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 1.3 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

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

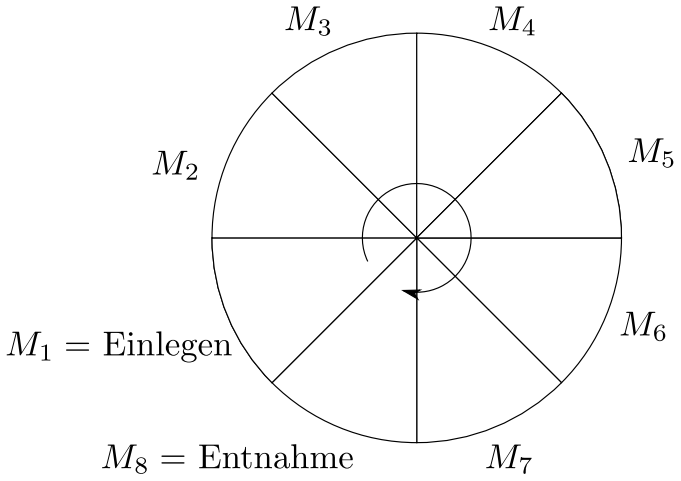

Abbildung 1.1: Kreisförmige Anlage mit $m = 8$ Maschinen.

	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}.

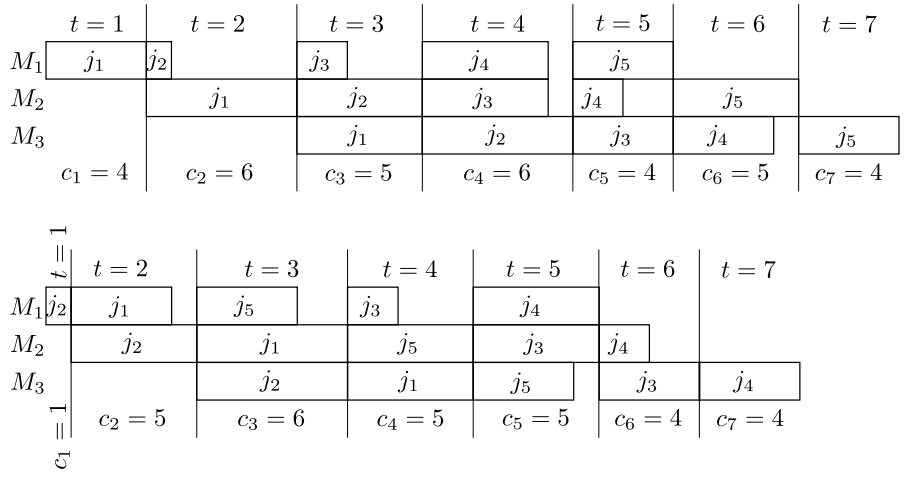

Abbildung 1.3: Gantt-Diagramme der initialen Reihenfolge von Beispiel 1.2 mit $C_{\max} = 34$ und einer optimalen Reihenfolge mit $C_{\max} = 30$.

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 $j' \notin \iota_r$ eingelegt werden soll.

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 $s_{ff} = 0$ angenommen für alle $f \in \mathcal{F}$, dass also keine Rüstkosten innerhalb einer Familie auftreten. Wenn Rüstkosten auftreten, soll nicht mehr nur die Summe aller Zykluszeiten minimiert werden, sondern zusätzlich noch die Summe aller auftretenden

	j_1	j_2	j_3	j_4	j_5
j_1	0	8	5	9	4
j_2	6	0	6	8	8
j_3	4	7	0	5	4
j_4	5	7	8	0	9
j_5	4	5	8	9	0

Abbildung 1.4: Rüstkosten für die Beispielinstanz aus Tabelle 1.2. Beispielsweise treten Rüstkosten von 6 auf, wenn von j_2 auf j_1 gewechselt werden muss.

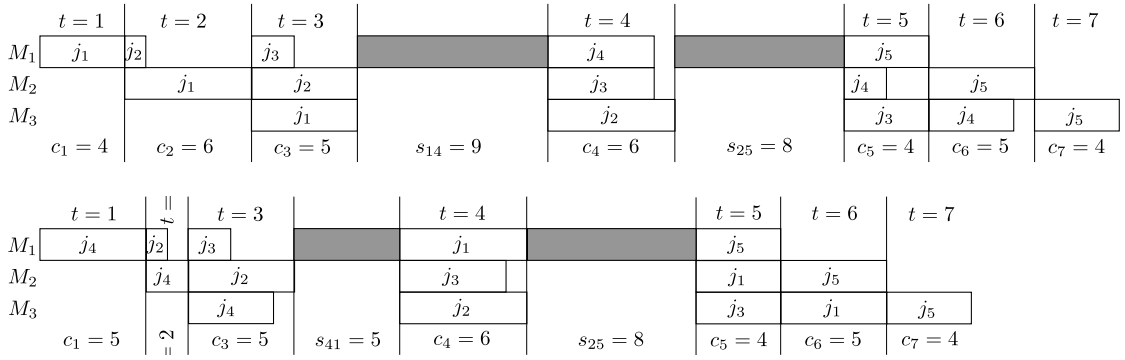

Abbildung 1.5: Initiale und eine optimale Lösung als Gantt-Diagramme für das Beispiel aus 1.2 erweitert um Rüstkosten aus 1.4.

