FernUniversität in Hagen
Fakultät für Mathematik und Informatik
Lehrgebiet Unternehmensweite Softwaresysteme
Prof. Dr. Lars Mönch

Genetische Algorithmen mit Random-Key-Repräsentation für ein Ablaufplanungsproblem mit Batchmaschinen

Masterarbeit

im Studiengang
Master of Science in Praktischer Informatik
an der Fakultät für Mathematik und Informatik
der FernUniversität in Hagen

Betreuer: Prof. Dr. Lars Mönch
Bearbeiter: Andreas Zeh-Marschke
Bearbeitungszeit: 01.07.2015 - 31.12.2015
Abgabetermin: 04.01.2016

Inhaltsverzeichnis

Inhaltsverzeichnis ii

1. Einleitung 1

1.1. Gegenstand der Arbeit 1

1.2. Zielsetzung 2

1.3. Aufbau der Arbeit 2

2. Ablaufplanung für Batchmaschinen 4

2.1. Prozessbedingungen in der Halbleiterindustrie 4

2.2. Beschreibung des zu lösenden Ablaufplanungsproblems für Batchmaschinen 5

2.3. Diskussion von Vorarbeiten 7

2.4. Beschreibung der ATC-BATC-DH-Swap-Heuristik 8

3. Genetische Algorithmen mit Random-Key-Repräsentationen 17

3.1. Funktionsweise genetischer Algorithmen 17

3.2. Genetische Algorithmen mit Random-Keys 22

3.3. Entwurf konventioneller RKGA 27

4. Implementierung der genetischen Algorithmen 37

4.1. Beschreibung des brkgaAPI-Rahmenwerkes 37

4.2. Implementierungsdetails 41

5. Numerische Experimente 45

5.1. Versuchsplanung 45

5.2. Parameterauswahl für die Heuristiken 48

5.3. Darstellung der numerischen Ergebnisse 56

5.4. Analyse und Interpretation der Ergebnisse 60

6. Zusammenfassung und Ausblick 62

A. Verwendete Bezeichnungen 64

B. Übersicht Programme 65

B.1. Verzeichnis `util` 65

B.2. Verzeichnis `brkgaAPI` 66

B.3. Verzeichnis `scheduling` 66

C. Inhalte der CD 68

C.1. Ausarbeitung 68

C.2. Quellcode 68

C.3. Ergebnisse 68

C.4. Visual Studio C++ 69

Liste der Algorithmen 71

Abbildungsverzeichnis 72

Tabellenverzeichnis 73

Literatur 74

Kapitel 1.

Einleitung

1.1. Gegenstand der Arbeit

Die Produktion von integrierten Schaltkreisen (*IC*) besteht aus teilweise hunderten von Prozessschritten, die auf verschiedenen, zum Teil extrem teuren Maschinen, durchgeführt werden. Daher ist eine hohe Auslastung der Maschinen gewünscht. Darüber hinaus bestehen enge Terminziele, da ein starker Wettbewerb am Markt vorherrscht. Daher ist eine rechtzeitige Fertigstellung der *IC*s gewünscht. Wird die Auslastung maximiert, dann bedeutet dies, dass sich die Fertigstellung einiger *IC*s verzögert. Wird eine frühzeitige Fertigstellung angestrebt, dann werden die Maschinen unter Umständen nicht richtig ausgelastet. Somit ergibt sich die Frage nach einem Vorgehen, bei der beide Ziele, also die Auslastung der Maschinen und die rechtzeitige Fertigstellung der *IC*, berücksichtigt werden.

Die Komplexität wird durch eine Mixtur von verschiedenen Produkten, durch die Verwendung von Batch-Maschinen, die es erlauben mehrere Jobs gleichzeitig auf einer Maschine zu bearbeiten und sehr lange Prozessflüsse, die nicht unterbrochen werden können, erhöht. Eine Beschreibung der Fertigungsprozesse findet sich beispielsweise bei Mönch *et al.* [22]. Siehe dazu beispielsweise auch Uzsoy *et al.* [27], Mehta und Uzsoy [18] oder Klemmt [14].

Im Bereich Diffusion und Oxidation in derFertigung von *IC*s werden die Wafer - hochreine Siliziumscheiben, Rohlinge - für die Halbleiterfertigung, in langwierigen Prozessschritten bearbeitet. In Öfen werden bei hohen Temperaturen chemische Prozesse auf den Wafern angeregt. Diese Prozesse dauern lange. Es ist dabei möglich, verschiedene Jobs gleichzeitig in den Öfen zu bearbeiten. Es können jedoch nur Wafer mit denselben Prozessanforderungen gleichzeitig bearbeitet werden.

Es ergibt sich somit eine Ablaufplanungsproblem. In welcher Reihenfolge können die Jobs (Bearbeitung der Wafer) auf die Maschinen (die Öfen) eingeplant werden. Aus den Jobs können Batches aus mehreren Jobs gebildet werden, wobei nur Jobs mit denselben Prozessanforderungen, aus derselben Familie, gemeinsam in einem Batch enthalten sein können. Somit gibt es inkompatible Familien, also Familien mit unterschiedlichen Prozessanforderungen, die nicht in einem

Batch gemeinsam bearbeitet werden können. Es gibt mehrere parallele, dies bedeutet auch identische, Batch-Maschinen. Eine effektive Ablaufplanung ist wichtig, damit die verschiedenen Ziele, eine Optimierung bezüglich der Auslastung der Maschinen und Einhaltung der gewünschten Fertigstellungstermine, erreicht werden kann. In dieser Arbeit wird jedoch nur die total gewichtet Verspätung (TWT).

Die Erstellung eines solches Ablaufplanes ist oftmals NP-schwer. Daher werden Metaheuristiken angewendet, um Lösungen zu erzielen. In Almeder und Mönch [1] sind verschiedene Heuristiken für die Lösung dieses Ablaufplanungsproblems verglichen worden. Für die Verwendung von Random-Keys wurde von Bean [3] ein Genetischer Algorithmus (GA) entwickelt, ein *Random-Key Genetic Algorithm* ($RKGA$). Bei Goncalves und Resende [10] wurde darauf eine *Biased-*$RKGA$ ($BRKGA$) entworfen, der bessere Ergebnisse als der $RKGA$ liefert.

1.2. Zielsetzung

In dieser Arbeit werden die nachfolgenden Ziele verfolgt:

1. Entwicklung eines GA auf Basis von Random-Key-Repräsentation mit lokaler Suche nach Verbesserungen.
2. Implementierung der Algorithmen, basierend auf dem von Toso und Resende [26] erstellten C++-Framework $brkgaAPI$, für die Erstellung der Ablaufplanungen.
3. Durchführung umfangreicher numerischer Experimente mit den Algorithmen.
4. Vergleich der Ergebnisse mit einem konventionellen Verfahren, eines List-Scheduling-Verfahren, wie bei Almeder und Mönch [1].
5. Vergleich der Ergebnisse mit anderen Metaheuristiken, wie dies ebenfalls bei Almeder und Mönch [1] durchgeführt wurde.

Es soll damit untersucht werden, ob die Ergebnisse, die mit $BRKGA$ erzielt werden, die Verbesserungen mit den Metaheuristiken, die in Almeder und Mönch [1] beschrieben sind, erreicht oder sogar verbessert werden.

1.3. Aufbau der Arbeit

Im Kapitel 2 wird zuerst das Problem in das Umfeld der Halbleiterindustrie eingeordnet (Abschnitt 2.1). Dann wird das Ablaufplanungsproblem, welches in dieser Arbeit bearbeitet wird, beschrieben (Abschnitt 2.2). Nach der Diskussion von

relevanten Arbeiten zu diesem Problem (Abschnitt 2.3) wird ein konventionelles List-Scheduling-Verfahren beschrieben (Abschnitt 2.4), welches Referenzwerte für den Vergleich der mit den Werte aus Random-Key-Verfahren herangezogen wird. Dieses Verfahren dient auch bei Almeder und Mönch [1] als Referenz.

Im Kapitel 3 werden genetische Algorithmen beschrieben. Ausgehend von der allgemeinen Beschreibung genetischer Algorithmen (Abschnitt 3.1), werden anschließend genetische Algorithmen, die auf Random-Keys basieren, wie von Bean [3] eingeführt, dargestellt (Abschnitt 3.2). Ein wichtiger Teil bei den genetischen Algorithmen, die auf Random-Keys basieren, ist die Frage, wie die Repräsentation einer Folge von Random-Keys für das gegebene Problem aussieht. Die in dieser Arbeit verwendeten Dekoder, welche die Dekodierung von der Folge der Random-Keys in eine zulässige Ablaufplanung für das Problem erarbeiten, werden am Ende vom Kapitel 3 beschrieben (Abschnitt 3.3).

Im Kapitel 4 werden Implementierungsdetails erläutert. Es wird insbesondere das Rahmenwerk für den *brkGA*, das von Toso und Resende [26] erstellt wurde, beschrieben (Abschnitt 4.1). Auch Implementierungsdetails zu den Decodern und zum Gesamtablauf werden dargestellt (Abschnitt 4.2)

Im Kapitel 5 werden die numerischen Experimente beschrieben. Nach Beschreibung der Versuchsplanung (Abschnitt 5.1) wird die Wahl der Parameter für die Heuristiken erläutert (Abschnitt 5.2). Anschließend werden Ergebnisse aus der Durchführung der Experimente aufgeführt (Abschnitt 5.3). Die Ergebnisse werden anschließend analysiert (Abschnitt 5.4).

Im abschließend Kapitel 6 erfolgt die Zusammenfassung der Ergebnisse und ein Ausblick.

Kapitel 2.

Ablaufplanung für Batchmaschinen

Im diesem Kapitel werden zuerst Herausforderungen der Halbleiterindustrie für die Ablaufplanung beschrieben (Abschnitt 2.1). Dann wird das Ablaufplanungsproblem, welches in dieser Arbeit bearbeitet wird, formal beschrieben (Abschnitt 2.2). Nach der kurzen Diskussion von relevanten Arbeiten zu diesem Problem (Abschnitt 2.3) wird ein konventionelles List-Scheduling-Verfahren beschrieben (Abschnitt 2.4), welches Referenzwerte für den Vergleich der mit den Werte aus Random-Key-Verfahren herangezogen wird. Dieses Verfahren dient auch bei Almeder und Mönch [1] als Referenz.

2.1. Prozessbedingungen in der Halbleiterindustrie

Durch den wachsenden Bedarf an *IC*s in der Industrie ist eine schnelle Produktion von *IC*s wichtig. In Klemmt [14] wird der Fertigungsprozess von Halbleiter ausführlich beschrieben. Der Gesamtprozess besteht aus vielen einzelnen Prozessschritten. Diese einzelnen Prozessschritte können kurz aber auch teilweise sehr lange, mehrere Stunden, dauern. Teilweise sind hierzu extrem teure Maschinen notwendig. Da es einen hohen Wettbewerb bei den Halbleitern gibt, ist ein Ziel bei der Produktion, die Produkte termingerecht fertig zu stellen.

Der Bereich Diffusion und Oxidation in der Fertigung der *IC*s wird nun genauer betrachtet. Beim Dotieren wird ein Trägergas mit einem Dotierstoff angereichert. Dieses Gemisch wird dann bei einer hohen Temperatur über den Wafer geleitet. In Öfen werden bei den hohen Temperaturen chemische Prozesse auf den Wafern angeregt. Diese Prozesse dauern lange. Es ist dabei möglich verschiedene Wafer gleichzeitig in den Öfen zu bearbeiten. Es können jedoch nur Wafer mit denselben Prozessanforderungen bezüglich verschiedener Kriterien (Gas, Dotierstoffe, Temperatur, Dauer) parallel bearbeitet werden.

Für die Ablaufplanung ist ein Job eine Gruppe von Wafern, welche zu einer Familie gehören, die somit gleiche Prozessbedingungen haben. Die Abaufplanung erstellt eine Reihenfolge, in welcher die Jobs in der Öfen bearbeitet werden. Die

Öfen werden dabei als Maschinen bezeichnet. Die Jobs sind in Familien eingeteilt. Für die Jobs aus einer Familie gelten dabei die selben Prozessbedingungen, so dass die Jobs parallel auf der Maschine bearbeitet werden können. In dieser Arbeit haben alle Jobs einer Familie die selbe Prozesszeit. Jobs aus unterschiedlichen Familien dagegen haben unterschiedliche Prozessbedingungen und können somit nicht gleichzeitig bearbeitet werden. Familien mit unterschiedlich Prozessbedingungen sind daher inkompatibel. Somit können die Jobs in inkompatible Familien eingeteilt werden. Jobs aus derselben Familie können dann zu Batches zusammen gefasst werden. Für die Bearbeitung in den Öfen werden Batches bearbeitet. Die Jobs in einem Batch werden hierbei parallel bearbeitet. Das bedeutet, dass die Bearbeitung der Jobs gleichzeitig beginnt und somit auch gleichzeitig endet. Vor der Bearbeitung auf einer Maschine können daher mehrere Jobs aus unterschiedlichen Familien warten. Wenn ein Ofen frei wird, ist zu entscheiden, welche Jobs zu einem Batch zusammen gefasst werden. Wenn es mehrere Batches gibt, dann ist zu entscheiden, welcher Batch als nächstes bearbeitet wird. Es kann teilweise extrem lange Bearbeitungszeiten und kürzere Bearbeitungszeiten geben (siehe Mehta und Uzsoy [18]).

In dieser Arbeit sind die Maschinen als parallele Batch-Maschinen modelliert. Die Jobs sind zerlegt in inkompatible Familien, wobei eine Batch-Maschine nur Jobs der gleichen Familie gleichzeitig bearbeiten kann. Als Maß für die Performanz dient dabei die totale gewichtete Verspätung (TWT). Die Verspätung T_j eines Jobs ist dabei die Differenz zwischen Fertigstellungstermin (C_j) und gewünschtem Fertigstellungstermin (d_j), wenn der Fertigstellungstermin nach dem Wunschtermin ist, ansonsten ist die Verspätung gleich 0. Somit ist $T_j = max(0, C_j - d_j)$. Jeder Job hat ein Gewicht w_j, das die Bedeutung oder Wichtigkeit des Jobs darstellt. Damit ist dann die gewichtete Verspätung eines Jobs durch $w_j T_j$ gegeben, und das TWT-Maß ist dann die Summe über die gewichteten Verspätungen aller Jobs. Der Ablaufplan soll gemäß des TWT-Leistungsmaßes optimiert werden, also ein möglichst niedriger Wert von TWT erzielt werden.

2.2. Beschreibung des zu lösenden Ablaufplanungsproblems für Batchmaschinen

Gegeben seien n Jobs. Alle Jobs sind zum Beginn des Betrachtungszeitraums verfügbar. Jeder Job gehört zu genau einer der f verschiedenen inkompatiblen Familien. Durch die Funktion $f(j)$ wird jedem Job j seine Familie zugeordnet. Ist n_s die Anzahl der Jobs in der Familie s, dann gilt $n = \sum_{s=1}^{f} n_s$. Alle Jobs aus einer Familie haben die identische Bearbeitungszeit $p_{f(j)} = p_s$, wenn $s = f(j)$ gilt. Eine Unterbrechung der Prozessbearbeitung ist nicht möglich. Das bedeutet ein gestarteter Job wird ohne Unterbrechnung bearbeitet.

