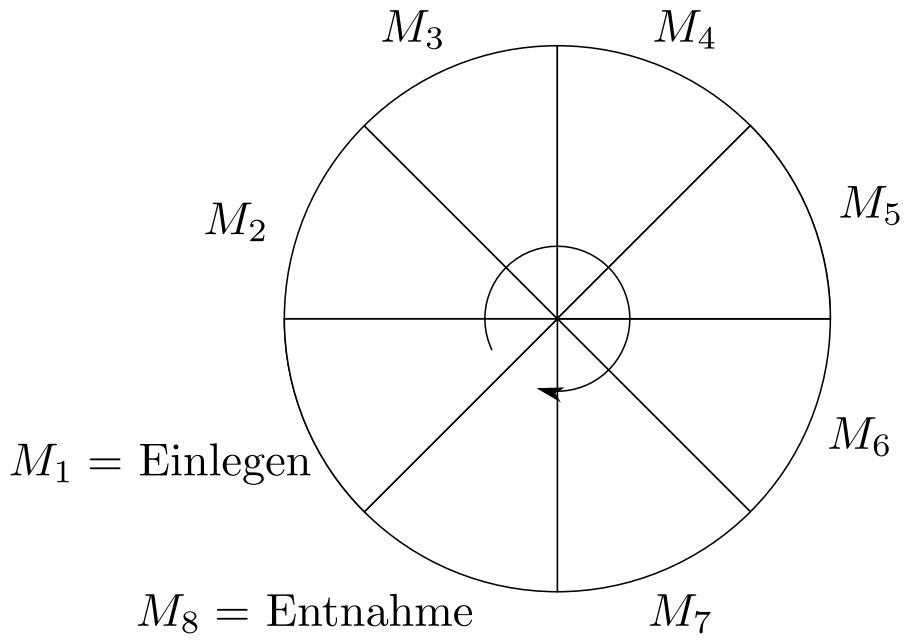

Abbildung 1.1: Kreisförmige Anlage mit $m = 8$ Maschinen.

	M_1	M_2	M_3
j_1	4	6	5
j_2	1	5	6
j_3	2	5	4
j_4	5	2	4
j_5	4	5	4

Abbildung 1.2: Beispielinstanz mit 5 Jobs und 3 Maschinen. Die Werte in der Tabelle sind die Prozesszeiten p_{ij}.

	t = 1	$t = 2$	$t = 3$	$t = 4$	t = 5	$t = 6$	t = 7
M_1	j_1	j_2	j_3	j_4	j_5		
M_2		j_1	j_2	j_3	j_4	j_5	
M_3			j_1	j_2	j_3	j_4	j_5

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 ρ_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 π 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 π zusammen sollen optimal sein in dem Sinne, dass die Summe aus den durch π 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 π aufgestellt, ohne Ressourcen und Rüstkosten zu betrachten, und anschließend wird das Mapping f basierend auf π erstellt, ohne π 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 π 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 π (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.

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

Als weitere Motivation für die Dekompositionsansätze dient folgendes Mixed Integer Linear Program (MIP):

$$\min \quad \sum_{t=1}^{n+m-1} c_t + \sum_{j=1}^{n}\sum_{h=1}^{n} s_{jh}y_{jh} \tag{1.1}$$

$$\text{s.t.} \quad \sum_{k=1}^{n} x_{jk} = 1 \qquad j \in N \tag{1.2}$$

$$\sum_{j=1}^{n} x_{jk} = 1 \qquad k \in N \tag{1.3}$$

$$c_t \geq \sum_{j=1}^{n} p_{t-k+1,j} \cdot x_{jk} \quad t \in T, k = \max\{1, t-m+1\}, \ldots, \min\{n, t\} \tag{1.4}$$

$$y_{jh} + 1 \geq x_{j,k-m} + x_{hk} \qquad j, h \in N, k = m+1, \ldots, n \tag{1.5}$$

$$c_t \geq 0 \qquad t \in T \tag{1.6}$$

$$x_{jk} \in \{0, 1\} \qquad j, k \in N \tag{1.7}$$

$$y_{jh} \in \{0, 1\} \qquad j, h \in N \tag{1.8}$$

N ist die Indexmenge der Jobs, also $N = \{1, \ldots, n\}$ und $T = \{1, \ldots, n+m-1\}$ ist die Indexmenge der Zykluszeiten. Die Binärvariablen x_{jk} geben an, an welcher Position k sich Job j befindet. Es gilt $x_{jk} = 1$ genau dann, wenn j an Position k ist. Die Nebenbedingungen 1.2 und 1.3 stellen sicher, dass sich jeder Job an genau einer Position befindet und dass sich an jeder Position genau ein Job befindet. In Nebenbedingung 1.4 werden die Zykluszeiten bestimmt. Den Binärvariablen y_{jh} wird in Nebenbedingung 1.5 folgende Bedeutung gegeben: $y_{jh} = 1$, wenn Job j genau m Positionen vor Job h in π liegt. In der Zielfunktion 1.1 wird dann die Summe aus allen Zykluszeiten gebildet und die Summe aller Rüstkosten s_{jh}, die beim Übergang von Job j zu h auftreten.