Rüstkosten. Diese Zielfunktion lässt sich als

$$\min \left(C_{\max} + \sum_{f,g \in \mathcal{F}} s_{fg} y_{fg} \right)$$

formulieren, wobei die Variablen y_{fg} angeben, wie oft die Rüstkosten s_{fg} von einem Job aus Familie f zu einem aus Familie g auftreten.

In Abbildung 1.4 ist eine mögliche Rüstkostentabelle für das Beispiel aus 1.2 gegeben. Die initiale Reihenfolge und eine optimale sind in Abbildung 1.5 als Gantt-Diagramm dargestellt.

Insgesamt gilt es, 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 schon das Optimieren bezüglich $C_{\max}$, also ohne Ressourcen und Rüstkosten, $\mathcal{NP}$-schwer ist. Versuche, dieses Problem – oder auch einige Spezialfälle davon – mit ganzzahliger linearer Programmierung optimal zu lösen, waren nur für sehr kleine Instanzen mit $n < 30$ in hinnehmbarer Zeit erfolgreich, was weit hinter praktischen Anforderungen zurückliegt (vgl. Abschnitt 2.1.2. Aufgrund der Komplexität des Gesamtproblems sollen in dieser Arbeit zwei Dekompositionsansätze verfolgt werden:

1. Zunächst wird eine Reihenfolge π aufgestellt, ohne Ressourcen und Rüstkosten zu betrachten, so dass $C_{\max}$ möglichst gut ist. Anschließend wird ein Mapping f basierend auf π erstellt, möglichst ohne nachträgliche Änderung an π vorzunehmen.

2. Es wird zuerst ein Mapping f erstellt, so dass die Ressourcen zulässig zugewiesen sind und die durch die Ressourcenwechsel verursachten Rüstkosten minimal sind. Anschließend werden, ohne f zu verändern, die Jobs so in einer Reihenfolge π angeordnet, dass die Zykluszeiten möglichst minimal sind.

Beide Ansätze liefern natürlich im Allgemeinen keine optimalen Lösungen, da jeweils getrennt bezüglich π und f optimiert wird, obwohl die optimale Lösung von beiden zusammen abhängig ist. Außerdem sind selbst die aus den Ansätzen resultierenden Teilprobleme teilweise noch $\mathcal{NP}$-schwer. Beispielsweise ist bei Ansatz (1) das Berechnen einer $C_{\max}$-optimalen Reihenfolge π in dem Spezialfall, dass es nur eine dominierende Maschine gibt trivial, da $C_{\max}$ bei jeder Reihenfolge identisch ist, und wenn es genau zwei benachbarte dominierende Maschinen gibt, ist es mit dem Algorithmus von Gilmore und Gomory [**?**] in Polynomialzeit lösbar. Bei der anschließenden Zuweisung von Ressourcen ist noch unbekannt, ob ein polynomieller Algorithmus existiert.

Je nachdem, ob die Rüstkosten die Zykluszeiten dominieren oder umgekehrt, ist Ansatz (2) bzw. Ansatz (1) vielversprechender.

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 Verfügbarkeiten 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 gute Alternative.

2 Der erste Dekompositionsansatz

Beim ersten Dekompositionsansatz wird zunächst eine (möglichst gute) 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 einer $C_{\max}$-optimalen Reihenfolge 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, 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. Eine Anwendung des Algorithmus von Gilmore und Gomory als Heuristik für ein Problem mit mehreren dominierenden Maschinen wird in 2.1.3 beschrieben.

2.1.2 Ganzzahlige Lineare Programmierung

Zum Finden einer $C_{\max}$-optimalen Lösung dieses Teilproblems wurde folgendes MIP aufgestellt. Es ist identisch mit dem MIP in Abschnitt 1.3 bis auf die Nichtberücksichtigung der Rüstkosten.

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

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

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

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

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

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

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.

Für Instanzen mit $n \leq 500$ ist die beste nach einer halben Stunde gefundene Lösung in ihrer Güte vergleichbar mit den Heuristiken, die in Unterabschnitt 2.1.3 vorgestellt werden (für einen Vergleich s. 2.1.4). In diesem Sinne kann dieses MIP daher ebenfalls als Heuristik betrachtet werden.

2.1.3 Heuristische Verfahren

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

Abbildung 2.1:

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. Sie arbeitet ähnlich wie die Nearest-Neighbor-Heuristik beim TSP. 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 Bewertungsfunktion 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. Das Vorgehen dieser Heuristik wird anhand der Beispielinstanz 1.2 in Abbildung 2.1 veranschaulicht.