Es gibt m funktionsgleiche und identische Maschinen. Da die Maschinen funktionsgleich sind, können die Jobs parallel auf den Maschinen bearbeitet werden. Da die Maschinen identisch sind, hängt die Bearbeitungszeit auf den Maschinen nur

vom jeweiligen Job ab. Die gegebenen Maschinen sind Batch-Maschinen. Das bedeutet, dass die Maschinen Batches bearbeiten. In einem Batch sind mehrere Jobs aus derselben Familie zusammen gefasst. Die maximale Kapazität eines Batches ist dabei durch B gegeben. Mit B_{ks} wird der k-te Job der Familie s bezeichnet

Jeder Job j hat ein Gewicht w_j, einen gewünschten Fertigstellungstermin d_j und einen Fertigstellungstermin C_j. Die Bearbeitungszeit eines Jobs ist gegeben durch die Bearbeitungszeit der entsprechenden Familie, es ist also die Bearbeitungszeit des Jobs j gegeben durch $p_{f(j)}$.

Da die Bearbeitung auf den Maschinen nicht unterbrochen werden kann, kann der Fertigstellungstermin C_j eines Jobs j berechnet werden, wenn der Beginn t der Bearbeitung bekannt ist. Es gilt $C_j = t + p_{f(j)}$. Die Verspätung von Job j ist somit $T_j = (C_j - d_j)^+$, wobei $x^+ = \max(0, x)$ ist. Die gewichtete Verspätung von Job j ist somit $w_j T_j$. Damit ist der TWT-Wert gegeben durch $TWT = \sum_{j=1}^{n} w_j T_j$.

Das Ablaufplanproblem besteht darin, für eine gegebene Probleminstanz einen Ablaufplan zu erstellen, für den ein minimaler TWT-Wert vorliegt.

Die verwendete Notation ist im Anhang, in der Tabelle A.1, ausgelistet.

Das beschriebene Ablaufplanungsproblem wird im nachfolgenden klassifiziert. Hier wird als Klassifizierung die $(\alpha|\beta|\gamma)$-Notation verwendet, die von Graham *et al.* [12] eingeführt wurde und Pinedo [24] beschrieben ist. Die Problembeschreibung orientiert sich hierbei an Almeder und Mönch [1].

Das α-Feld beschreibt die verwendeten Maschinen. Für die gegebene Problemstellung gibt es m parallele und identische Maschinen. Daher gilt $\alpha = Pm$.

Das Feld β beschreibt charakteristische Eigenschaften und Restriktionen für den Prozessablauf. Für das gegebene Problem ist die Bildung von Batches wesentlich. Es können somit Jobs zu einem Batch zusammen gefasst werden. Hierbei können jedoch nur Job aus derselben Familie, also kompatible Jobs, zu einem Batch zusammengefasst werden. Die Bearbeitung der einzelnen Jobs im Batch erfolgt dabei parallel, also gleichzeitig. Die Bearbeitungszeit ist hierbei die Bearbeitungszeiten der Jobs im Batch. Da alle Jobs in einem Batch zur selben Familie gehören, ist auch die Bearbeitungszeit identisch. Bei Brucker [4] wird dies als *p-batch* bezeichnet. Hierbei ist bei Brucker die Bezeichnung *p-batch* allgemeiner gefasst. Die Bearbeitungszeit ist dort die längste Bearbeitungszeit der Jobs, die den Batch bilden, wobei die Bearbeitung gleichzeitig beginnt. In dieser Arbeit sind die Bearbeitungszeiten der Jobs derselben Familie identisch. Somit gilt $\beta = p\text{-}batch,\ incompatible$.

Im Feld γ wird die Zielfunktion beschrieben, also das Maß für die Bewertung der gefundenen Ablaufpläne. Es ist die total gewichtete Verspätung (total weighted tardiness, TWT). Somit ist $\gamma = TWT$.

Zusammenfassend ergibt sich somit

$$Pm \mid p - batch,\ incompatible \mid TWT \tag{2.1}$$

als Klassifizierung des gegebenen Ablaufplanproblems.

Wird die Größe der Batches auf 1 gesetzt, also $B = 1$ und wird nur auf einer Maschine gearbeitet, also $m = 1$, dann reduziert sich das Problem zum Ablaufplanungsproblem mit der Klassifizierung $1 \mid\mid TWT$. Das Problem $1 \mid\mid TWT$ ist *NP*-schwer (siehe Lawler [15] oder auch Pinedo [24], Theorem 3.6.2). Daher werden für Lösungen der Ablaufplanung bei großen Probleminstanzen in Heuristiken eingesetzt.

Eine bewährte Heuristik ist die ATC-$BATC$-Heuristik, die im Abschnitt 2.4 dargestellt wird. Die Ergebnisse, der Ablaufplanungen, die mit dieser Heuristik erstellt werden, dienen als Referenz. Verbesserungen der Ablaufplanungen werden mit Hilfe der Dekompositionsheuristik und der abschießenden Swap-Heuristik erzielt, was ebenfalls im Abschnitt 2.4 dargestellt wird. Zusammen ergibt dies die ATC-$BATC$-DH-Swap-Heuristik.

Bei Balasubramanian *et al.* [2] werden verschiedene Varianten von Algorithmen untersucht, wobei der Kern immer ein genetischer Algorithmus (GA) ist. In der Arbeit von Almeder und Mönch [1] werden zwei andere Heuristiken untersucht und mit dem GA aus Balasubramanian *et al.* [2] verglichen. Es sind dies die beiden Heuristiken Variable Nachbarschaftssuche (VNS) und Ameisenalgorithmus (*ACO*). Hierbei wird bei Almeder und Mönch [1] festgestellt, dass VNS besser ist als *ACO* oder die verschiedenen Varianten des GA bei Balasubramanian *et al.* [2]. Diese verschiedenen Heuristiken verbessern die Ergebnisse der ATC-$BATC$-DH-Swap-Heuristik.

Bean [3] hat genetische Algorithmen auf Basis von Random-Keys vorgestellt. Dieser Algorithmus wird als Random-Key Genetic Algorithm (*RKGA*) bezeichnet. Von Goncalves und Resende [10] wurde eine Ergänzung / Erweiterung, der Biased-*RKGA* (*BRKGA*) vorgeschlagen. Bei Goncalves *et al.* [11] wurde diese beide Verfahren verglichen. Dabei wurde gezeigt, dass *BRKGA* besser ist als *RKGA*. Die genetischen Algorithmen und speziell *RKGA* und *BRKGA* werden im Kapitel 3 untersucht.

2.3. Diskussion von Vorarbeiten

Batching-Probleme werden von vielen Autoren behandelt, da es sich um Probleme handelt, die in der realen Welt, insbesondere in der Halbleiterindustrie sehr oft vorkommen. In Übersichtsartikeln haben Brucker *et al.* [5], Potts und Kovalyov [25] und auch Mathirajan und Sivakumar [16] diese Problematik beleuchtet.

Bei Balasubramanian *et al.* [2] wird für die Problemstellung, die auch in dieser Arbeit untersucht wird, also für $Pm|p\text{-}batch, incompatible|TWT$, verschiedene Heuristiken entwickelt. Dort sind zwei Heuristiken, basierend auf GA entwickelt worden. Dort wird insbesondere auf drei Arbeiten verwiesen, die allerdings alle nur auf einer Maschine agieren. Bei Mehta und Uzsoy [18]) ist das

Ziel die Minimierung der Verspätung bei einem Batchingproblem mit inkompatiblen Familien auf einer Maschine, also dem Problem $1|p\text{-}batch, incompatible|\sum T_j$. Hier wird eine Dekompositionsheuristik eingeführt, bei der die Positionen der Jobs verändert werden können. Bei Perez *et al.* [23] wird das Problem und die Verfahren auf die gewichtete Verspätung erweitert, also auf das Problem $1|p\text{-}batch, incompatible|TWT$. Bei Devprua *et al.* [7] wird bei fixierter Position der Batches ein Verfahren, das Swapping, eingeführt, bei dem Jobs zwischen den Batches derselben Familie getauscht werden können. Bei Balasubramanian *et al.* [2] wird dieses Swapping auf parallele Maschinen übertragen. Im Rahmen der Beschreibung der Berechnung der Referenzwerte (siehe Abschnitt 2.4) werden neben dem List-Scheduling-Verfahren auch die Dekompositionsheuristik und das Swapping detaillierter beschrieben, da beide Heuristiken auch im Rahmen dieser Arbeit verwendet werden.

In der Arbeit von Almeder und Mönch [1] wird das Batchingproblem mit inkompatiblen Familien auf parallelen Maschinen behandelt. Es wird ebenso die Problemstellung $Pm|p\text{-}batch, incompatible|TWT$, wie auch bei Balasubramanian *et al.* [2] und in dieser Arbeit. Dabei werden neben dem List-Scheduling-Verfahren, welches als Referenz für die Vergleiche herangezogen wird, dort verschiedene heuristische Verfahren untersucht, der Ameisenalgorithmus (ACO) und die variable Nachbarschaftssuche (VNS). Die Ergebnisse werden auch mit Berechnungen mit genetischen Algorithmen (GA) verglichen, die bei Balasubramanian *et al.* [2] entwickelt und dargestellt wurden.

Somit führt diese Arbeit die beiden Arbeiten von Balasubramanian *et al.* [2] und Almeder und Mönch [1] fort, indem für die gegebene Problemstellung $Pm|p\text{-}batch, incompatible|TWT$ weitere Heuristiken untersucht werden und mit den bekannten Ergebnissen verglichen werden.

In Mönch *et al.* [21] wird ein Ablaufproblem auf parallelen Maschine und inkompatiblen Jobs betrachtet, wobei die Jobs unterschiedliche Bereitstellungstermine haben. Es wird somit das Problem $Pm|r_j\ p\text{-}batch,\ incompatible|TWT$ behandelt.

2.4. Beschreibung der ATC-BATC-DH-Swap-Heuristik

Als Referenz für den Vergleich der Heuristiken dient ein einfaches, aber erfolgreiches Verfahren, ein List-Scheduling-Verfahren, das im Nachfolgenden beschrieben wird.

Das Verfahren (siehe Algorithmus 2.1) beginnt (Schritt 1) mit der $ATC\text{-}BATC$-Heuristik (siehe Abschnitt 2.4.1). Damit werden Batches aus den Jobs der Probleminstanz gebildet und den Maschinen zugeordnet. Der Ablaufplan ist somit eine Liste von Batches, die mit L_{Batch} bezeichnet wird. Anschließend (Schritte 02 - 04) wird mit der Dekompositionsheuristik (siehe Abschnitt 2.4.2) die Reihenfolge der

Batches je Maschine optimiert. Dazu werden die Batches der jeweiligen Maschinen selektiert und geprüft, ob eine veränderte Reihenfolge ein besseres Ergebnis liefert. Der abschließende Teil (Schritte 05 - 07) ist die Swap-Heuristik (siehe Abschnitt 2.4.3) je Familie. Hier werden maschinenübergreifend Jobs der selben Familie zwischen Batches ausgetauscht. Es werden somit die Batches der einzelnen Familien selektiert und dann mit der Heuristik Verbesserungen gesucht.

Algorithmus 2.1 : Gesamtablauf Referenz

Data : Probleminstanz P;
Result : L_{Batch} für die Jobs gemäß ATC-$BATC$-DH-Swap-Heuristik;
1 L_{Batch} := führe ATC-$BATC$-Heuristik durch;
2 **foreach** *(Maschine i)* **do**
3 | L_{Batch} := führe $Dekompositionsheuristik$ (L_{Batch}, i) durch;
4 **end foreach**
5 **foreach** *(Familie s)* **do**
6 | L_{Batch} := führe $Swap - Heuristik$ (L_{Batch}, s) durch;
7 **end foreach**

2.4.1. ATC-BATC-Heuritik

Ziel der ATC-$BATC$-Heuristik ist es, einen zulässigen Ablaufplan mittels eines List-Scheduling-Verfahrens zu erstellen. Im Algorithmus 2.2 ist das Verfahren beschrieben.

Dieser Algorithmus bestimmt einen zulässigen Ablaufplan L_{Batch} für die Jobs der Probleminstanz zum Parameter κ, der dem Algorithmus übergeben wird.

Zuerst wird der aktuelle Zeitpunkt t der Bearbeitung wird auf 0 gesetzt (Schritt 01), die Liste J der noch nicht verplanten Jobs der Probleminstanz erstellt (Schritt 02). Im Schritt 03 wird jede der Maschine zum Zeitpunkt $t = 0$ auf frei gesetzt.

In der Schleife (Schritte 04 - 18) wird die Liste der der noch nicht verplanten Jobs J durchlaufen. Sie endet, wenn alle Jobs im Ablaufplan verplant sind.

Es wird die nächste freie Maschine gesucht (Schritt 05). Sind mehrere Maschinen zum selben Zeitpunkt frei, dann wird die Maschine mit dem kleinsten Index. ausgewählt. Der Bearbeitungszeitpunkt dann auf den Zeitpunkt gesetzt, an dem die selektierte Maschine frei wird (Schritt 06).

Im Schritt 07 wird für jeden Job j, der sich in der Liste der noch nicht verplanten Jobs J befindet, der *Apparent Tardiness Cost*-(ATC)-Index $I_j(t)$ für den Zeitpunkt t und zum Parameter κ berechnet. Der Wert für κ wird nach der Beschreibung des Algorithmus erläutert. Der ATC-Index wird gemäß der Formel

$$I_j(t) = \frac{w_j}{p_{f(j)}} \exp\left(-\frac{(d_j - p_{f(j)} - t)^+}{\kappa \cdot \overline{p}}\right) . \quad (2.2)$$

Algorithmus 2.2 : ATC-BATC-Heuristik

Data : κ Parameter
Result : L_{Batch} Ablaufplan zum Parameter κ;

```
1  t := 0;
2  J := Liste der noch nicht verplanten Jobs;
3  foreach (Maschine i) do frei[i] := 0;
4  repeat
5  |  maschine := bestimme nächste freie Maschine;
6  |  t := frei[maschine];
7  |  berechne ATC-Werte für jeden Job j in J für t und κ;
8  |  sortiere J nicht wachsend nach dem ATC-Wert;
9  |  foreach (Familie s) do
10 |  |  Batch_s := bilde Batch für Familie s;
11 |  |  berechne BATC-Werte für Batch Batch_s;
12 |  end foreach
13 |  s := bestimme Familie mit dem höchsten BATC-Werte für Batch_s;
14 |  setze Parameter für Batch_s;
15 |  frei[maschine] := t + p_s;
16 |  entferne Jobs von Batch_s aus J;
17 |  L_Batch := L_Batch ∪ {Batch_s};
18 until (J = ∅);
19 berechne TWT-Wert;
```

berechnet. Hier sind w_j das Gewicht des Jobs und $p_{f(j)}$ die Bearbeitungszeit für die Jobs der Familie $f(j)$, zu der der Job j gehört. Gemäß Voraussetzung habe alle Jobs aus einer Familie dieselbe Bearbeitungszeit. Der Term $(d_j - p_{f(j)} - t)^+$ überprüft, ob der Job eine Verspätung hat. Es ist d_j der gewünschter Fertigstellungstermin vom Job j. Der berechnete Fertigstellungstermin vom Job, wenn die Bearbeitung zum Zeitpunkt t startet, ist $t + p_{f(s)}$. Ist $d_j > p_{f(j)} + t$, also $d_j - p_{f(j)} - t > 0$, dann würde der Job rechtzeitig, also vor dem gewünschter Fertigstellungstermin fertig werden. Damit ist der Exponent der Exponentialfunktion kleiner als 0, der Wert der e-Funktion somit kleiner als 1. Der ATC-Index ist damit kleiner als der Wert vom Term $\frac{w_j}{p_{f(j)}}$. Falls der Job nicht rechtzeitig fertig wird, also $d_j < p_{f(j)} + t$, dann ist der Exponent der e-Funktion gleich 0, der ATC-Index ist damit gleich dem Term $\frac{w_j}{p_{f(j)}}$. Der Parameter $\overline{p}$ ist der Durchschnitt der Bearbeitungszeiten der noch nicht verplanten Jobs, also der Jobs in der Liste J. Darüber hinaus ist κ ein Skalierungsfaktor, der weiter unten noch genauer untersucht wird.