Obwohl in diesem MIP nicht die Verfügbarkeit von Ressourcen beachtet werden, benötigt es schon bei $n = 20$ mehrere Stunden zum Finden der optimalen Lösung. Es ist also für Instanzen mit mehreren Tausend Jobs nicht geeignet. Aufgrund dieser Tatsache ist eine heuristische Herangehensweise an synchrone Flow-Shop-Probleme mit Ressourcen und Rüstkosten eine notwendige Alternative.

2 Der erste Dekompositionsansatz

Beim ersten Dekompositionsansatz wird zunächst eine (möglichst optimale) Jobreihenfolge π bestimmt. Dabei werden Ressourcen und Rüstkosten außer Acht gelassen. Es wird also zunächst ausschließlich $C_{\max}$ optimiert. Anschließend wird ein 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 π vorgestellt, was im Allgemeinen $\mathcal{NP}$-schwer ist. Anschließend werden in Abschnitt 2.2 Verfahren vorgestellt, die ein möglichst gutes Mapping f erzeugen. Dabei wird speziell darauf eingegangen, ob mit der gegebenen Reihenfolge π und den gegebenen Ressourcen überhaupt eine zulässige Lösung möglich ist, und darauf, wie dann ggf. die Zulässigkeit durch nachträgliche Änderungen an π 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 gerichteten Travelling-Salesman-Problems (TSP), bei dem alle Knoten zwei Parameter x, y haben und die Kantenkosten c_{ij} zwischen je zwei Knoten i und j nur vom x-Wert von i und vom y-Wert 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. Jobs können durch Knoten repräsentiert werden und die beiden Prozesszeiten auf den dominierenden Maschinen liefern die Parameter x und y. Der Abstand zwischen zwei Knoten entspricht dann der Zykluszeit, die die entsprechenden Jobs verursachen, wenn sie nebeneinander liegen. Die Berechnung der Zykluszeiten vereinfacht sich hier 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 π_{t-1}, π_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.

Auf die Funktionsweise des Algorithmus soll in dieser Arbeit nicht näher eingegangen werden.

2.1.2 Lineare Programmierung

Zum Finden einer optimalen Lösung dieses Teilproblems wurde folgendes (MIP) aufgestellt. Es ist identisch mit dem MIP in Abschnitt 1.3 bis auf die Nichtberücksichtigung der Rüstkosten.

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

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

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

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

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

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

Dieses MIP liefert für Instanzen mit $n \leq 30$ eine optimale Lösung in unter einer Stunde. Diese Laufzeiten sind zwar schon deutlich besser als die Variante mit Rüstkosten in Abschnitt 1.3, bei größeren Instanzen ist dieser Zeitaufwand allerdings immer noch nicht praktikabel.

2.1.3 Heuristische Verfahren

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

Non-Full-Schedule-Heuristik

Diese Heuristik ist eine konstruktive Greedy-Heuristik, die Schritt für Schritt einen Job an π anhängt, beginnend mit einer leeren Reihenfolge. In jeder Iteration werden alle noch verbleibenden Jobs bewertet und der Job mit der besten Bewertung wird an π 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 π 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 um einen Wert c_+ ansteigen. Ist umgekehrt die Zykluszeit sehr viel größer als die zugehörige Prozesszeit von j, dann würde diese kurze Prozesszeit verschenkt werden. Die Differenz aus Zykluszeit und Prozesszeit wird mit c_- bezeichnet. Die Berwertungsfunktion berechnet für jeden Job die Summe aus den $m-1$ c_+-Werten. Diese Summe ist die Bewertung für einen Job j. Der Job mit der kleinsten Bewertung wird an π angehängt. Falls mehrere Jobs eine optimale Bewertung haben, wird für diese Jobs als zweites Kriterium die Summe der c_--Werte betrachtet. Der Job, bei dem diese Summe am kleinsten ist, verschenkt am wenigsten Zeit und wird ausgewählt. 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 π nicht nur von vorne, sondern gleichzeitig auch von hinten zur Mitte hin aufgebaut wird. Jeder noch nicht in π 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 π 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 π 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 $\mathcal{O}(nm + n \log n)$.