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. Es 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 dominierende Maschinen gewählt. O.B.d.A seien dies die Maschinen M_1 und M_2. Seien

$$d_{\min} \coloneqq \min_{\substack{i \in \{1,2\} \\ j \in J}} p_{ij} \tag{2.7}$$

$$e_{\max} \coloneqq \max_{\substack{i \notin \{1,2\} \\ j \in J}} p_{ij} \tag{2.8}$$

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

wobei $d_{\min} = 0$ und daraus folgend $K = \infty$ erlaubt sind. 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 Maschinen nicht zufällig gewählt werden, sondern so, dass die $d_{\min}$- bzw. $e_{\max}$-Werte optimal sind (und somit auch K).

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 soll der Begriff der *Semidominanz* eingeführt werden. 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 $\mathcal{D}_{\{M_i, M_{i+1}\}}$, wenn M_i und M_{i+1} als benachbarte Maschinen ausgewählt werden. Die Gilmore-Gomory-Heuristik liefert also besonders gute Näherungen, wenn zwei benachbarte Maschinen fast dominierend sind und nur wenige Prozesszeiten auf anderen Maschinen dies verhindern.

Nachbarschaftssuche

Mit einer 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 außerdem z.B. Iterative Improvement, eine Tabu-Suche oder Simulated Annealing implementieren.

Im Rahmen dieser Arbeit wurde ein Simulated Annealing Algorithmus auf Grundlage der xch_{ij}-Nachbarschaft implementiert. Es wird min einer zufälligen initialen Reihenfolge

π begonnen. Pro Iteration werden zufällige i und j bestimmt, mit denen eine benachbarte Lösung $\pi' = xch_{ij}(\pi)$ erzeugt wird. Ist π' besser als die aktuelle Lösung π, d.h. $c(\pi') \geq c(\pi)$, wird π' als neue Lösung übernommen. Ist π' schlechter als π, wird π' nur mit einer bestimmten Wahrscheinlichkeit übernommen. Diese Wahrscheinlichkeit beträgt $e^{\frac{c(\pi')-c(\pi)}{t_k}}$, wobei $(t_k)_k$ eine Nullfolge ist und k die Anzahl der bisher durchgeführten Iterationen angibt. Die Folge $(t_k)_k$ wird auch als „Temperatur“ bezeichnet, die im Laufe des Verfahrens abnimmt. Es wurde mit unterschiedlichen Temperaturen experimentiert. Es wurden

- $t_k = \alpha t_{k-1}$ für unterschiedliche $\alpha \in]0, 1[$ und
- $t_k = \frac{c}{k}$,
- $t_k = \frac{c}{\log k}$ und
- $t_k = c\frac{|\cos \frac{k}{30}|}{k}$

jeweils für unterschiedliche $c \in \mathbb{R}_+$ getestet. Die besten Resultate lieferte $t_k = 50\frac{|\cos \frac{k}{30}|}{k}$. Durch den Cosinus schwankt die Temperatur, so dass abwechselnd verschlechternde Lösungen erlaubt bzw. nicht erlaubt werden. Das Verfahren terminiert, sobald in 1000 aufeinander folgenden Iterationen keine neue beste Lösung gefunden werden konnte.

Dieser Simulated Annealing Algorithmus lieferte in Tests meist etwas bessere Ergebnisse als die übrigen Heuristiken. Die Heuristiken werden im nachfolgenden Unterabschnitt 2.1.4 evaluiert.

2.1.4 Vergleich der Heuristiken

In diesem Abschnitt werden die Rechenergebnisse der Heuristiken aus Unterabschnitt 2.1.3 vorgestellt und miteinander verglichen. Es wurden drei Testreihen mit zufälligen Instanzen erzeugt, jeweils mit $m = 8$ und $n = 50, 100, 150, \ldots, 5000$. Getestet wurde auf einem PC mit Intel(R) Xeon(R) CPU E3-1230 v3, 8 Kerne @ 3.30GHz, und 16GB RAM unter Ubuntu 13.10. Die MIPs wurden mit ZIMPL [Koc04] und CPLEX v12.6 gelöst. Die übrigen Heuristiken wurden in C++ implementiert und mit dem GCC v4.8.1 mit -O2-Optimierung compiliert.

In der ersten Testreihe sind die Maschinen M_3, M_4, M_5 dominierend. Ihre Prozesszeiten wurden gleichverteilt aus dem Intervall $[10, 100]$ gewählt. Die Prozesszeiten der übrigen Maschinen wurden aus dem Intervall $[0, 9]$ gleichverteilt gewählt. In Abbildung 2.2 sind die Resultate aufgetragen. Aufgetragen sind jeweils die absoluten Differenzen $C_{\max,h} - C_{\max,nfs}$ der übrigen Heuristiken (h) zur Non-Full-Schedule-Heuristik (nfs). CPLEX LB und UB bezeichnen die untere und obere Schranke, die CPLEX nach 30 Mi-