Dann werden die Jobs nach nicht wachsendem ATC-Index sortiert (Schritt 08). Da es Jobs mit demselben ATC-Index geben kann, werden diese Jobs gemäß Ihrer ursprünglichen Reihenfolge (in der Probleminstanz) sortiert.

Für jede Familie s wird ein Batch aus Jobs aus der Liste der noch nicht verplanten Jobs J gebildet und bewertet (Schritte 09 - 12). Für jede Familie, für die es noch Jobs in der Liste der nicht verplanten Jobs gibt, wird ein Batch geformt. Hierzu

werden die ersten, maximal B (Batchgröße) Jobs gemäß der nach dem ATC-Index sortierten Liste der Jobs (Schritt 10). Wenn für eine Familie weniger als B Jobs zur Verfügung stehen, dann bilden diese den Batch, es ist ein unvollständiger Batch. Jedem Batch *Batch* wird ein Index zugeordnet (Schritt 11), der *Batch-Apparent-Tardiness-Cost*-($BATC$-)Index $I_{BATC}(k,s,t)$, gemäß

$$I_{BATC}(k,s,t) = \sum_{j \in B_{ks}} I_j(t) \; . \tag{2.3}$$

Der $BATC$-Index ist also die Summe der ATC-Indizes der Jobs im Batch.

Im Schritt 13 wird der Batch der Familie mit dem höchsten $BATC$-Index selektiert. Wenn mehrere Batches denselben $BATC$-Index haben, dann wird der Batch mit der Familie mit dem niedrigsten Index selektiert. Der selektierte Batch wird auf der im Schritt 05 ausgewählten freien Maschine eingeplant (Schritt 14). Die Fertigstellungstermine der Jobs aus dem Batch (und damit für den Batch) wird auf $t + p_s$ gesetzt.

Der nächste Termin für die ausgewählte Maschine, an dem sie wieder frei wird, wird auf $t + p_s$ gesetzt (Schritt 15). Die Jobs aus $Batch_s$ werden aus der Liste der noch nicht verplanten Jobs entfernt (Schritt 16). Dadurch wird diese Liste mit jedem Durchlauf kleiner. Der erstellte $Batch_s$ wird zum Ablaufplan L_{Batch} hinzugefügt (Schritt 17).

Da stets alle alle noch nicht verplanten Jobs in der Liste J enthalten sind, ist für jede Familie höchstens der letzte der erstellten Batches nicht vollständig gefüllt. Alle anderen Batches sind jeweils voll belegt.

Für den erstellten Ablaufplan wird dann im Schritt 19 der TWT-Wert berechnet.

Je nach dem Wert vom Skalierungsfaktor κ bei der Berechnung der ATC-Indizes kommen unterschiedliche Ablaufpläne zustande und somit auch unterschiedliche TWT-Werte. Es werden verschiedene κ-Werte untersucht und dabei abschließend der κ-Wert und damit der dazu gehörige Ablaufplan gewählt, welcher zum kleinsten TWT-Wert führt. Gemäß Balasubramanian *et al.* [2] und Almeder und Mönch [1] wird eine Gittersuche durchgeführt. Es werden die Werte $\kappa = 0,5 \cdot k$ für $k = 1, \ldots 10$ untersucht.

2.4.2. Dekompositionsheuristik

Zur Verbesserung des TWT-Wertes wird für jede einzelne Maschine die *Dekompositionsheuristik* (DH) angewendet. Diese Heuristik wurde von Mehta und Uzsoy [18] für eine einzelne Maschine eingeführt. Hier ist diese Heuristik auf mehrere parallele Maschinen übertragen, so dass die Heuristik für jede der verschiedenen Maschinen einzeln betrachtet wird. Im Algorithmus 2.3 ist das Verfahren je Maschine beschrieben.

Der im Rahmen de *ATC-BATC*-Heuristik gebildete Ablaufplan wird in Ablaufpläne je Maschine aufgeteilt. Für jede Maschine wird die nach aufsteigendem Fertigstellungstermin sortierte Sequenz der Batches durch *DH* bearbeitet. Dazu wird der Ablaufplan in Teilsequenzen zerlegt und für die Teilsequenzen die optimale Sequenz mit dem kleinsten *TWT*-Wert ermittelt.

Algorithmus 2.3 : Dekompositionsheuristik

Data : L_{Batch} Ablaufplan;
λ, α Parameter der Dekompositionsheuristik
Result : L_{Batch} Ablaufplan optimiert;

```
1 start := 1;
2 repeat
3 |   ende := max(start + λ − 1, |L_Batch|);
4 |   finde beste Reihenfolge L_Batch[start, ende];
5 |   start := min(start + α, |L_Batch|);
6 until (start = |L_Batch|);
```

Zuerst wird der Startwert *start* für die Sequenz wird auf 1 gesetzt (Schritt 01). Es wird solange eine Teilsequenz überarbeitet, bis der Startwert für die nächste Sequenz ganz am Ende des Ablaufplanes ist (Schritte 02 - 06). Der Endwert *ende* für die Sequenz wird gesetzt (Schritt 03). Es werden (maximal) λ Batches überprüft. Im Schritt 04 wird die Teilsequenz vom Batch an der Stelle *start* bis zum Batch an der Stelle *ende* überprüft. Hierzu werden alle Permutationen der Batches überprüft. Das bedeutet, dass alle möglichen Reihenfolgen untersucht werden, also die Fertigstellungstermine der Batches vertauscht. Mit einem veränderten Fertigstellungstermin für den Batch ändert sich auch die Fertigstellungstermine der Jobs im Batch. Somit ändert sich auch der *TWT*-Wert der Jobs und damit auch der *TWT*-Wert des Batches als Summe der *TWT*-Werte der Jobs im Batch. Die Summe der *TWT*-Werte der Batches in der Sequenz ist der *TWT*-Wert der Permutation. Die Sequenz mit dem kleinsten *TWT*-Wert wird dann ausgewählt.

Der Startwert *start* wird im Schritt 05 um (maximal) α Positionen nach vorne geschoben. Das bedeutet, dass die Position der ersten (maximal) $start + \alpha - 1$ Batches für den Ablaufplan fixiert werden.

Mit diesem Verfahren wird Schritt für Schritt überprüft, ob der Ablaufplan verbessert werden kann. Es werden jeweils Teilsequenzen der Länge λ optimiert. Nach der Optimierung werden die ersten α Batches der Teilsequenz fixiert. Somit werden bei jeder Teilsequenz $\lambda!$ Permutationen untersucht.

Wenn es bei einem Durchlauf Veränderungen der Reihenfolge der Batches gibt, wird das Verfahren nochmals durchgeführt. Hierbei werden maximal *iter* Durchläufe durchgeführt. Die Heuristik hat somit drei Parameter, λ, die Länge der Teilsequenz die untersucht werden, α, die Anzahl der Batches, die jeweils fixiert werden und *iter*, die maximale Anzahl der Durchläufe der Heuristik. Die Heuristik wird daher auch mit $DH(\lambda, \alpha, iter)$ bezeichnet.

Je größer λ gewählt wird, desto besser ist das Ergebnis, aber desto höher ist auch der Rechenaufwand, da alle $\lambda!$ Permutationen zu untersuchen sind. Die Dekompositionsheuristik vertauscht somit Batches auf den einzelnen Maschinen, belässt jedoch die einzelnen Batches auf der Maschine, auf der sie eingeplant sind. Auch die Jobs bleiben in den jeweiligen Batches, in denen sie eingeplant sind. Vertauscht wird bei Bedarf die Reihenfolge der Batches auf den Maschinen.

In Anlehnung an Balasubramanian *et al.* [2] und Almeder und Mönch [1] wird für die Erstellung der Referenzwerte $\lambda = 5$, $\alpha = 2$ und $iter = 15$ gewählt.

2.4.3. Swap-Heuristik

Zur Verbesserung des Ablaufplanes wird ein dritter Schritt, eine dritte Heuristik angewendet. Von Devpura *et al.* [7] wurde eine Vertauschungstechnik für Ein-Maschinen-Probleme entwickelt, die von Balasubramanian *et al.* [2] auf ein Mehr-Maschinen-Problem übertragen wurde. Diese Heuristik wird *Swapping* genannt wird und im weiteren Verlauf kurz mit Swap oder Swap-Heuristik bezeichnet. Bei Swap bleiben die Positionen der Batches fixiert. Es werden jedoch Jobs zwischen Batches derselben Familie getauscht. Das Tauschen kann und wird hierbei auch maschinenübergreifend durchgeführt. Da Jobs derselben Familie die selbe Bearbeitungszeit haben, ist ein Tausch durchführbar, ohne dass sich an der Position oder der Fertigstellungstermine der Batches etwas verändert. Im Algorithmus 2.4 ist das Verfahren für eine Familie beschrieben.

Algorithmus 2.4 : Swap-Heuristik

Data : L_{Batch} Liste der Batches im Ablaufplan für eine Familie ;
Result : L_{Batch} Liste der Batches im Ablaufplan für eine Familie optimiert;

```
1 sortiere L_Batch nicht aufsteigend nach dem Fertigstellungstermin;
2 foreach (Batch in L_Batch) do
3 |   überprüfeBatch(Batch);
4 end foreach
```

Bei der Swap-Heuristik, werden zuerst die Batches einer Familie nicht aufsteigend nach der Fertigstellungstermin sortiert. Es werden dabei alle Batches über alle Maschinen hinweg berücksichtigt. Es wird jeder Batch des Ablaufplanes vom ersten Batch beginnend bearbeitet. Der gewählte Batch, genauer genommen die Jobs in diesem Batch, werden überprüft (Schritt 03). Das Überprüfen wird im Algorithmus 2.5 detaillierter beschrieben.

Beim Teil überprüfe Batch der Swap Heuristik wird zuerst der Schalter *changed*, der sich merkt ob es Veränderungen im Ablaufplan gab, wird auf *true* gestellt. Damit wird die nachfolgende Schleife bearbeitet.

Diese Schleife (Schritte 02 - 13) wird solange durchgeführt, bis keine Veränderung mehr für den Batch durchgeführt wurde.

Algorithmus 2.5 : Swap-Heuristik, überprüfe Batch

Data : L_{Batch} Liste der Batches im Ablaufplan für eine Familie ;
$Batch1$ Batch der geprüft wird
Result : L_{Batch} Liste der Batches im Ablaufplan für eine Familie, aktualisiert;

```
1  changed := true;
2  while (changed) do
3      changed := false;
4      j := erster Job im Batch Batch1;
5      while ((noch nicht alle Jobs im Batch bearbeitet) und (!changed)) do
6          Batch2 := letzter Batch in L_Batch;
7          while ((Batch1 ungleich Batch2) und (!changed)) do
8              changed := überprüfe Job (j, Batch2);
9              if (!changed) then Batch2 := vorheriger Batch in L_Batch vor Batch2;
10         end while
11         j := nächster Job im Batch;
12     end while
13 end while
```

Am Schleifenanfang (Schritt 03) wird der Schalter *changed* auf *false* gestellt, da in diesem Schleifendurchlauf noch keine Veränderung durchgeführt wurde. Es wird dann der erste Job des Batches ausgewählt (Schritt 04).

In der nächsten Schleife (Schritte 05 - 12) werden alle Jobs des Batches von vorne bis hinten durchgearbeitet, bis alle Jobs überprüft wurden. Die Schleife wird abgebrochen, sobald eine Veränderung erkannt wurde.

Im Schritt 06 wird die Variable $Batch2$ auf den letzten Batch der Ablaufplanung gestellt,denn jetzt werden die Batches von hinten nach vorne durchgearbeitet.

In der nächsten Schleife (Schritte 07 - 10) werden die Batches von hinten nach vorne durchgearbeitet, bis der $Batch1$ erreicht ist, oder es eine Veränderung gab. Es wird überprüft (Schritt 08), ob der Job j mit einem Job des Batches $Batch2$, getauscht werden kann. Es kann getauscht werden, wenn es eine Verbesserung des TWT-Wertes des Ablaufplans ergibt. Dies ist im Algorithmus 2.6 dargestellt. Wenn bisher keine Veränderung durchgeführt wurde, wird der nächste Batch, das bedeutet der Batch vor $Batch2$ im Ablaufplan zur Überprüfung herangezogen (Schritt 09).

Es wird der nächste Job im Batch $Batch1$ zur Überprüfung herangezogen (Schritt 11).

Im Nachfolgenden wird dargestellt, wie überprüft wird, ob der Job j mit einem Job aus dem Batch $Batch2$ getauscht werden kann. Dies wird im Algorithmus 2.6 dargestellt.

Beim Algorithmus überprüfe Job in der Swap Heuristik wird zuerst ein Schalter *changed* auf *false* gestellt (Schritt 01). Das bedeutet, dass kein Tausch stattge-

Algorithmus 2.6 : Swap-Heuristik, überprüfe Job

Data : *j* Job, der überprüft wird ;
Batch Batch, mit dessen Jobs der Job *j* verglichen wird
Result : *true*, wenn ein Tausch durchgeführt wird, *false*, wenn kein Tausch durchgeführt wurde

```
1 changed := false;
2 i := erster Job im Batch Batch;
3 while ((noch nicht alle Jobs im Batch bearbeitet) und (!changed)) do
4 |  changed := vergleich(j,i);
5 |  if (changed) then  vertausche (j, i) ;
6 |  ;
7 |  i := nächster Job im Batch Batch;
8 end while
```

funden hat. Dann (Schritt 02) wird der Job *j*2 auf den ersten Job im Batch *Batch* gesetzt.

In der Schleife (Schritte 03 - 07) werden die Jobs im Batch *Batch* betrachtet, bis ein Job gefunden wird, der mit dem Job *j* vertauscht wird oder bis alle Jobs im Batch *Batch* überprüft wurden. Es werden die beiden Jobs *j* und *i* verglichen (Schritt 04). Ein Tausch von zwei Jobs aus zwei verschiedenen Batches wird durchgeführt, wenn sich bezüglich der Verspätung eine Verbesserung einstellt, wenn somit die Verspätung nach dem Tausch geringer ist als vor dem Tausch. Für die Jobs *i* und *j* seien gegeben die Gewichte w_i und w_j, die gewünschten Fertigstellungstermine d_i und d_j und die Fertigstellungstermine C_i und C_j. Der Fertigstellungstermine sind die Fertigstellungstermine der Batches, in denen die Jobs *i* beziehungsweise *j* enthalten sind. Aus diesen beiden Jobs ergibt sich die aktuelle, gewichtete Verspätung T_{akt} für diese beiden Jobs durch

$$T_{act} = w_i \cdot (C_i - d_i)^+ + w_j \cdot (C_j - d_j)^+ \ . \tag{2.4}$$

Werden die beiden Jobs getauscht, dann tauschen die beiden Jobs die Fertigstellungstermine C_i und C_j. Für die dann erzielte gewichtete Verspätung nach dem Tausch T_{chg} ist gegeben durch

$$T_{chg} = w_i \cdot (C_j - d_i)^+ + w_j \cdot (C_i - d_j)^+ \ . \tag{2.5}$$

Gilt nun $T_{chg} < T_{act}$, dann wird der Tausch durchgeführt, da es eine Verbesserung für die gesamte gewichtete Verspätung gibt, ansonsten wird keine Veränderung durchgeführt. Dies bedeutet insbesondere, dass kein Tausch durchgeführt wird, wenn aktuell keine der beiden Jobs verspätet sind. Wenn die Überprüfung ergab, dass die beiden Jobs *i* und *j* getauscht werden sollen, dann wird der Tausch vorgenommen (Schritt 05). Die beiden Jobs tauschen Ihren Platz in den Batches. Wenn ein Tausch durchgeführt wird, dann wird die Überprüfung des Batches wieder beim ersten Job des Batches begonnen. Ansonsten (Schritt 06) wird zum nächsten Job (im Batch *Batch*2) gegangen.