Für die weitere Analyse dieser Heuristik ist der Begriff der *Semidominanz* sinnvoll. Die Semidominanz ist ein Maß dafür, wie weit eine Teilmenge von Maschinen davon

entfernt ist, dominierend zu sein. Eine Semidominanz von 0 bedeutet, die Maschinen sind dominierend. Für eine Teilmenge von Maschinen $D \subseteq \{M_1, \ldots, M_m\}$ sei

$$d_{\min} = \min_{\substack{i:M_i \in D \\ j \in \{1,\ldots,n\}}} p_{ij}$$

die kleinste ihrer Prozesszeiten, analog zur obigen Definition von $d_{\min}$. Die Semidominanz ist dann definiert als

$$\mathcal{D}_D \coloneqq \sum_{i:M_i \notin D} \sum_{j=1}^{n} \max\{0, p_{ij} - d_{\min}\}.$$

Unabhängig von der relativen Gütegarantie K liefert diese Heuristik auch eine absolute Gütegarantie von $\mathcal{D}_{\{M_1,M_2\}}$, wenn M_1 und M_2 als benachbarte Maschinen ausgewählt werden.

Nachbarschaftssuche

Mit einer simplen Nachbarschaftssuche können bereits existierende (nicht optimale) Reihenfolgen verbessert werden. Mögliche Nachbarschaftsoperatoren sind

- xch_{ij}: vertauscht die Jobs an den Positionen i und j miteinander,
- $shift_{ij}$: „shiftet“ den Job an Position i nach Position j. Alle Jobs zwischen i und j rücken um eins nach links (falls $i < j$) bzw. nach rechts (falls $i > j$).

Auf Grundlage dieser Nachbarschaften lassen sich z.B. Iterative Improvement, eine Tabu-Suche oder Simulated Annealing implementieren.

2.1.4 Vergleich der Heuristiken

2.2 Zuweisung von Ressourcen

In diesem Abschnitt geht es darum, den Jobs in der Reihenfolge π, die in Abschnitt 2.1 aufgestellt wurde, Ressourcen zuzuweisen. An diese Zuweisung werden zwei Anforderungen gestellt:

1. Die Zuweisung muss zulässig sein. Das heißt, ist eine Ressource einem Job zugewiesen, darf sie den nachfolgenden $m - 1$ Jobs nicht mehr zugewiesen werden. Das Problem der Zulässigkeit einer Zuweisung wird im ersten Unterabschnitt 2.2.1 betrachtet.

2. Die Zuweisung soll möglichst optimal sein. Wenn zwei Jobs, die im Abstand von m in π liegen, die selbe Ressource benutzen können, werden Rüstkosten eingespart, da die Ressource nicht ausgetauscht werden muss. Dieses Problem wird in Unterabschnitt 2.2.2 diskutiert.

Es ist natürlich möglich, dass bei gegebenem π keine zulässige Zuweisung von Ressourcen möglich ist. In diesem Fall muss π nachträglich geändert werden, worauf in Unterabschnitt 2.3 eingegangen wird.

2.2.1 Zulässigkeit der Zuweisung

Abhängig davon, welche Ressourcen für welche Jobs geeignet sind, kann das Zulässigkeitsproblem unterschiedlich schwer zu lösen sein. Folgende Situationen werden betrachtet:

- Alle Ressourcen sind für alle Jobs geeignet (trivial, es müssen m Ressourcen vorhanden sein).
- Die Ressourcenmengen sind disjunkt ($\rho_i \cap \rho_j \neq \emptyset \;\Rightarrow\; \rho_i = \rho_j$).
- Die Ressourcen sind für hierarchische Jobgruppen geeignet ($\iota_q \cap \iota_r \neq \emptyset \;\Rightarrow\; \iota_q \subseteq \iota_r$ oder $\iota_r \subseteq \iota_q$).
- Die ρ_i sind beliebige Teilmengen von R.