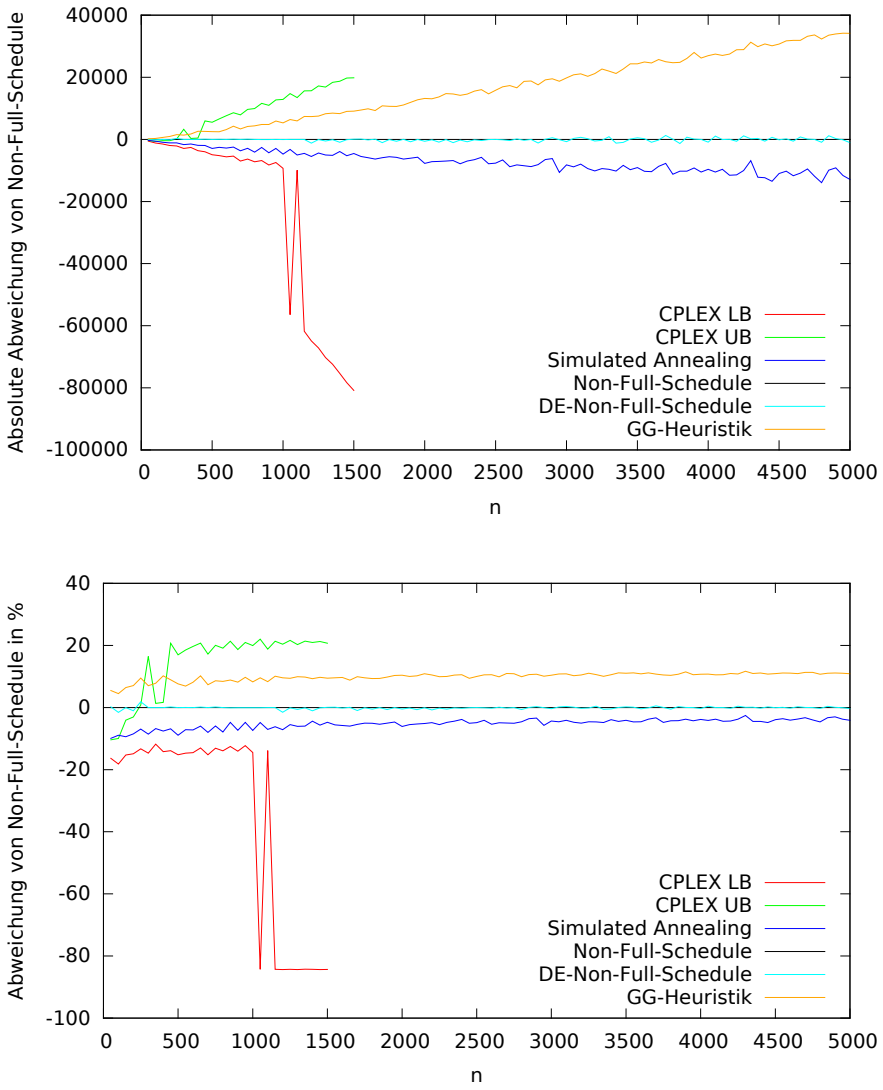

Abbildung 2.2: Absolute (oben) und relative (unten) Unterschiede zwischen der Non-Full-Schedule-Heuristik und den übrigen Heuristiken bei drei dominierenden Maschinen (gleichverteilte Prozesszeiten).

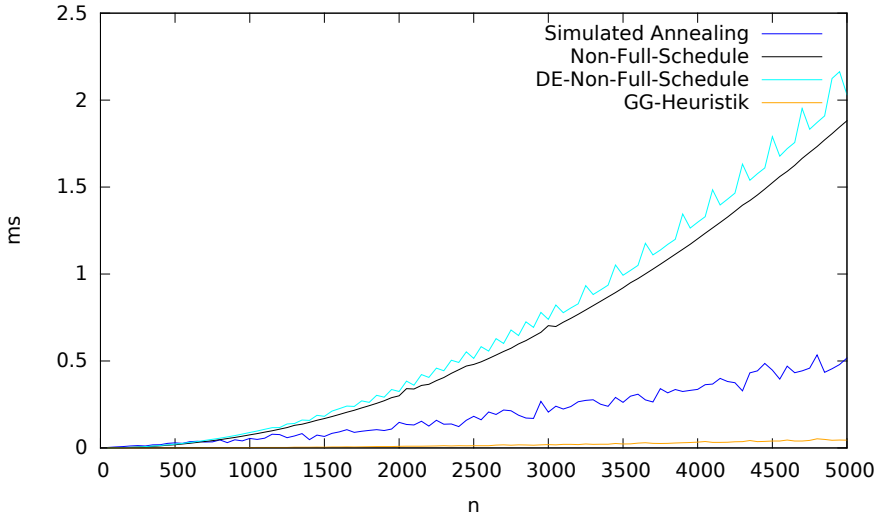

Abbildung 2.3: Laufzeiten der drei Heuristiken bei drei dominierenden Maschinen (gleichverteilte Prozesszeiten).

nuten errechnet hat. Dabei ist die obere Schranke der Zielfunktionswert einer tatsächlich errechneten Lösung und kann daher als Heuristik verwendet werden. Zu sehen ist, dass die Lösungen des MIPs bei sehr kleinen Instanzen ($n < 300$) nach einer halben Stunde Rechenzeit vergleichbar mit den Lösungen der übrigen Heuristiken sind. Allerdings benutzt CPLEX alle 8 Prozessorkerne parallel, während die übrigen Heuristiken nicht parallel implementiert sind. Außerdem liegen die Rechenzeiten der übrigen Heuristiken weit unter einer halben Stunde (s. Abbildung 2.3). Aus diesen Gründen wurde das MIP auch nur bis zu Instanzgrößen von $n = 1500$ berechnet.