Dieses Verfahren wird für jede vorkommende Familie der Probleminstanz durchgeführt. Die Batches einer Familie werden hierbei in einer nicht absteigenden Folge betrachtet. Vom ersten Batch der Folge bis zum letzten Batch der Folge wird geprüft, ob es einen Job des zu untersuchenden Batches gibt, der mit einem Job aus einem nachfolgenden Batch der Folge getauscht werden kann. Hierzu werden die Jobs des zu untersuchenden Batches vom ersten bis zum letzten Job untersucht. Für den zu untersuchenden Job werden, wird mit dem ersten nachfolgenden Batch geprüft, ob einer der Jobs aus diesem nachfolgenden Batch, mit dem zu untersuchenden Job getauscht werden muss, ob also $T_{chg} < T_{act}$ (gemäß den Formeln (2.4) und (2.5)) gilt. Wird ein Tausch durchgeführt, dann wird die Untersuchung wieder mit dem ersten Job des zu untersuchenden Jobs durchgeführt. Hat kein Tausch stattgefunden, dann wird der nächste Job im nachfolgenden Batch geprüft. Sind alle Jobs im nachfolgenden Batch geprüft, dann wird der nächste Batch in der Folge der Batches geprüft. Wenn bis zur Prüfung des letzten Jobs im letzten Batch kein Tausch stattgefunden hat, dann wird der nächste Job im zu untersuchenden Batch geprüft. Sind alle Jobs im zu untersuchenden Batch bearbeitet, dann wird der nächste Batch in der Folge der Batches untersucht. Dies wird durchgeführt, bis alle Batches untersucht sind.

Kapitel 3.

Genetische Algorithmen mit Random-Key-Repräsentation für das Batchproblem

In diesem Kapitel werden Basisprinzipien genetischer Algorithmen beschrieben. Zuerst (siehe Abschnitt 3.1) wird die generelle Funktionsweise genetischer Algorithmen beschrieben. Dann (Abschnitt 3.2) wird die Funktionsweise von genetischen Algorithmen beschrieben, die auf Random-Keys basieren ($RKGA$). Neben den problemunabhängigen Teilen ist ein wichtiger Teil der Dekoder, welcher eine problemabhängige Umsetzung der Random-Keys in eine Lösung des Problems erzeugt. Im Abschnitt 3.3 werden verschiedene Dekoder beschrieben, die im Rahmen der Arbeit entwickelt wurden.

3.1. Funktionsweise genetischer Algorithmen

Genetische Algorithmen wurden in den 1960er und 1970er Jahren von John Holland entwickelt. Er veröffentlichte auch eine der ersten Monographien zum Thema genetische Algorithmen im Jahr 1975 (siehe Holland [13], siehe dazu auch Goldberg [9] und Michalewicz [19]). Ziel ist es, die aus der Biologie bekannten Mechanismen der Evolution auf die Lösung von Problemen mittels Suchalgorithmen zu übertragen.

3.1.1. Verwendete Begriffe

Zuerst werden einige Begriffe erläutert, die aus der Biologie adaptiert werden, siehe dazu auch Gerdes *et al.* [8].

Ein **Chromosom** ist ein Individuum, welches die vollständige Kodierung für eine Lösung des Problems beinhaltet. Ein Chromosom besteht aus einzelnen Teilen, wobei jedes einzelne Teil **Gen** genannt wird. Eine konkrete Ausprägung eines Gens heißt **Allel**. Mit **Genotyp** wird die formale Kodierung der Lösung des Problems genannt, während die dekodierte Lösung als **Phänotyp** bezeichnet wird.

Im Nachfolgenden werden die Sachverhalte anhand eines einfachen Beispiels erläutert. Auf einer Maschine soll ein Ablaufplan für $n = 4$ Jobs gebildet werden. Die Zielfunktion sei γ, wobei es auf die konkrete Ausprägung für die Erläuterung nicht ankommt. Somit wird das Problem mit $1||\gamma$ klassifiziert.

Ein Chromosom stellt eine Sequenz der Reihenfolge, in der die Jobs auf der Maschine bearbeitet werden, dar. Für $n = 4$ ist durch die Sequenz $(1, 3, 2, 4)$ ein Ablaufplan gegeben. In dieser Zuordnung wird den Jobs die Position im Ablaufplan zugeordnet. Job 1 ist an Position 1, Job 2 ist an Position 3, Job 3 an Position 2 und Job 4 an Position 4. Das erste Gen hat den Wert 1, also das Allel 1, das zweite Gen hat das Allel 3 und so weiter. Der Genotyp ist die Sequenz $(1, 3, 2, 4)$, der Phänotyp ist der Ablaufplan Job 1, Job 3, Job 2 und dann Job 4.

Bei einem GA wird eine **Population**, das heißt eine Menge von Chromosomen, betrachtet. Eine **Generation** ist eine Population zu einem festen Zeitpunkt. Es existiert eine Fitnessfunktion φ. Die Fitnessfunktion φ berechnet die Fitness für ein Chromosom. Die Fitness drückt aus, wie geeignet das Individuum ist, um in die nächste Generation aufgenommen zu werden. Der Fitnesswert wird durch eine lineare Skalierung des Zielwertes $\varphi := af + b$ berechnet (siehe dazu Goldberg [9]). Mit dieser linearen Skalierung wird erreicht, dass die Fitnesswerte nicht so nahe beieinander sind. Gemäß Goldberg werden die Parameter a und b so gewählt, dass der mittlere Zielfunktionswert einer Generation auf sich abgebildet wird und der beste Zielfunktionswert auf ein vorgegebenes Vielfaches des mittleren Zielfunktionswertes abgebildet wird.

Für den Übergang von einer Generation zur nächsten Generation gibt es verschiedene Möglichkeiten für das Erzeugen eines neuen Individuums (Chromosom) aus einem oder mehreren Individuen der aktuellen Generation.

Die einfachste Möglichkeit die **Reproduktion** ist das Kopieren eines Individuums von einer Generation zur nächsten Generation ohne jegliche Veränderung. Dies wird insbesondere für Chromosomen durchgeführt, die einen guten Fitnesswert besitzen.

Bei einem **Crossover** werden aus zwei Individuen einer Generation, den sogenannten Eltern, ein oder mehrere neue Individuen für die nächste Generation erzeugt. Für das Crossover gibt es verschiedene Möglichkeiten. So können die Chromosome der beiden Elternteile an einer Stelle geteilt werden und über Kreuz wieder zusammen gesetzt werden. Dies ist ein **one-point crossover**. Hierbei kann ein oder zwei neue Kinderteile entstehen. Eine andere Möglichkeit ist es, per Zufall zu bestimmen, welches Gen von welchem Elternteil auf das Kind übertragen wird. Dies ist ein **uniform crossover**. Für die Beschreibung verschiedener crossover-Methoden siehe [20].

Bei einer **Mutation** werden einzelne Gene in einem Chromosom verändert und somit ein neues Chromosom erstellt.

Im Unterabschnitt 3.1.3 wird dargestellt, welche Probleme es mit diesen Operationen beim obigen Beispiel, dem Ablaufplan für $n = 4$ Jobs geben kann.

3.1.2. Kanonischer genetischer Algorithmus

Im Algorithmus 3.1 ist ein kanonischer Genetischer Algorithmus aufgeführt, angelehnt an Gerdes *et al.* [8].

Algorithmus 3.1 : Kanonischer GA

Data : Probleminstanz;
Result : Lösung des Problems mit guter Fitness;
1 setze die Parameter p, p_c und p_m;
2 setze t := 0;
3 erzeuge Generation Gen_0;
4 **repeat**
5 | bestimme Fitness der Generation Gen_t;
6 | setze $Gen_{t+1} := \emptyset$;
7 | führe Reproduktion durch;
8 | führe Crossover durch;
9 | führe Mutation durch;
10 | setze $t := t + 1$;
11 **until** *(Abbruchkriterium erreicht)*;

Im Schritt 01 werden drei Parameter für den Algorithmus gesetzt. Jede Generation besteht aus p Individuen. Der Parameter p_c ist die Crossoverwahrscheinlichkeit. Er wird weiter unten, bei der Erläuterung des Crossoveroperators, genauer diskutiert. Der Parameter p_m ist die Mutationswahrscheinlichkeit. Dieser Parameter wird auch weiter unten bei der Beschreibung der Mutation erläutert. Der Parameter t, der im Schritt 02 gesetzt wird, ist der Zähler für die Generationen. Im Schritt 03 wird die Startgeneration Gen_0 erzeugt. Es sind dabei p Lösung zu erzeugen, da jede Generation aus p Individuen besteht. Damit sind die Initialisierungen abgeschlossen.

Die Schleife zwischen den Schritten 04 und 10 dient der Erzeugung der nächsten Generation Gen_{t+1} aus der Generation Gen_t. Zuerst (Schritt 05) werden die Fitnesswerte der einzelnen Individuen bestimmt. Dann (Schritt 06) wird zunächst die Nachfolgegeneration Gen_{t+1} initialisiert. Im Schritt 07 werden die Operationen für das Crossover durchgeführt, bis die neue Generation ausreichend Individuen enthält. Dieser Teil ist im Algorithmus 3.2 aufgelistet und im Nachfolgenden beschrieben. Im Schritt 08 werden Mutationen durchgeführt, was in Tabelle 3.3 aufgelistet und im Nachfolgenden beschrieben wird. Als Abbruchkriterium (Schritt 10) kann die Anzahl der Generationen oder auch ein Zeitlimit gewählt werden. Auch weitere Abbruchkriterien sind möglich. Im Rahmen dieser Arbeit wurde als Abbruchkriterium entweder die Laufzeit oder die Anzahl der Generationen verwendet.

Im Algorithmus 3.2 ist die Durchführung der Crossover-Operationen aufgeführt. Es ist eine Schleife, die durchgeführt wird, bis so viele neue Chromosomen erzeugt wurden, wie gewünscht.

Algorithmus 3.2 : Crossover

```
   Data : Gen_t;
   Result : Gen_{t+1};
 1 repeat
 2 |  wähle zufällig Individuum e_1 aus Gen_t;
 3 |  erzeuge Zufallszahl z ~ U[0,1];
 4 |  if (z < p_c) then
 5 |  |  wähle zufällig Individuum e_2 aus Gen_t;
 6 |  |  erzeuge zwei Kinder k_1, k_2 aus den beiden Elternteilen;
 7 |  |  setze Gen_{t+1} = Gen_{t+1} ∪ {k_1, k_2};
 8 |  else
 9 |  |  setze Gen_{t+1} = Gen_{t+1} ∪ {e_1};
10 |  end if
11 until (|Gen_{t+1}| ≥ p);
```

Im Schritt 02 wird ein Individuum aus der Population der Generation Gen_t per Zufall ausgewählt. Es ist im Folgenden das Elternteil 1 e_1. Die Wahrscheinlichkeit $p(i)$ für die Wahl eines Individuums i ist dabei gegeben durch

$$p(i) = \frac{fitness(i)}{\sum_{c \in Gen_t} fitness(c)} , \tag{3.1}$$

wobei $fitness(i)$ der Fitnesswert des Individuums i ist. Die Wahrscheinlichkeit für die Wahl von Individuum i ist dabei proportional zur Fitness des Individuums. Diese Art der Selektion ist eine fitness-proportionale Selektion, die auch *Roulettrad-Selektion* genannt wird.

Im Schritt 03 wird eine Zufallszahl $z \sim U[0,1]$ generiert. Die Zufallszahl z ist somit eine Realisierung einer auf dem Intervall $[0,1]$ gleichverteilten Zufallsvariablen. Ist die Zufallszahl $z < p_c$ (Prüfung in Schritt 04), dann wird ein Crossover durchgeführt (Schritte 05 - 07). Es wird ein zweites Elternteil ausgewählt (Schritt 05), wobei für die Wahrscheinlichkeit für die Wahl vom zweiten Elternteil die selben Regeln gelten wie bei der Wahl vom Elternteil 1. Es ist auch möglich, dass das Elternteil 2 gleich dem Elternteil 1 ist. Im Schritt 06 werden aus den beiden Elternteilen dann zwei Kindindividuen erzeugt, die dann (Schritt 07) zur nächsten Generation hinzugefügt werden. Wird kein Crossover durchgeführt, dann wird das Elternteil e_1 in die nächste Generation kopiert (Schritt 09).

Dieses Crossover wird solange durchgeführt, bis die Populationsgröße der nachfolgenden Generation erreicht ist, bis also ausreichend neue Chromosome generiert wurden..

Die Durchführung der Mutation ist im Algorithmus 3.3 beschrieben.

Es wird jedes Individuum betrachtet. Über eine Zufallszahl $z \sim U[0,1]$ wird bestimmt, ob eine Mutation durchgeführt wird ($z < p_m$) oder nicht.

Algorithmus 3.3 : führe Mutation durch

Data : Gen_t;
Result : Gen_t mit durch Mutation modifizierte Chromosome ;

```
1 foreach (Individuum i in Gen_t) do
2 |  erzeuge Zufallszahl z ~ U[0,1];
3 |  if (z < p_m) then
4 |  |  wende Mutationsoperator auf Individuum i an;
5 |  end if
6 end foreach
```

3.1.3. Probleme mit dem kanonischen genetischen Algorithmus

Für genetische Algorithmen ist es wichtig, dass bei der Veränderung oder der Neugestaltung eines Chromosoms aus dem neuen Chromosom wieder eine Lösung des Problems gebildet werden kann. Dies gilt sowohl bei den neuen Chromosomen, die durch Crossover erzeugt werden, als auch aus den Chromosomen, die aus einer Mutation entstehen.

Beim oben eingeführten Beispiel der Ablaufplanung mit der Charakteristik $1||\gamma$ und $n = 4$ Jobs ist ein Chromosom der Repräsentant für einen Ablaufplan. Ein Chromosom ist dabei eine Sequenz von Zahlen. Die Zahl an der Position i der Sequenz repräsentiert die Position im Ablaufplan, an welcher der Job i steht. So repräsentiert die Sequenz $(1, 2, 3, 4)$ dem Ablaufplan mit der Reihenfolge Job 1, Job 2, Job 3 dann Job 4. Eine Mutation kann bedeuten, dass in der Sequenz an der zweiten Stelle eine 3 steht und nicht mehr eine 2. Die neue Sequenz $(1, 3, 3, 4)$ repräsentiert jedoch keine zulässige Lösung der Ablaufplanung, da hier sowohl Job 2 als auch Job 3 an der dritten Stellen stehen soll.