Zulässigkeit bei disjunkten Ressourcenmengen

Bei disjunkten Ressourcenmengen können die Jobs in Gruppen unterteilt werden, so dass zu jeder Jobgruppe eine für sie exklusive Menge an zulässigen Ressourcen zur Verfügung steht. Für die Zulässigkeit reicht es aus, statt der Ressourcenmengen ρ_i nur deren Größen $|\rho_i|$ zu betrachten (bei der Optimierung der Zuteilung ist diese Vereinfachung nicht sinnvoll). Mit einem Greedy-Algorithmus kann eine gegebene Reihenfolge π dann auf Zulässigkeit überprüft werden: Der Algorithmus durchläuft π von vorne nach hinten. Jede Jobgruppe erhält einen Counter, der mitzählt, wie viele Ressourcen momentan für sie zur Verfügung stehen. Diese Counter werden zu Beginn mit $|\rho_i|$ initialisiert. An jeder Position in π wird der Counter der zugehörigen Jobgruppe dekrementiert. Nach m Schritten wird dieser Counter wieder inkrementiert. Sollte ein Counter einmal negativ werden, gibt es keine zulässige Ressourcenzuteilung.

Zulässigkeit bei hierarchischen Jobgruppen

Zulässigkeit bei beliebigen Ressourcenteilmengen

Es ist nicht bekannt, ob das Problem, zu entscheiden, ob es bei gegebenem π ein zulässiges Mapping f gibt, $\mathcal{NP}$-vollständiges ist oder ob es einen polynomiellen Algorithmus gibt.

Es besteht allerdings die Vermutung, dass zumindest einen polynomiellen Algorithmus in n gibt. Ein Indiz für diese Vermutung liefert folgendes MIP:

$$\max \quad 0 \tag{2.13}$$

$$\text{s.t.} \quad \sum_{r \in \rho_i} x_{ir} = 1 \qquad i \in N \tag{2.14}$$

$$x_{ir} + x_{jr} \leq 1 \qquad i \in N, j = i+1, \ldots, i+m-1\,, r \in \rho_i \cap \rho_j \tag{2.15}$$

$$x_{ir} \in \{0,1\} \qquad i \in N, r \in \rho_i \tag{2.16}$$

Die Binärvariablen x_{ir} geben hier an, ob dem Job i die Ressource r zugeteilt wird. In diesem Fall ist $x_{ir} = 1$ und sonst $x_{ir} = 0$. Die erste Nebenbedingung 2.14 bewirkt, dass jedem Job genau eine Ressource zugewiesen wird. Durch die zweite Nebenbedingung 2.15 kann jede Ressource jeweils nur einmal an m aufeinander folgende Jobs vergeben werden. Eine Zielfunktion ist nicht notwendig, da die Zulässigkeit nur durch die Nebenbedingungen entschieden wird. Mit diesem MIP können selbst sehr große Instanzen mit $n \approx 100000$ (allerdings mit $m \leq 8$) in weniger als einer Sekunde gelöst werden.

Nebenbedingung 2.15 verhindert immer nur für Paare von Jobs, deren Abstand kleiner als m ist, dass ihnen die selbe Ressource zugewiesen wird. Das ist für die Korrektheit des MIPs zwar ausreichend, durch zusätzliche Nebenbedingungen, die dasselbe für Tripel, Quadrupel, bis hin zu m-Tupeln verhindern, können aber einige fraktionale Lösungen der Relaxation „abgeschnitten“ werden. Für ein k-Tupel von Jobs, die alle in einem Abschnitt von π der Länge m stehen, sieht diese zusätzliche Nebenbedingung so aus:

$$\sum_{l=1}^{k} x_{i_l r} \leq 1 \quad i_l \in N, i_1 < i_2 < \ldots < i_k < i_1 + m, r \in \bigcap_{l=1}^{k} \rho_{i_l} \tag{2.17}$$

Für $k = 2$ ist diese Nebenbedingung identisch mit 2.15. Wird sie für $k = 3, \ldots, m$ zu obigem MIP hinzugefügt, so war bei allen bislang getesteten Instanzen die Relaxation bereits ganzzahlig. Ein Beweis, dass dies tatsächlich bei allen Instanzen der Fall ist, ist noch nicht gelungen. Gelingt er, ist bewiesen, dass es in polynomieller Zeit möglich ist, zu entscheiden, ob es für eine gegebene Jobreihenfolge π und gegebene (beliebige) Ressourcenteilmengen ρ_i möglich ist, allen Jobs eine Ressource zuzuweisen.

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 Beweis zur $\mathcal{NP}$-Vollständigkeit.

2.3 Unzulässige Reihenfolgen

Für den Fall, dass für eine gegebene Reihenfolge π kein zulässiges Mapping f erstellt werden kann, muss π nachträglich verändert werden.

2.3.1 Nachbarschaftssuche

Anfang August hiermit anfangen.

Nachbarschaftenssuchen formulieren, die eine unzulässige Reihenfolge 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?