Außerdem ist zu erkennen, dass die Double Ended Non-Full-Schedule-Heuristik fast identische Resultate liefert im Vergleich zur Non-Full-Schedule-Heuristik und dass die Gilmore-Gomory-Heuristik hier relativ schlechte Resultate liefert und mit Simulated Annealing die besten Lösungen erzeugt werden. Der Plot der relativen Abweichungen zeigt außerdem, dass diese Abweichungen offenbar nahezu konstant sind zwischen den Heuristiken (nicht beim MIP) und nicht von der Instanzgröße abhängen. Einzig die relative Abweichung des Simulated Annealings scheint mit zunehmender Instanzgröße leicht abzunehmen.

In Abbildung 2.3 sind die Zeiten aufgetragen, die die Heuristiken benötigten, um die in Abbildung 2.2 dargestellten Resultate zu berechnen. Deutlich zu erkennen ist der quadratische Anstieg der beiden Non-Full-Schedule-Heuristiken, wobei die Double Ended Variante etwas langsamer ist, da pro Iteration immer zwei Jobs statt nur einem gesucht werden. Außerdem treten in der Kurve der Double Ended Non-Full-Schedule-Heuristik teilweise kleine Ausreißer nach oben auf. Eine plausible Begründung hierfür konnte nicht

gefunden werden. Die Kurve des Simulated Annealings ist, wie zu erwarten war, sehr chaotisch, was leicht durch die nicht deterministische Funktionsweise des Algorithmus erklärt werden kann. Am schnellsten ist die Gilmore-Gomory-Heuristik, die dafür aber auch vergleichsweise schlechte Resultate liefert.

Die zweite Testreihe unterscheidet sich nur dadurch von der ersten, dass die Prozesszeiten der dominierenden Maschinen nicht gleichverteilt aus dem Intervall $[10, 100]$ ausgewählt wurden, sondern normalverteilt mit einem Erwartungswert von $\mu = 50$ und einer Standardabweichung von $\sigma = 20$. Die Resultate sind analog zur ersten Testreihe in Abbildung 2.4 dargestellt und die Laufzeiten dazu in Abbildung 2.5.

Die Laufzeiten der zweiten Testreihe unterscheiden nur marginal von denen der ersten, was zu erwarten war, da die Laufzeiten der Heuristiken nicht von den Prozesszeiten abhängig sind. Die Kurven in Abbildung 2.4 unterscheiden sich im Verlauf kaum von denen in Abbildung 2.2. Allerdings sind diese Kurven im Vergleich zur ersten Testreihe in etwa um den Faktor 2 gestaucht. Die Resultate der Heuristiken unterscheiden sich hier also weniger. Das muss daran liegen, dass die Prozesszeiten (da sie normalverteilt sind) weniger unterschiedlich sind. Die beiden Non-Full-Schedule-Heuristiken ziehen daraus einen Vorteil, da sie nach eben diesem Prinzip vorgehen, möglichst ähnliche Prozesszeiten zu finden. Auch die Gilmore-Gomory-Heuristik profitiert davon, da beim (theoretischen) Wiederhochskalieren der Prozesszeiten der dritten dominierenden Maschine nur vergleichsweise wenige Zykluszeiten mit vergrößert werden müssen. Und auch CPLEX kommt bei diesen Daten offenbar ebenfalls schneller zu besseren Schranken.

Die dritte und letzte Testreihe wurde mit einem identischen Verfahren wie die zweite Testreihe generiert. Allerdings wurde der Erwartungswert der Prozesszeiten auf Maschine M_3 auf $\mu = 30$ festgesetzt, also um 20 weniger gegenüber M_4 und M_5. Ziel sollte es sein, dass M_4 und M_5 nicht dominierend sind, sondern nur eine geringe Semidominanz haben, so dass die Vorzüge der Gilmore-Gomory-Heuristik zum Tragen kommen.

Und tatsächlich bestätigen die Rechenergebnisse diese Vermutung, wie in Abbildung 2.6 zu sehen ist. Fast alle Heuristiken liefern bessere Ergebnisse als sie es bei den ersten beiden Testreihen tun (in der gleichen Zeit, s. Abbildung 2.7). Simulated Annealing ist hier allerdings nicht mehr die beste Heuristik. Die besten Resultate liefern hier die Gilmore-Gomory-Heuristik, die die größte Verbesserung gegenüber den anderen Testreihen erfährt, und die Non-Full-Schedule-Heuristik. Die Resultate von beiden sind nahezu identisch. Die Double Ended Non-Full-Schedule-Heuristik ist hier bei allen Instanzen schlechter als die Non-Full-Schedule-Heuristik, wenn auch nur um weniger als 1% bzw. bis zu 3% bei sehr kleinen Instanzen. Bemerkenswert ist außerdem, dass die untere Schranke, die von CPLEX für Instanzen mit $n \leq 1500$ berechnet wurde, oft bis an die Ergebnisse der Gilmore-Gomory- und der Non-Full-Schedule-Heuristik heranreicht. Das bedeutet, dass die Resultate dieser beiden Heuristiken nahezu optimal sind. Andererseits sind die oberen Schranken von CPLEX vergleichsweise weit von den Resultaten der Heuristiken entfernt. Als Ausblick könnten diese Heuristiken daher in CPLEX integriert