Seien die beiden zulässigen Sequenzen $(1, 2, 3, 4)$ und $(1, 3, 2, 4)$ gegeben. Die beiden Sequenzen können jeweils in der Mitte geteilt und dann über Kreuz wieder zusammen gesetzt werden. Dabei entstehen die beiden Sequenzen $(1, 2, 2, 4)$ und $(1, 3, 3, 4)$. Beide neu entstandenen Sequenzen repräsentieren keine zulässige Lösung des Ablaufplanungsproblems dar.

Hier müssen somit problemabhängig Operationen verwendet werden, die einen Reperaturmechanismus enthalten, um wieder zulässige Lösungen für das Problem zu erzeugen. Somit müssen die Operationen, für jeden Problemfall individuell angepasst werden. In Michalewicz [19] wird ausführlich die Problematik beim *Travelling Salesman Problem* (*TSP*) diskutiert und problemabhängige Operationen zur Lösung der Problematik vorgestellt.

Die Einführung von Repräsentationen von Lösungen, die auf Random-Keys basieren, führt zu Vereinfachungen. Dies wurde von Bean [3] eingeführt und wird im nächsten Abschnitt 3.2 dargestellt. Mit dieser Veränderungen können die Operationen problemunabhängig gestaltet werden.

3.2. Genetische Algorithmen mit Random-Keys

3.2.1. Random-Keys

Bei genetischen Algorithmen mit *Random-Keys* wird eine Lösung des Problems mit Hilfe der Folge von Random-Keys erstellt. Dies wird von Bean [3] vorgeschlagen.

Die Folgen von Random-Keys sind typischerweise Elemente aus $S^n = [0,1]^n$. Für das im vorherigen Abschnitt aufgeführte Beispiel der Ablaufplanung auf einer Maschine ist $n = 4$. Eine Folge von Random-Keys kann dann beispielsweise $(0,51,\ 0,24,\ 0,75,\ 0,68)$ sein. Aus der sortierten Folge der Random-Keys in nichtabfallender Reihenfolge kann ein Ablaufplan decodiert werden. Aus der Folge der Random-Keys ergibt sich die Sequenz $(2,1,4,3)$ und die entsprechende Reihenfolge der Jobs: Job 2, Job 1, Job 4 und Job 3. Mit einer solchen Repräsentation von Lösungen mit Hilfe von Random-Keys sind die im vorherigen Abschnitt aufgeführten problematischen Operationen durchführbar.

Wird durch eine Mutation in der Folge der Random-Keys an der zweiten Stelle die $0,24$ durch $0,69$ ersetzt, dann ist $(0,51,\ 0,69,\ 0,75,\ 0,68)$ die neue Folge der Random-Keys, die zur Folge $(1,4,2,3)$ für die Jobs führt.

Ein Crossover zweier Folgen von Random-Keys ist ebenso wieder eine Folge von Random-Keys, so dass auch hier eine zulässige Lösung erstellt werden kann. Sind $(0,51,\ 0,69,\ 0,75,\ 0,68)$ und $(0,43,\ 0,91,\ 0,88,\ 0,90)$ zwei Folgen von Random-Keys mit den dazugehörigen Ablaufplänen $(1,4,2,3)$ und $(1,3,4,2)$, dann können beide Folgen von Random-Keys in der Mitte geteilt werden. Durch vertauschen der hinteren Teile entstehen zwei neue Folgen von Random-Keys. Diese lauten $(0,51,\ 0,69,\ 0,88,\ 0,90)$ und $(0,43,\ 0,91,\ 0,75,\ 0,68)$. Dies führt zu den beiden zulässigen Ablaufplänen $(1,2,3,4)$ und $(1,4,3,2)$.

Somit werden die Folgen von Random-Keys so interpretiert (dekodiert), dass keine unzulässige Lösungen entstehen. Die Suche nach der besten Lösung erfolgt somit in der Menge der Folgen aus Random-Keys.

3.2.2. Genetischer Algorithmus auf Basis von Random-Keys

Auf Basis der Folgen von Random-Keys hat Bean [3] einen genetischen Algorithmus entwickelt. Eine Lösung verändert sich in einer zeitlichen Abfolge durch Vererbung von Merkmalen zu besseren Lösungen hin.

Beim Übergang von einer Generation zu einer neuen Generation, werden aus den Individuen der alten Generation durch verschiedene Operationen Individuen der neuen Generation erzeugt. Hierbei werden drei Operationen angewendet:

- Kopieren (*reproduction*),
- Crossover (*crossover*) und

- Migration (*migration*).

Kopieren

Die Population wird nach den Fitness-Werten der Individuen sortiert. Das fitteste Individuum, also das Individuum mit dem besten Fitness-Wert, steht an oberster Stelle. Für den Algorithmus ist vorab ein Parameter p_e festgelegt. Der Parameter p_e gibt an, wie viele Elemente der Population zur Elite gehören. Die besten p_e Elemente einer Population sind die Elite. Der Wert p_e kann entweder als ganze Zahl abgegeben werden oder als eine reelle Zahl im Intervall $[0,1]$. Im zweiten Fall stellt es den Anteil der Elite an der Gesamtpopulation dar, die Anzahl der Elemente der Elite ist dann durch $\lfloor p \cdot p_e \rfloor$ gegeben, wobei p die Anzahl der Elemente der Population darstellt. Die Individuen, die zur Elite gehören, werden in die Population der nächsten Generation kopiert, also ohne Veränderung reproduziert. Der Vorteil hiervon ist, gemäß Bean [3], eine monotone Verbesserung der Fitness des besten Individuums. Der Nachteil ist, dass das Verfahren zu einem lokalen Optimum konvergieren kann, welches jedoch kein globales Optimum ist.

Crossover

Der von Bean [3] gewählte Crossoveroperator ist ein *uniform crossover* (siehe Unterabschnitt 3.1.1). Es werden zuerst per Zufall zwei Elternteilen aus dem gesamten Bestand der aktuellen Generation gewählt. Es kann zufällig auch das selbe Elternteil gewählt werden. Die Wahrscheinlichkeit hierfür ist bei einer Größe der Population von p gleich $1/p$. Für jedes einzelne Gen des Chromosoms, welches aus den beiden Eltern-Individuen erzeugt wird, wird per Zufalls entscheiden, welches Allel übernommen wird, das vom Chromosom 1 (Elternteil 1) oder das vom Chromosom 2 (Elternteil 2). Dazu wird ebenso vorab für den Algorithmus ein Parameter ρ_e, der Crossover-Parameter, festgelegt. Für jedes Gen wird eine gleichverteilte Zufallszahl z im Bereich $[0,1]$ erzeugt. Ist $z > \rho_e$, dann wird das Allel vom zweiten Chromosom genommen, ansonsten das Allel vom ersten Chromosom. Ist $\rho_e = 0,5$ dann entspricht dies einem Münzwurf mit einer fairen Münze, die Gene im Kindindividuum ist mit jeweils der Wahrscheinlichkeit von 50% aus jedem der beiden Elternteile. Wenn ein $\rho_e > 0.5$ gewählt wird, dann entspricht dies mit dem Wurf mit einer einseitig verfälschten (*biased*) Münze. Im Kindindividuum sind die Gene des einen Elternteils stärker vertreten als die des zweiten Elternteils.

In der Tabelle 3.1 ist für das bereits oben aufgeführte Probleminstanz für ein Ablaufplanungsproblem mit vier Jobs, ein Crossover aufgeführt. Hier ist $\rho_e = 0,7$ gesetzt.

Nur an der dritten Stelle ist die Zufallszahl größer als das ρ_e, und somit wird das Allel vom zweiten Chromosom gewählt. Bei den anderen Genen wird jeweils das Allel vom ersten Chromosome übertragen. Die beiden Elternteile führen zu den Ablaufplänen $(1,4,2,3)$ und $(1,3,4,2)$, das Kind führt zum Ablaufplan $(1,4,2,3)$.

Tabelle 3.1.: Beispiel Crossover

Chromosom 1	0,51	0,69	0,75	0,68
Chromosom 2	0,43	0,91	0,88	0,90
Zufallszahl z	0,26	0,12	0,92	0,03
$z > \rho_e$	nein	nein	ja	nein
Neues Chromosom	0,51	0,69	0,88	0,68

Der Ablaufplan vom Kind ist gleich dem Ablaufplan vom Elternteil 1, obwohl die Chromosomen unterschiedlich sind. Zwei unterschiedliche Chromosomen können somit zur selben Lösung führen.

Migration

Statt der Durchführung einer Mutation, bei der eine sehr kleine Anzahl von Chromosomen an weniger Stellen verändert werden, schlägt Bean [3] eine Migration vor. Bei einer Migration werden in jeder Generation eine bestimmte Anzahl oder ein bestimmter Anteil von Chromosomen per Zufall neu bestimmt und der Population zugefügt. Die Mutationsrate p_m ist ein Parameter, der am Anfang des Verfahren festgelegt wird. Durch neue zufällige Chromosomen soll verhindert werden, dass das Verfahren gegen ein lokales Optimum konvergiert. Durch neue zufällig gewählte Chromosome besteht die Möglichkeit, ein lokales Optimum zu verlassen, das heißt zu einem besseren Wert zu gelangen.

Zusammenfassend kann der Schritt von einer Generation zur nächsten Generation somit folgendermaßen darstellt werden, was auch in der Abbildung 3.1 (in Anlehnung an Goncalves und Resende [10]) grafisch dargestellt ist.

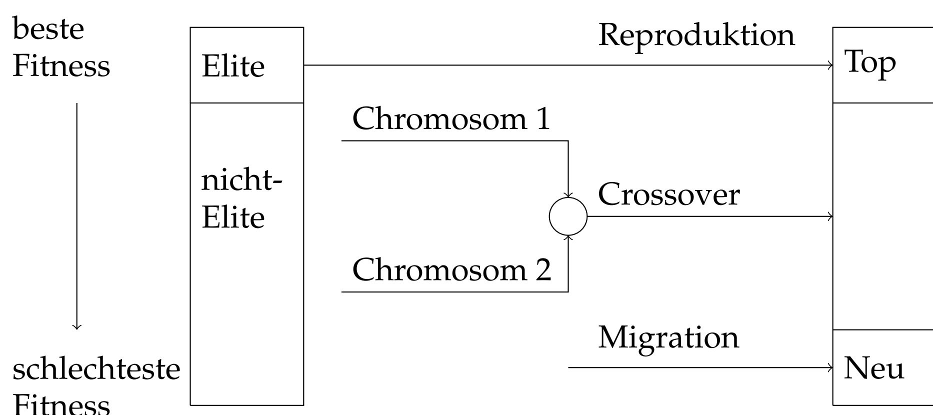

Abbildung 3.1.: Übergang von einer Generation zur nächsten

Für die p Individuen einer Generation werden die Fitnesswerte berechnet. Die Individuen werden dann nach ihrem Fitnesswert absteigend sortiert. Die besten

p_e Elemente bilden die Elite, welche unverändert in die nächste Generation kopiert werden. Es werden p_m neue Chromosome per Zufall erstellt und in die neue Generation hinzugefügt. Für die verbleibenden $p - p_e - p_m$ Plätze der neuen Generation wird jedes Mal ein Crossover mit zwei zufällig gewählten Elternteilen aus dem gesamten Bestand der aktuellen Generation durchgeführt. Hier gibt es den Crossover-Parameter ρ_e.

3.2.3. Veränderung beim Crossover

Aufbauen auf dem genetischen Algorithmus mit Random-Keys gemäß Bean [3] wurden von Goncalves und Resende [10] und dann nochmals von Goncalves *et al.* [11] Änderungen vorgenommen.

Die Veränderung betrifft nur den Teil der Auswahl, der Selektion der Elternteile für den Crossover. Bei Bean [3] werden die beiden Elternteile per Zufall aus der gesamten Menge der Population einer Generation ausgewählt. Dieser Algorithmus heißt $RKGA$.

Bei Goncalves und Resende [10] wird diese Selektion verändert. Für das erste Elternteil wird ein Individuum aus der Elite gewählt (siehe Abbildung 3.1). Für das zweite Elternteil wird ein Individuum aus der nicht-Elite gewählt. Damit hat das erste Elternteil in der Regel eine bessere Fitness als das zweite Elternteil. Vom ersten Elternteil werden, wegen $\rho_e > 0,5$, mehr Allele übernommen als vom zweiten Elternteil. Die Autoren nennen diese einen Biased Random-Key Genetic Algorithmus. Dieser Algorithmus heißt kurz $BRKGA$.

In Goncalves *et al.* [11] haben die Autoren eine weitere Variante eingeführt. Es agiert wie $RKGA$, jedoch mit dem Unterschied, dass bei den beiden per Zufall selektierten Elternteilen über den Fitnesswert bestimmt wird, welches das Elternteil 1 darstellt und welches das Elternteil 2. Das Elternteil mit dem besseren Fitnesswert wird das Elternteil 1, das andere das Elternteil 2. Da die Elternteile gemäß der Fitness sortiert werden, wird das Verfahren als geordnetes $RKGA$ bezeichnet, oder kurz als $RKGA^*$.

In Goncalves *et al.* [11] haben die Autoren auf Basis von numerischen Experimenten festgestellt, dass $BRKGA$ besser agiert als $RKGA^*$ und diese beiden besser als $RKGA$. Für das Ausgangsproblem, der Ablaufplanung auf mehreren parallelen Maschinen, das in Abschnitt 2.2 beschrieben wurde, ergibt sich die Frage, welche Verbesserungen mit $BRKGA$ erreicht werden kann, insbesondere im Vergleich zu den anderen Heuristiken, die bei Alemeder und Mönch [1] untersucht wurden.

3.2.4. $BRKGA$, generelles Vorgehen

Der im Algorithmus 3.1 beschriebene kanonische GA, kann nun angepasst werden. Diese Anpassung ist im Algorithmus 3.4 dargestellt.

Algorithmus 3.4 : BRKGA

```
  Data : Probleminstanz;
  Result : Ablaufplan ;
1 setze die Parameter p, p_e, p_m und ρ_e;
2 setze t = 0;
3 erzeuge Generation Gen_0;
4 repeat
5 |  bestimme Fitness der Individuen von Gen_t;
6 |  sortiere die Individuen nach der Fitness;
7 |  setze Gen_{t+1} = ∅;
8 |  kopiere die p_e Elite-Individuen von Gen_t nach Gen_{t+1};
9 |  erzeuge p_m neue Individuen und füge sie Gen_{t+1} hinzu;
10|  führe p − p_e − p_m Crossover-Operationen durch und füge die neuen
  |  Individuen zu Gen_{t+1} hinzu;
11|  setze t = t + 1;
12 until (Abbruchkriterium erreicht);
```

In den Schritten 08 bis 10 sind die wesentlichen Operationen hinterlegt. Im Schritt 08 die Reproduktion, im Schritt 09 die Migration und im Schritt 10 das Crossover durchgeführt. Diese Operationen sind problemunabhängig, da diese Operationen nur auf den Chromosomen stattfinden. Nur im Schritt 05 ist ein problemabhängiger Teil zu finden, da dort die Fitness für die im Chromosom kodierte Lösung berechnet wird. Das heißt, hier werden die Zufallszahlen im Chromosom in eine Lösung der konkreten Problemklasse umgesetzt und die Fitness berechnet.

In der Abbildung 3.2 ist der Ablauf, in Anlehnung an Goncalves und Resende [10], graphisch dargestellt.

In der Grafik sind die problemunabhängigen Teile (linke Seite) von dem problemabhängigen Teil (rechte Seite, oben) getrennt. Hier ist nochmals deutlich zu sehen, dass der Algorithmus zum großen Teil problemunabhängig ist. Der problemabhängige Teil ist die Bestimmung der Fitness der Lösungen. Dies wird durch Dekoder erstellt. Die in dieser Arbeit untersuchten Dekoder sind im nachfolgenden Abschnitt 3.3 beschrieben. Beim ersten Durchlauf wird der Dekoder für jedes Chromosom aufgerufen, also p mal. Ab dem zweiten Durchlauf wird der Dekoder $p - p_e$ mal für jeden Durchlauf aufgerufen, da die Elite-Individuen in die nächste Generation kopiert werden, die Fitnesswerte bleiben somit erhalten. Ist gen die Anzahl der Generationen, die durchgeführt werden, dann ist die Anzahl der Aufrufe des Dekoders durch

$$p + (p - p_e) \cdot gen \tag{3.2}$$

gegeben. Sind $p = 1.000$, $p_e = 200$ und $gen = 100$, dann ist die Anzahl der Aufrufe von Dekoder gleich 81.000. Um geringe Laufzeiten zu erzielen, ist es vorteilhaft, wenn der Zeitbedarf für einen Aufruf des Dekoders gering ist.

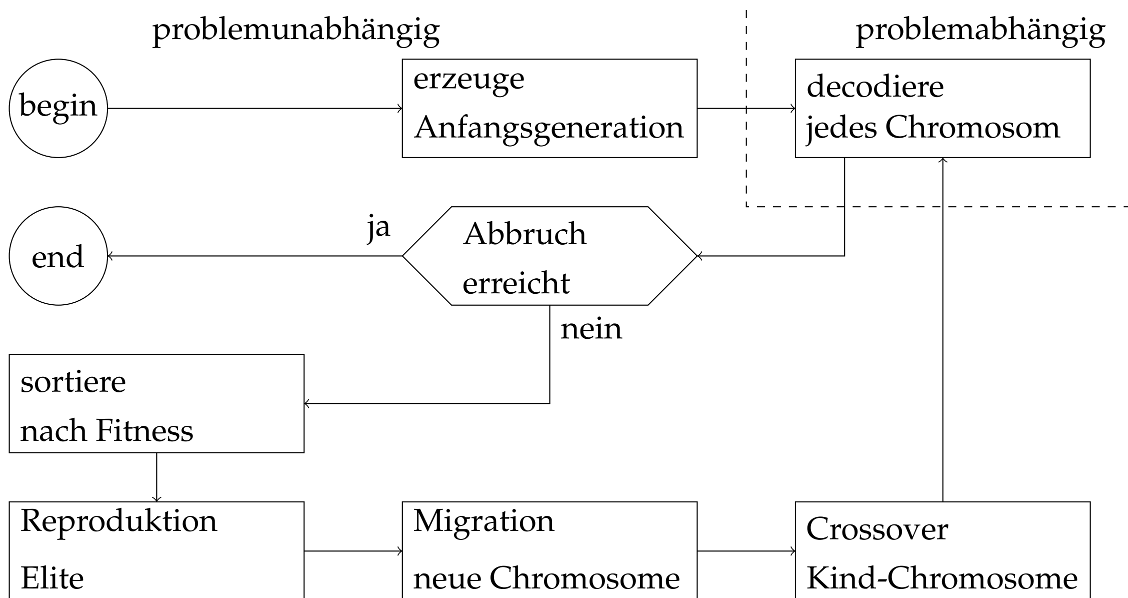

Abbildung 3.2.: Ablaufdiagram BRKGA

3.3. Entwurf konventioneller RKGA

Für einen *RKGA* ist von entscheidender Bedeutung, wie ein Chromosom, also eine Folge von Random-Keys, in eine zulässige Lösung des Problems umgesetzt wird. Dies ist der Teil vom Algorithmus, der problemabhängig ist. Diese Aufgabe übernehmen Dekoder. In dieser Arbeit werden zwei verschiedene Dekoder vorgestellt.

Eine Probleminstanz umfasst n Jobs aus f inkompatiblen Familien. Ein Ablaufplan für m parallele identische Maschinen soll erstellt werden. Die Jobs werden dabei in Batches mit einer maximalen Größe von B zusammen gefasst.

Hat die Probleminstanz n Jobs, die eingeplant werden, dann hat die Folge der Random-Keys ebenfalls n Werte. Der Folge der Jobs

$$(J_1, \ldots, J_n) \tag{3.3}$$

wird ein Chromosom

$$chr = (z_1, \ldots, z_n) \tag{3.4}$$

zugeordnet. Hierbei ist z_i der Random-Key, der dem Job J_i zugeordnet ist ($j = 1, \ldots, n$). Dann wird die Folge der Jobs sortiert, nach dem nicht-absteigenden Wert der zugehörigen Random-Keys.

In den nachfolgenden Beschreibungen werden die theoretischen Beschreibungen an Hand eines Beispiels praktisch erläutert. Es wird für beide Dekoder das selbe Beispiel verwendet.

Beispiel 3.1 (Beschreibung der Daten einer Probleminstanz). *Gegeben seien eine Probleminstanz mit $n = 8$ Jobs, die $f = 2$ inkompatiblen Familien angehören. Die Bearbeitungszeiten sind $p_0 = 4$ und $p_1 = 6$. Es sind $m = 2$ Maschinen, die maximale Batchgröße beträgt $B = 2$. In der Tabelle 3.2 ist eine derartige Probleminstanz beschrieben. Zu den Beispieldaten gehört auch eine Folge von Random-Keys.*

Tabelle 3.2.: Probleminstanz für Beispiel

Job j	0	1	2	3	4	5	6	7
Familie s	0	0	0	0	1	1	1	1
Random-Key	0,04	0,89	0,88	0,58	0,23	0,70	0,03	0,43

3.3.1. Permutationsrepräsentation

Bei der Permutationsrepräsentation bilden die Jobs eine Liste. Sobald eine Maschine frei wird, wird ein Batch gebildet, der auf diese freie Maschine platziert wird. Im Algorithmus 3.5 ist der Ablauf dargestellt, der im Nachfolgenden genauer beschrieben wird.

Algorithmus 3.5 : Dekoder Permutationsrepräsentation

Data : Probleminstanz;
Result : Ablaufplan L_{Batch} ;

1 bilde die sortierte Liste L_{Job} der noch nicht verplanten Jobs;
2 initialisiere die Liste L_{Batch} der gebildeten Batches;
3 setze $t = 0$;
4 setze alle Maschine zum Zeitpunkt $t = 0$ auf frei;
5 **repeat**
6   suche die nächste freie Maschine (i) und den Zeitpunkt (t);
7   bestimme die Familie f und die Bearbeitungszeit p des ersten Jobs in der Liste L_{Job};
8   bilde einen Batch b aus den ersten Jobs in der Liste L_{Batch}, die zur Familie f dazu gehören;
9   setze den Fertigstellungstermin des Batches b auf $t + p$;
10   entferne die Jobs des Batches b aus der Liste L_{Job};
11   setze für die Maschine i den nächsten freien Termin auf $t + p$;
12   füge den Batch b zur Liste der Batches L_{Batch} hinzu;
13 **until** *(Liste der nicht verplanten Jobs ist nicht leer)*;
14 bestimme den TWT-Wert;

Bei der Initialisierung des Algorithmus wird zuerst eine Liste L_{Job} der noch nicht eingeplanten Jobs der Probleminstanz gebildet (Schritt 01). Die Jobs in der Liste werden dabei nicht-absteigend nach dem Random-Key sortiert. Die Liste der gebildeten Batches L_{Batch} wird als leere Liste ($L_{Batch} = ()$) initialisiert (Schritt 02).

Im Schritt 03 wird der Zeitpunkt $t = 0$ initialisiert. Der Zeitzähler wird somit auf den Wert 0 gesetzt. Im letzten Schritt der Initialisierung (Schritt 04) wird die Liste der Maschinen initialisiert. Alle Maschinen sind zum Zeitpunkt $t = 0$ frei. Für $i = 0, \ldots, (m-1)$ gilt $free[i] = 0$.

In der Schleife (Schritte 05 - 13) werden solange Batches gebildet und eingeplant, bis alle Jobs eingeplant sind, bis somit die Liste L_{Job} der noch nicht eingeplanten Jobs leer ist.

Zuerst wird nächste freie Maschine (i) bestimmt und den Zeitpunkt ($t = free[i]$), an dem die Maschine frei wird (Schritt 06). Werden mehrere Maschinen zum gleichen Zeitpunkt frei, dann wird die Maschine mit dem kleinsten Index genommen. Diese selektierte Maschine ist diejenige, auf der der nächste gebildete Batch eingeplant wird. Im Schritt 07 wird die Familie s des ersten Jobs j in der Liste L_{Job} der noch nicht eingeplanten Jobs bestimmt und die Bearbeitungszeit $p_{f(j)}$ des Jobs. Die Jobs für den zu bildenden Batch gehören alle der selben Familie an, da Batches nur aus Jobs derselben Familie gebildet werden können. Die Bearbeitungszeit des Jobs stellt auch die Bearbeitungszeit des Batches dar, da alle Jobs derselben Familie die selbe Bearbeitungszeit haben und die Jobs parallel bearbeitet werden. Dann wird ein Batch b der Familie f gebildet (Schritt 08). Dazu wird in der Liste L_{Job} der noch nicht verplanten Jobs die ersten (maximal B) Jobs die zur Familie f gehören selektiert. Höchstens der letzte gebildete Batch einer Familie kann dabei nicht voll besetzt sein, also weniger als B Jobs beinhalten. Für den gebildeten Batch b wird der Fertigstellungstermin auf $t + p$ und die Maschine des Batches auf i gesetzt (Schritt 09). Die eingeplanten Jobs wird aus der Liste L_{Job} der noch nicht eingeplanten Jobs entfernt (Schritt 10). Die Bearbeitungszeit jedes Jobs j im Batch, also auch die Bearbeitungszeit des Batches, ist $p_{f(j)}$. Im Schritt 11 wird die Zeit, an der die Maschine i wieder frei wird gesetzt. Für die Maschine muss daher der Termin, wann sie wieder frei ist auf $t + p$ gestellt werden: $free[i] = t + p$. Dann wird der neu gebildete Batch b an die Liste der Batches L_{Batch} angehängt (Schritt 12). Die Liste der Batches ist damit sortiert nach der Reihenfolge, in der die Batches gebildet wurden, jedoch nicht etwa nach dem Fertigstellungstermin.

Wenn noch nicht alle Jobs eingeplant sind, wenn also die Liste L_{Job} noch nicht leer ist, dann wird die Einplanung fortgesetzt. Es wird wieder zum Schritt 06 zurückgegangen. Ansonsten sind alle Jobs eingeplant, der Ablaufplan ist erstellt und die Schleife kann verlassen werden.

Diese Repräsentation ist ähnlich der Repräsentation bei Alemder und Mönch [1] bei der *ACO*-Heuristik. Dort wird die Reihenfolge der Jobs durch die Stärke der Pheromene bestimmt und es wird je Familie ein Batch gebildet. Aus dieser Menge von Batches wird dann ein Batch ausgewählt. Hier ist die Reihenfolge der Batches durch die Random-Keys bestimmt und es wird nur ein Batch gebildet. Die Auswahl eines Batches aus einer Menge von einem Batch je Familie ist eine mögliche Veränderungsoption für den Dekoder.

Bei Bedarf kann der erstellte Ablaufplan mittels der maschinenübergreifenden Swap-Heuristik (siehe Unterabschnitt 2.4.3) verbessert werden. Da die Swap-

Heuristik aufwändig ist, ist der Algorithmus ohne die Swap-Heuristik deutlich schneller als mit Swap-Heuritik.

Der Ablaufplan ist nun fixiert. Der TWT-Wert kann berechnet werden (Schritt 14). Für jeden Job j gibt es den gewünschten Fertigstellungstermin d_j und den Fertigstellungstermin C_j, der sich aus dem Termin ergibt, an dem der Batch fertig wird, in dem der Job eingeplant ist.

Beispiel 3.2 (Dekoder Permutationsrepräsentation)**.** *Für die Probleminstanz, die im Beispiel 3.1 dargestellt wurde, ergeben sich folgende Sachverhalte. Die Liste der gemäß der Random-Keys sortierten Jobs ist*

$$L_{Job} = (6, 0, 4, 7, 3, 5, 2, 1) \ . \tag{3.5}$$

Der erste Batch der gebildet wird, besteht aus dem ersten Job der Liste, dem Job 6 *und dem nächsten Job derselben Familie in der Liste, also Job* 4*. Dieser Batch wird auf der Maschine* 0*, der ersten freien Maschine, eingeplant. Die Maschine* 0 *ist dann erst wieder zum Zeitpunkt* $t = 6$ *frei. Der gebildete Batch wird zur Liste der gebildeten Batches hinzugefügt. Die Jobs des Batches werden aus der Liste* L_{Job} *entfernt. Damit ist nun*

$$L_{Job} = (0, 7, 3, 5, 2, 1) \ . \tag{3.6}$$

Die nächste freie Maschine ist die Maschine 1*, die zum Zeitpunkt* 0 *frei ist. Es wird nun ein Batch mit dem Job* 0*, dem jetzt ersten Job der Liste, gebildet. Der zweite Job des Batches ist der Job mit der Nummer* 3*. Die Bearbeitungszeit für dieses Job ist* 4*, so dass die Maschine* 1 *dann wieder zum Zeitpunkt* $t = 4$ *frei wird. Die Jobs* 3 *und* 0 *werden aus der Liste* L_{Job} *entfernt. Als nächster Batch wird dann ein Batch mit den Jobs* 7 *und* 5 *gebildet. Der letzte Batch besteht aus dem Jobs* 2 *und* 1*. Mit jedem Schritt wird die Liste der Jobs kleiner, bis letztendlich alle Jobs in Batches eingeplant sind. Der so erhaltene Ablaufplan ist in der Abbildung 3.3 zu sehen.*

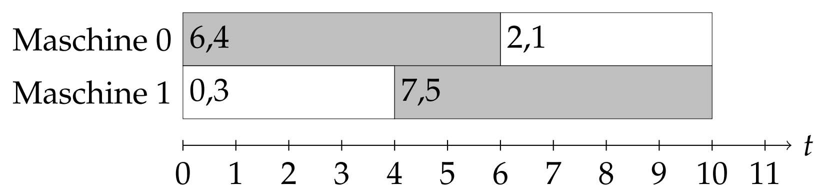

Abbildung 3.3.: Ablaufplan Beispiel Dekoder 1

Die Batches, die Jobs der Familie 0 *beinhalten, sind mit weißem Hintergrund gezeichnet, die Batches mit Jobs der Familie* 1 *sind grau hinterlegt.*

Der erste Dekoder arbeitet ähnlich wie das List-Scheduling. Zwei wesentliche Unterschiede gibt es jedoch: Zum einen bleibt die Reihenfolge der noch nicht eingeplanten Jobs unverändert und zum anderen wird ein Batch auf Basis des ersten Jobs in der Liste der noch nicht eingeplanten Jobs bestimmt.