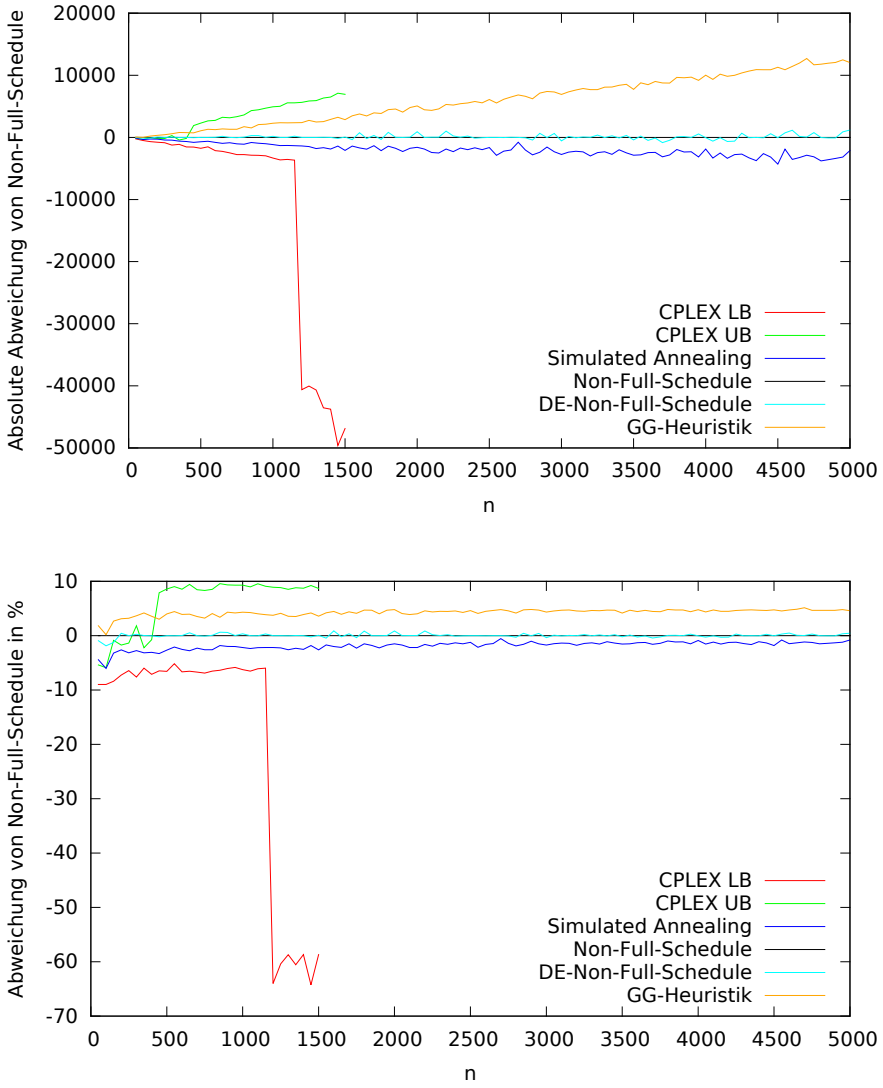

Abbildung 2.4: Absolute (oben) und relative (unten) Unterschiede zwischen der Non-Full-Schedule-Heuristik und den übrigen Heuristiken bei drei dominierenden Maschinen (normalverteilte Prozesszeiten).

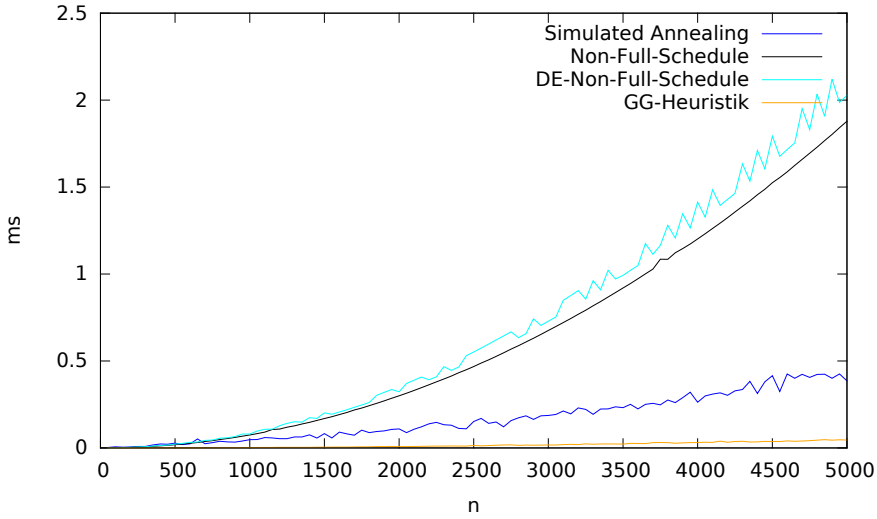

Abbildung 2.5: Laufzeiten der drei Heuristiken bei drei dominierenden Maschinen (normalverteilte Prozesszeiten).

werden, um die oberen Schranken zu verbessern.