3.3.2. Repräsentation für simultane Zuordnungen-Reihenfolge-Entscheidungen

Bei der Repräsentation für simultane Zuordnung-Reihenfolge-Entscheidungen werden die Job bereits durch die Random-Keys auf die verschiedenen Maschinen verteilt. Die generierte Zufallszahl $z \sim U[0, 1]$ wird mit der Anzahl m der Maschinen multipliziert. Die daraus resultierende Zahl z' ist dann im Intervall $[0, m]$. Die Vorkommastelle der Zahl z' repräsentiert dann die Maschine, auf welcher der Job bearbeitet werden soll. Die Nachkommastellen $z' - \lfloor z' \rfloor$ repräsentieren dann den Wert für die Reihenfolge der Job auf der Maschine für die Jobauswahl. Im Algorithmus 3.6 ist der Ablauf dargestellt, der im Nachfolgenden genauer beschrieben wird.

Algorithmus 3.6 : Dekoder simultane Entscheidungen

Data : Probleminstanz P mit n Jobs und ein Chromosom c der Länge n
Result : L_{Batch} Ablaufplan

```
1  foreach (Maschine i) do J_i := ∅;
2  foreach (Job j) do füge Job j in die passende Jobliste ein;
3  L_Batch := ∅;
4  foreach (Maschine i) do
5  |  L := ∅;
6  |  t := 0;
7  |  sortiere die Jobs j in J_i nicht-absteigend nach dem Random-Key z_j;
8  |  repeat
9  |  |  bestimme die Familie f des ersten Jobs in der Liste J_i;
10 |  |  bilde Batch b zur Familie f aus den ersten Jobs in der Liste J_i;
11 |  |  setze die Parameter für den Batch;
12 |  |  entferne die Jobs des Batches b aus der Liste J_i;
13 |  |  t := t + p;
14 |  |  L := L ∪ {b};
15 |  until (|J_i| = 0);
16 |  L_Batch := L_Batch ∪ L;
17 end foreach
18 bestimme der TWT-Wert;
```

Die Eingabedaten sind die Probleminstanz und ein Chromosom. In der Probleminstanz sind die n einzuplanenden Jobs aus f inkompatiblen Familien und m Maschinen, für die der Ablaufplan erstellt wird. Das Chromosom hat die Länge n wobei jedes Allel des Chromosoms eine Zufallszahl $z \sim U[0, 1]$ beinhaltet. Jedes Allel ist genau einem Job j zugeordnet.

Ziel des Verfahrens ist ein Ablaufplan, das heißt eine Liste von Batches mit zugehörigen Jobs, die auf die Maschinen zugeordnet sind. Darüber hinaus wird der TWT-Wert des Jobs berechnet.

Die Initialisierung beginnt, damit, dass die Liste der noch nicht eingeplanten Jobs für jede der Maschinen initialisiert wird (Schritt 01). Im Schritt 02 wird für jeden

Job j entschieden, auf welcher Maschine der Job bearbeitet werden soll. Dazu wird das Allel z_j, das dem Job zugeordnet ist, mit der Anzahl der Maschinen m multipliziert. Die Vorkommastelle dieser Zahl $i = \lfloor z_j \cdot m \rfloor$ ist die Nummer der Maschine, auf die der Job zugeordnet wird. Der Nachkommaanteil $z_j - i$ ist das neue Allel für den Job. Der Job wird dann in die Liste für die Maschine J_i der noch nicht eingeplanten Jobs hinzugefügt. Anschließend wird der Ablaufplan initialisiert (Schritt 03).

Für jede Maschine wird nun simultan der Ablaufplan für die Maschine erstellt (Schritte 04 - 17). Der Ablaufplan für die Probleminstanz ist dann die Vereinigung der Ablaufpläne der Maschinen.

Der Ablaufplan der Maschine wird initialisiert (Schritt 05). Die Bearbeitungszeit wird auf 0 gesetzt, da die Ablaufpläne zum Zeitpunkt $t = 0$ beginnen (Schritt 06). Die Jobs der Liste werden nicht-absteigend nach dem Random-Key z_j, die den Jobs zugeordnet sind, sortiert (Schritt 07). Dies ist analog zur Sortierung der Jobs nach dem ATC-Wert in der ATC-$BATC$-Heuristik. Es gibt jedoch eine wichtigen Unterschied. Bei der ATC-$BATC$-Heuristik werden die Werte für das Sortieren zu jedem Zeitpunkt, an dem eine Maschine frei wird neu bestimmt. Hier basiert die Sortierung auf den Random-Keys. Die Werte ändern sich nicht, so dass auch die Reihenfolge der Jobs im Rahmen der Durchläufe nicht verändert wird.

Für jede Maschine wird die Liste der Jobs solange bearbeitet, bis die Liste der noch nicht eingeplanten Jobs J_i abgearbeitet ist, bis also jeder Job verplant ist (Schritte 08 - 15).

Für die Bildung der Batches wird ein Vorgehen gewählt, welches von Dauzère-Pérès und Mönch [6] für eine einzelne Maschine vorgeschlagen wurde. Es wird die Familie f des ersten Jobs in der Liste bestimmt. Für diese Familie wird ein Batch bestimmt (Schritt 09). Für die Familie f wird nun ein Batch b gebildet (Schritt 10). Dabei wird die Liste der Jobs der Maschine vom Anfang her durchgearbeitet. Wenn der Job zur Familie f gehört und der Batch noch nicht voll ist, dann wird der Job zum Batch hinzugefügt. Wenn die Liste der Job durchlaufen ist, der Batch jedoch noch nicht voll ist, dann bleibt der Batch unvollständig. Im Schritt 11 werden für den Batch b nur die Parameter gesetzt, die Familie des Batches, also die Familie der Jobs, die im Batch enthalten sind, und der Fertigstellungstermin des Batches. Da alle Jobs des Batches zur selben Familie gehören und ein paralleles Batching durchgeführt wird, ist die Prozesszeit des Batches gleich der Prozesszeit jedes einzelnen Jobs, also die Prozesszeit der Familie $p(f)$. Da die Bearbeitung zum Zeitpunkt t beginnt, ist der Fertigstellungstermin für den Job gleich $t + p_f$. Anschließend werden die Jobs, die im Batch b enthalten sind, die also im Ablaufplan verplant sind, aus der Liste J_i der auf der Maschine i noch nicht verplanten Jobs entfernt (Schritt 12). Die Bearbeitungszeit t wird auf den nächsten freien Zeitpunkt $t + p_f$ gesetzt (Schritt 13). Der gebildete Batch b wird zum Ablaufplan L der Maschine hinzugefügt (Schritt 14).

Für die Maschine, die gerade bearbeitet wird, ist somit der Ablaufplan erstellt. Der Ablaufplan der Maschine wird zum Ablaufplan für die gesamte Probleminstanz hinzugefügt (Schritt 16).

Bei Bedarf kann der erstellte Ablaufplan mittels der maschinenübergreifenden Swap-Heuristik (siehe Unterabschnitt 2.4.3) verbessert werden. Da die Swap-Heuristik aufwändig ist, ist der Algorithmus ohne die Swap-Heuristik deutlich schneller als mit Swap-Heuritik.

Der Ablaufplan ist nun fixiert. Der *TWT*-Wert kann berechnet werden (Schritt 18).

Beispiel 3.3 (Dekoder Repräsentation für simultane Zuordnungen-Reihenfolge-Entscheidungen)**.** *Für die Probleminstanz, die im Beispiel 3.1 dargestellt wurde, ergeben sich folgende Sachverhalte. Die Werte der Allele werden zuerst mit der Anzahl der Maschinen $m = 2$ multipliziert. Die Werte sind in der Tabelle 3.3 aufgeführt.*

Tabelle 3.3.: Modifiziertes Chromosom

Job j	0	1	2	3	4	5	6	7
Random-Key z_j	0,04	0,89	0,88	0,58	0,23	0,70	0,03	0,43
$z_j \cdot m$	0,08	1,78	1,76	1,16	0,46	1,40	0,06	0,86
Maschine	0	1	1	1	0	1	0	0
modifiziertes z_j	0,08	0,78	0,76	0,16	0,46	0,40	0,06	0,86

Somit sind die Jobs der Maschinen 0 und 1, sortiert gemäß dem modifizierten Random-Key,

$$J_0 = (6, 0, 4, 7) \, , \tag{3.7}$$
$$J_1 = (3, 5, 2, 1) \, . \tag{3.8}$$

Der erste Job, der auf der Maschine $i = 0$ gebildet wird, beinhaltet den Job $j = 6$. Da der Job $j = 6$ zur Familie $s = 1$ gehört, ist der zweite Job im ersten Batch der Job $j = 4$. Der zweite Batch auf der Maschine $i = 0$ besteht nur aus dem Job $j = 0$. Dieser Batch ist nicht vollständig, ebenso wie der dritte Batch auf dieser Maschine, der nur aus dem Job $j = 7$ besteht.

In der Abbildung 3.4 ist der Ablaufplan, der mit Hilfe des Dekoders für die Repräsentation für simultane Zuordnungen-Reihenfolge-Entscheidungen (im Nachfolgenden manchmal kurz Dekoder 2 genannt) entsteht, grafisch dargestellt.

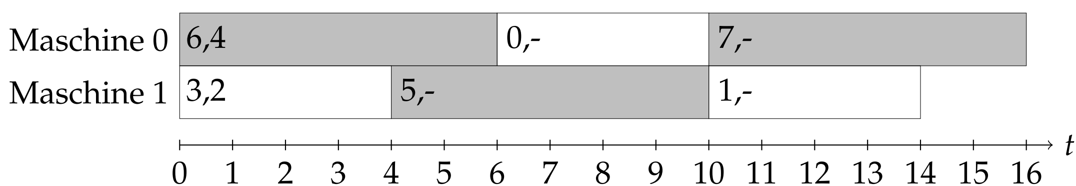

Abbildung 3.4.: Ablaufplan Beispiel Dekoder 2

Die Batches, die Jobs der Familie 0 beinhalten, sind mit weißem Hintergrund gezeichnet, die Batches mit Jobs der Familie 1 sind grau hinterlegt.

In diesem Beispiel gibt es auf jeder Maschine für jede Familie einen unvollständigen Batch.

Am Beispiel 3.3 ist zu sehen, dass durch die Verteilung der Jobs auf die Maschinen für jede Familie unvollständige Batches auf der Maschine entstehen können. Eine Verbesserung des TWT-Wertes kann erzielt werden, wenn unvollständige Batches gefüllt werden und dafür andere Batches geleert werden. Vollständig geleerte Batches können aus der Liste der Batches entfernt werden und dafür nachfolgende Batches nach vorne geschoben werden. Dieses Balancing wird mit einer Balancing-Heuristik bearbeitet. Wenn die Anzahl der Jobs einer Familie ein Vielfaches der Batchgröße ist, dann wird am Ende des Balancings für diese Familie keine unvollständigen Batches übrig bleiben.

Wenn ein Ablaufplan mit optimalen TWT-Wert vorhanden ist, dann gibt es dazu einen Ablaufplan mit demselben TWT-Wert, der jedoch je Familie höchstens einen unvollständigen Batch hat. Hat der optimale Ablaufplan zwei unvollständige Batches derselben Familie, so können Job aus einem Batch in den anderen Batch verschoben werden, bis einer der beiden Batches voll ist. Sind die Fertigstellungstermine unterschiedlich, dann werden Jobs aus dem Batch mit dem späteren Fertigstellungstermin in den Batch mit dem früheren Fertigstellungstermin verschoben.

Wenn ein beliebiger Ablaufplan mit unvollständigen Batches mit der Balancing-Heuristik bearbeitet wird, dann kann es Verbesserungen des TWT-Wertes geben, wenn der Fertigstellungstermin des Ziel-Batches kleiner ist als der Fertigstellungstermin des Quell-Batches. Ein weitere Möglichkeit der Verbesserung des TWT-Wertes ist dann möglich, wenn ein Batch vollständig geleert wird und nachfolgende Batches auf derselben Maschine nach vorne geschoben werden, also der Fertigstellungstermin früher gesetzt werden kann.

Das Verfahren der Balancing-Heuristik, wie sie im Rahmen dieser Arbeit verwendet wird, ist im Algorithmus 3.7 dargestellt.

Zuerst werden die Batches der Ablaufplanung nicht-absteigend nach dem Fertigstellungstermin sortiert (Schritt 01). Es wird $b1$ auf den ersten Batch im Ablaufplan gestellt (Schritt 02).

In der Schleife (Schritte 03 - 15) wird die Liste der Batches von vorne nach hinten durchgearbeitet, bis alle Batches betrachtet wurden. Nur wenn der Batch $b1$ unvollständig ist, wird versucht, Jobs aus späteren Batches (in der Liste nach dem Batch $b1$ kommend) auf den aktuellen Batch zu verschieben (Schritte 05 - 12).

Der Batch $b2$ wird auf den letzten Batch im Ablaufplan gestellt (Schritt 05). Die Liste der Batches wird nun von hinten nach vorne durchgearbeitet, bis der Batch $b1$ erreicht ist (Schritte 06 - 12). Damit werden die späteren Batches derselben Familie, die nicht vollständig sind, egal auf welcher Maschine, bearbeitet. Eine Bearbeitung erfolgt nur, wenn der Batch $b2$ unvollständig ist und darüber hinaus die Familie der Jobs im Batch $b2$ (kurz die Familie von Batch $b2$) gleich der Familie von Batch $b1$ ist, also gleich der Familie der Jobs der im Batch $b1$ ist (Schritte

Algorithmus 3.7 : Balancing-Heuristik

Data : Probleminstanz P mit n Jobs und ein Ablaufplan L_{Batch} für die Jobs
Result : L_{Batch} Ablaufplan mit verringerter Anzahl der unvollständigen Batches

```
1  sortiere Batches des Ablaufplanes nicht-absteigend nach Fertigstellungstermin;
2  b1 := erster Batch im Ablaufplan;
3  repeat
4      if (b1 unvollständig) then
5          b2 := letzter Batch der Liste;
6          repeat
7              if ((b2 unvollständig) und (Familie von b1 = Familie von b2)) then
8                  verschiebe Jobs von b2 nach b1;
9                  if (b2 leer) then lösche Batch b2 aus dem Ablaufplan;
10             end if
11             b2 := vorheriger Batch in der Liste;
12         until (b2 = b1);
13     end if
14     b1 := nächster Batch in der Liste;
15 until (Liste der Batches durchgearbeitet);
```

07 - 10). Aus dem Batch $b2$ werden Jobs in den Batch $b1$ verschoben, bis der Batch $b1$ vollständig gefüllt ist oder der Batch $b2$ leer ist (Schritt 08). Wenn der Batch $b2$ leer ist, also keinen Job mehr beinhaltet, dann wird der Batch aus dem Ablaufplan gelöscht (Schritt 09). Dies bedeutet gleichzeitig, dass die Batches auf der Maschine, zu dem der gelöschte Batch gehört nach vorne geschoben werden können. Damit werden die Fertigstellungstermine der Batches auf dieser Maschine reduziert werden. Damit wird die Liste der Batches modifiziert. Im Schritt 11 wird der Batch $b2$ auf den Batch vor dem Batch $b2$ gestellt.

Im Schritt 14 wird $b1$ auf den nächsten Batch im Ablaufplan gesetzt. Die Batches im Ablaufplan werden somit zum einen von vorne nach hinten durchlaufen ($b1$) und zum anderen von hinten nach vorne ($b2$).