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.

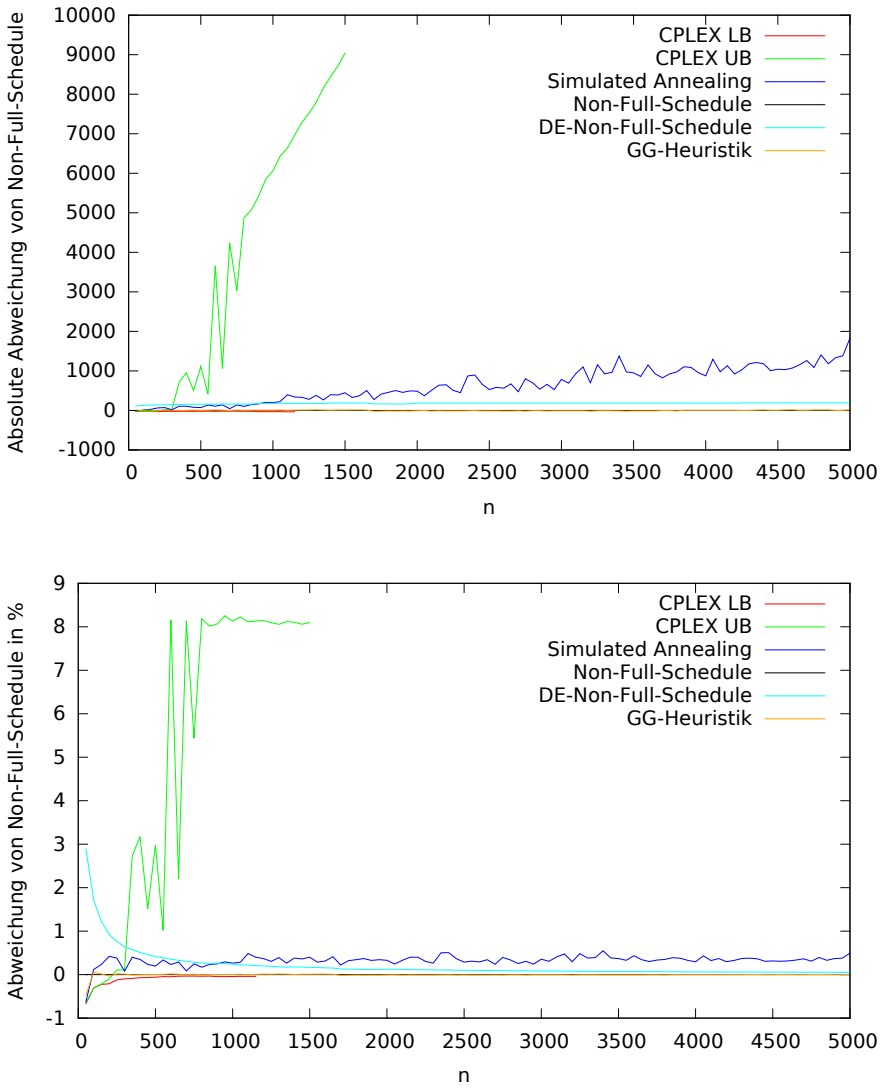

Abbildung 2.6: Absolute (oben) und relative (unten) Unterschiede zwischen der Non-Full-Schedule-Heuristik und den übrigen Heuristiken bei zwei semidominierenden Maschinen (normalverteilte Prozesszeiten).

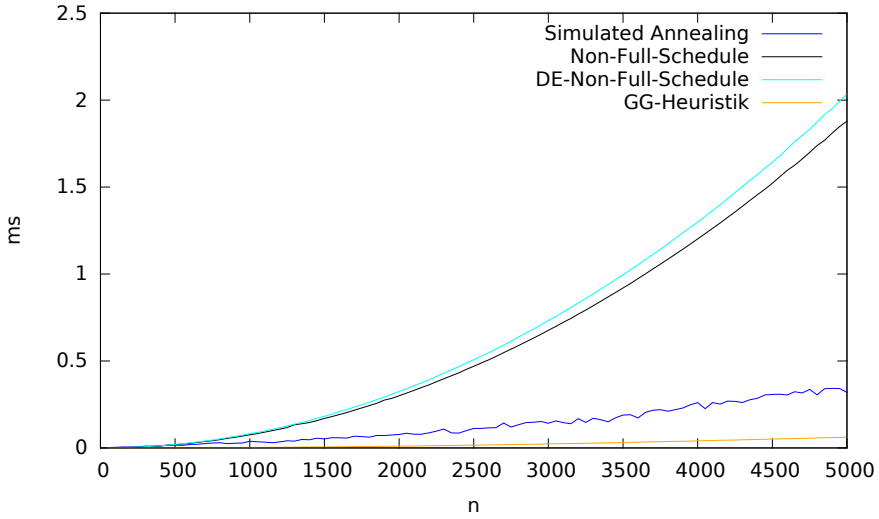

Abbildung 2.7: Laufzeiten der drei Heuristiken bei zwei semidominierenden Maschinen (normalverteilte Prozesszeiten).

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 und beliebigen Ressourcenmengen ein zulässiges Mapping f gibt, $\mathcal{NP}$-vollständig ist oder ob es einen polynomiellen Algorithmus gibt.

Es besteht allerdings die Vermutung, dass es zumindest bei konstantem m 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

Da unbekannt ist, ob das Finden *irgendeiner* Ressourcenzuweisung $\mathcal{NP}$-schwer ist, ist es natürlich ebenfalls unbekannt, ob das Finden einer *optimalen* Ressourcenzuweisung $\mathcal{NP}$-schwer ist.

Eine Ausnahme bildet der Fall von disjunkten Ressourcenteilmengen ρ_i. Immer, wenn zwei Jobs aus einer Familie im Abstand m aufeinander folgen, kann ihnen dieselbe Ressource zugewiesen werden. Wenn zwei Jobs im Abstand m nicht zur selben Familie gehören, gibt es auch keine Möglichkeit, Rüstkosten zu vermeiden oder zu verringern.

Für den allgemeinen Fall lässt sich das MIP aus Unterabschnitt 2.2.1 erweitern:

$$\max \quad \sum_{i=m}^{n} \sum_{r \in \rho_i \cap \rho_{i-m}} y_{ir} \tag{2.18}$$

$$\text{s.t.} \quad \sum_{r \in \rho_i} x_{ir} = 1 \qquad i \in N \tag{2.19}$$

$$x_{ir} + x_{jr} \leq 1 \qquad i \in N, j = i+1, \ldots, i+m-1, r \in \rho_i \cap \rho_j \tag{2.20}$$

$$y_{ir} \leq x_{ir} \qquad i = m, \ldots, n, r \in \rho_i \cap \rho_{i-m} \tag{2.21}$$

$$y_{ir} \leq x_{i-m,r} \qquad i = m, \ldots, n, r \in \rho_i \cap \rho_{i-m} \tag{2.22}$$

$$x_{ir} \in \{0, 1\} \qquad i \in N, r \in \rho_i \tag{2.23}$$

$$y_{ir} \in \{0, 1\} \qquad i = m, \ldots, n, r \in \rho_i \cap \rho_{i-m} \tag{2.24}$$

Für die Binärvariablen y_{ir} gilt $y_{ir} = 1$ genau dann, wenn den Jobs an den Positionen i und $i - m$ die selbe Ressource r zugeteilt ist. Diese Eigenschaft wird durch die Nebenbedingungen 2.21 und 2.22 erzwungen. In der Zielfunktion 2.18 wird daher die Anzahl der y-Variablen, die auf 1 gesetzt sind, maximiert, da so die wenigsten Rüstkosten auftreten. Dieses MIP liefert genau wie jenes aus 2.2.1 in weniger als einer Sekunde auch für sehr große Instanzen eine Lösung. Dieser Umstand lässt vermuten, dass auch das Optimierungsproblem der Ressourcenzuteilung in polynomieller Zeit lösbar ist.

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.

Nachbarschaftssuchen 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 Der zweite Dekompositionsansatz

In diesem Kapitel wird der zweite Dekompositionsansatz vorgestellt. Hierbei wird erst jedem Job eine Ressource zugewiesen, also ein Mapping f erzeugt, und anschließend werden die Jobs in eine Reihenfolge π gebracht. Neben der reinen Ressourcenzuteilung wird im ersten Schritt auch teilweise schon festgelegt, welche Jobs in der späteren Reihenfolge einen Abstand von m haben sollen. So wird sichergestellt, dass die durch das Mapping f vorgeschriebenen Ressourcenzuteilungen auch dazu führen, dass Rüstkosten eingespart werden.

In Abschnitt 3.1 wird erklärt, wie ein optimales Mapping f mit möglichst wenigen Rüstkosten mit einem Binpacking-Ansatz erstellt werden kann. Im nächsten Abschnitt 3.2 wird diskutiert, ob das in Abschnitt 3.1 beschriebene Verfahren auch unzulässige Mappings erzeugen kann bzw. wie diese Situation verhindert oder behoben werden kann. Anschließend wird in Abschnitt **??** beschrieben, wie basierend auf dem gegebenen Mapping f eine Reihenfolge π erstellt wird, so dass die Summe der Zykluszeiten ($C_{\max}$) minimal ist.

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 Rechenergebnisse 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?

Literaturverzeichnis

[Koc04] Koch, Thorsten: *Rapid Mathematical Programming*, Technische Universität Berlin, Diss., 2004. http://opus4.kobv.de/opus4-zib/frontdoor/index/index/docId/834. – ZIB-Report 04-58