Beispiel 3.4 (Fortsetzung Beispiel 3.3)**.** *Die Anwendung der Balancing-Heuristik in diesem Beispiel sorgt dafür, dass der Job $j = 7$ in den Batch mit dem Job $j = 5$ verschoben wird. Der Job $j = 1$ wird zum Batch mit dem Job $j = 0$ verschoben. Der neue Ablaufplan hat dann die Gestalt, die in Abbildung 3.5 dargestellt ist.*

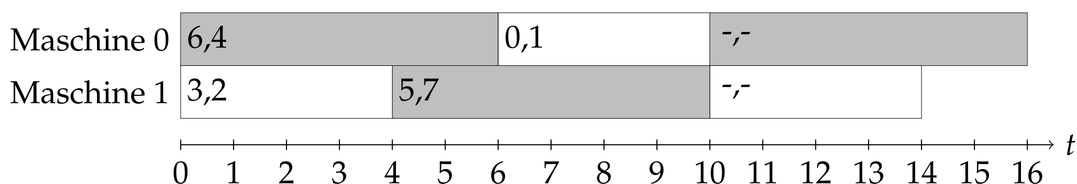

Abbildung 3.5.: Ablaufplan Beispiel Dekoder 2 nach Balancing-Heuristik

Die leeren Batches werden entfernt, auch wenn sie in der Grafik noch als Rahmen enthalten sind.

Diese Balancing-Heuristik kann nach Algorithmus 3.6 durchgeführt werden. Es wird damit vor der Durchführung der Swap-Heuristik ausgeführt.

Kapitel 4. Implementierung der genetischen Algorithmen

Im diesem Kapitel werden Implementierungsdetails erläutert. Dazu wird das Rahmenwerk für den *brkGA*, das von Toso und Resende [26] erstellt wurde, erläutert (Abschnitt 4.1). Insbesondere die Änderungen, die im Rahmen der Arbeit vorgenommen wurden, werden erläutert. Im Abschnitt 4.2 werden Details der Implementierung zu den im Rahmen dieser Arbeit erstellten Decodern und zum Gesamtablauf dargestellt.

4.1. Beschreibung des brkgaAPI-Rahmenwerkes

Basis ist das *brkgaAPI*-Framework für *BRKGA* gemäß Toso und Resende [26]. Es ist in der Programmiersprache C++ erstellt. Durch einfache Anpassungen kann es auch für *RKGA* und *RKGA∗* verwendet werden.

Das *brkgaAPI*-Framework besteht aus drei Teilen, die als Header-Dateien realisiert sind und drei Dateien, die als Beispiele für die Implementierung dienen. Diese Teile sind in der Tabelle 4.1 aufgeführt und werden im Nachfolgenden beschrieben.

Tabelle 4.1.: Komponenten des *brkgaAPI*-Frameworks

Datei	Beschreibung
`BRKGA.h`	Biased Random-Key Genetic Algorithm
`Population.h`	Datenstruktur für die Darstellung einer Population
`MersenneTwister.h`	Pseudozufallsgenerator
`SampleDecoder.h` `SampleDecoder.cpp`	Beispiel für einen Dekoder
`samplecode.cpp`	Beispiel für einen Ablauf

Der Kern ist `BRKGA.h`, während die beiden anderen Teile zur Unterstützung dienen.

Es wurden einige wenige technische Änderungen am Framework vorgenommen, um Fehlermeldungen des Compilers zu bereinigen. Diese werden hier nicht beschrieben. Die inhaltlichen Änderungen am Framework, um das Framework auch für die Methoden *RKGA* einsetzbar zu machen, werden im Abschnitt 4.1.3 beschrieben.

4.1.1. Pseudozufallsgenerator

Zur Ermittlung von Zufallszahlen verwendet das *brkgaAPI*-Framework den von Matsumoto und Nishimura [17] beschrieben Mersenne-Twister Algorithmus. Es ist ein schneller Algorithmus, der eine Periodenlänge von $p = 2^{19937} - 1$ (eine Mersenne-Primzahl) hat und nahezu gleichverteilte Zufallszahlen erzeugt. Dieser Zufallszahlengenerator ist in `MersenneTwister.h` realisiert.

4.1.2. Datenstruktur für eine Population

In der Header-Datei `Population.h` wird die Datenstruktur der Population als Klasse erzeugt. Eine Population wird dabei als ein zwei-dimensionales Feld der Dimension $p \times n$ dargestellt. Hierbei ist p die Größe einer Population, also die Anzahl der Chromosomen, die zu einer Generation gehören. Der Parameter n ist die Länge eines Chromosoms. Die Definition der Datenstruktur erfolgt durch

```
std::vector< std::vector< double > > population; ,
```

also ein Vektor über einen Vektoren von reellen Zahlen. Der innere Vektor repräsentiert ein Chromosom.

Neben der Datenstruktur für die Population werden auch die Fitnesswerte der Chromosomen gespeichert. Dies wird in einem Feld der Länge p gespeichert. Dazu dient die Definition

```
std::vector< std::pair< double, unsigned > > fitness; .
```

Im Paar in diesem Vektor besteht dabei aus dem Fitnesswert und dem Index, der auf die Spalte in der Datenstruktur `population` verweist, zu dem der Fitnesswert gehört. Wenn damit die Population nach dem Fitnesswert sortiert wird, dann wird nur der Vektor `fitness` sortiert. Die Felder sind dabei mit der flexiblen Container-Klasse `vector` implementiert.

Für die Implementierung des *brkgaAPI*-Frameworks werden zwei Populationen definiert, die alte Population (`previous`) und die aktuelle (nächste) Population (`current`), die aufgebaut wird, was im nachfolgenden Unterabschnitt beschrieben wird. Ein Generationswechsel wird dadurch als ein Wechsel des Zeigers auf die entsprechende Population realisiert.

4.1.3. Rahmenwerk

Der Kern des *brkgaAPI*-Frameworks ist in `BRKGA.h` implementiert. Zum einem werden in der Klasse `BRKGA` die notwendigen Datenstrukturen für den Algorithmus definiert, zum anderen werden die benötigten Methoden für den Übergang von einer Generation zur nächsten Generation definiert.

Im Konstruktor

```
BRKGA(unsigned n, unsigned p, double pe, double pm, double rhoe,
  const Decoder& refDecoder, RNG& refRNG,
  unsigned K = 1, unsigned MAX_THREADS = 1);
```

werden die wichtigen Parameter für das Verfahren übergeben.

- Der Parameter `n` bestimmt die Anzahl der Gene in jedem Chromosom.
- Der Parameter `p` bestimmt die Anzahl der Chromosomen in einer Population.
- Der Parameter `pe` (p_e) bestimmt den Anteil der Elemente einer Population, die zur Elite gehören. Die Eingabe ist ein `double`-Wert. Er wird beim Aufruf in die interne `unsigned`-Variable `pe` übertragen. Hierzu wird die Größe der Population p mit dem `double`-Wert p_e multipliziert. Ist $p = 1.000$ und $p_e = 0,2$, dann gehören je Generation $p \cdot p_e = 200$ Chromosome zur Elite.
- Der Parameter `pm` ist der Anteil der Mutanten, die bei jeder Generation neu in die Population eingefügt werden. Die Eingabe ist ein `double`-Wert. Er wird beim Aufruf in die interne `unsigned`-Variable `pm` (p_m) übertragen. Hierzu wird die Größe der Population `p` mit dem `double`-Wert `pm` multipliziert. Ist $p = 1.000$ und $p_m = 0,1$, dann werden je Generation $p \cdot p_m = 100$ Chromosome neu zur Generation hinzugefügt.
- Der Parameter `rhoe` (ρ_e) ist die Wahrscheinlichkeit, dass bei einem Crossover zweier Elternteile die Allele des ersten Elternteils ausgewählt werden.
- Der Parameter `refDecoder` ist eine Referenz auf einer Dekoder. Der Dekoder hat die Aufgabe, ein Chromosom zu übernehmen und über die Dekodierung ein Fitnesswert zu berechnen. Die Dekoder sind die angepasst an das jeweilige Problem.
- Der Parameter `refRNG` ist eine Referenz auf den Pseudozufallsgenerator, der für die Generierung der Zufallszahlen verwendet wird. Im Rahmen dieser Arbeit wurde der vom *brkgaAPI*-Framework angebotene Pseudozufallsgenerator (siehe 4.1.1) verwendet.

Die beiden optionalen Parameter `K` und `MAX_THREAD` können dazu verwendet werden, eine parallele Verarbeitung durchzuführen. Es wird im Rahmen dieser Arbeit `K = 1` und `MAX_THEADS = 1` verwendet, das bedeutet, es wird ohne parallele Verarbeitung gearbeitet.

Die wesentlichen Anpassungen wurden in der Funktion

```
void BRKGA< Decoder, RNG >::
evolution(Population& curr, Population& next, std::string method )
```

durchgeführt. Die erste Änderung besteht darin, dass in der Signatur der Funktion der zusätzliche Parameter `std::string method` übergeben wird. Hierdurch wird beim Aufruf der Funktion festgelegt, ob die Bildung der nächsten Generation nach dem $RKGA$-Verfahren, dem $RKGA*$-Verfahren oder dem $BRKGA$-Verfahren durchgeführt wird. Im Rahmen dieser Arbeit werden mit dieser Variation keine Experimente durchgeführt.

Der Anfang der Funktion ist unabhängig von dem gewählten Verfahren. Im zweiten Schritt der Funktion wird die Elite von der vorherigen Generation zur nächsten Generation übertragen. Im nächsten Schritt wird das Crossover durchgeführt. In diesem Schritt sind die Änderung am Algorithmus eingebaut, damit der Algorithmus die verschiedenen Verfahren unterstützen kann. Dies wird im nächsten Absatz erläutert. Im abschließenden Schritt wird die Mutation durchgeführt, das bedeutet, die Migration von p_m neuen Chromosomen durchgeführt.

Im Original-$brkgaAPI$-Framework wird zuerst das Elternteil aus dem Elitebereich selektiert:

```
// Select an elite parent:
const unsigned eliteParent = (refRNG.randInt(pe - 1)); .
```

Die Index des Elternteils in der Elite ist im Bereich $[0, \ldots, (p_e - 1)]$. Anschließend wird das Elternteil selektiert, welches nicht in der Elite ist:

```
// Select a non-elite parent:
const unsigned noneliteParent = pe + (refRNG.randInt(p - pe - 1)); .
```

Der Index dieses zweiten Elternteils ist im Bereich $[(p_e - 1), \ldots, (p - 1)]$. Danach wird das Crossover durchgeführt:

```
// Mate:
for(j = 0; j < n; ++j) {
  const unsigned sourceParent
    = ((refRNG.rand() < rhoe) ? eliteParent : noneliteParent);
  next(i, j) = curr(curr.fitness[sourceParent].second, j);
} .
```

Bei jedem Gen wird per Zufall entschieden, ob das Allel vom Elternteil aus der Elite übernommen wird (mit einer Wahrscheinlichkeit von ρ_e) oder ob das Allel vom Elternteil aus der nicht-Elite genommen wird. So wird Gen für Gen das neue Chromosom aufgebaut.

Für $RKGA$ und $RKGA*$ werden beide Elternteile aus dem gesamten Bereich selektiert, also ist der Index im Bereich $[0, \ldots, (p - 1)]$. Für die verschiedenen Verfahren werden die Grenzen für den Selektionsbereich gesetzt:

```
unsigned border1;
unsigned offset2;
unsigned border2;
if (method == "BRKGA") {
  border1 = pe - 1;
  offset2 = pe;
  border2 = p - pe - 1;
} else if ((method == "RKGA") || (method == "RKGA*")) {
  border1 = p - 1;
  offset2 = 0;
  border2 = p - 1;
}; .
```

Für die Selektion der Elternteile werden die definierten Grenzen wie folgt berücksichtigt:

```
// Select parent 1:
const unsigned parent1 = (refRNG.randInt(border1));
// Select a parent 2:
const unsigned parent2 = offset2 + (refRNG.randInt(border2)); .
```

Basierend darauf wird für jedes Gen das Elternteil ausgewählt, welches das Allel bestimmt:

```
const unsigned sourceParent
  = ((refRNG.rand() < rhoe) ? parent1 : parent2);
next(i, j) = curr(curr.fitness[sourceParent].second, j); .
```

In diesem Ausschnitt ist nur die Veränderung aufgeführt, wenn mit dem Rahmenwerk *RKGA* durchgeführt wird. Die Anpassungen für $RKGA*$ sind hier nicht detailliert aufgeführt. Bei $RKGA*$ wird nach der Selektion von zwei Elternteilen die Zuordnung zu Elternteil 1 und Elternteil 2 durchgeführt. Das Elternteil mit der besseren Fitness wird das Elternteil 1, welches somit mehr Allele an das Kindindividuum überträgt als das Elternteil 2. Da die Elternteile nach der Fitness sortiert sind, bedeutet dies, dass der niedrigere Index auf das Elternteil 1 verweist, der höhere Index auf das Elternteil 2. Mehr Details können in der Source vom Programm nachgesehen werden, die auf der CD gespeichert sind.

Weitere fachliche Veränderungen wurden am *brkgaAPI*-Rahmenwerk nicht derzeit durchgeführt.

4.2. Implementierungsdetails

In diesem Abschnitt werden einige ausgewählte Aspekte der Implementierung, die im Rahmen dieser Arbeit erstellt wurde, dargestellt.

4.2.1. Dekoder

In der abstrakten Klasse `Decoder` wird die virtuelle Methode `decode`

```
virtual double decode(const DVector& chromosome) const;
```

definiert, so wie sie vom *brkgaAPI*-Framework gefordert wird. Damit wird aus einem Chromosom mittels einem Dekoder ein Ablaufplan erstellt und der dazugehörige *TWT*-Wert berechnet. Hierbei ist für den *RKGA*-Algorithmus nur der *TWT*-Wert (oder genauer der Fitness-Wert) für das Chromosom wichtig. Der konkrete Ablaufplan wird nicht benötigt.

Darüber hinaus werden in dieser abstrakten Klasse einige konkrete Methoden definiert, die bei den verschiedenen Dekodern eingesetzt werden.

In der Definition ist ein konkretes Datenfeld definiert. In der Variablen `withSwap` vom Typ *bool* wird gespeichert, ob bei der Anwendung des Dekoders die Swap-Heuristik (siehe 2.4.3) angewendet wird oder nicht. Wird die Swap-Heuristik angewendet, dann ist es notwendig, den erstellten Ablaufplan zu speichern, damit nach der Erstellung die Swap-Heuristik angewendet werden kann. Darüber hinaus ist beim `Decoder2`, dem Decoder für die simultane Zuordnungen-Reihenfolge-Entscheidungen (siehe Abschnitt 3.3.2), die Speicherung des Ablaufplanes auch notwendig, wenn die Balancing-Heuristik angewendet wird. Muss der Ablaufplan nicht gespeichert werden, dann kann nach der Erstellung des Batches und der Aktualisierung des *TWT*-Wertes der Batch vergessen, also gelöscht werden. Dadurch ist die Verarbeitungszeit für einen Aufruf des Dekoders damit deutlich niedriger, als wenn dies erforderlich ist. Untersuchungen und numerische Experimente dazu sind in den Abschnitten 5.2.4 (Swap-Heuristik) und 5.2.5 (Balancing-Heuristik) beleuchtet.

Von der abstrakten Klasse `Decoder` sind im Rahmen dieser Arbeit zwei spezielle Dekoder implementiert worden. In der abgeleiteten, konkreten Klasse `Decoder1` ist die im Abschnitt 3.3.1 beschriebene Permutationsrepräsentation implementiert. Die Implementierung ist vergleichbar mit der Implementierung der *ATC-BATC*-Heuristik (siehe Abschnitt 2.4.1). Bei der *ATC-BATC*-Heuristik werden die Jobs zu jedem Zeitpunkt an dem eine Entscheidung für die Bildung eines Batches notwendig ist nach dem *ATC*-Wert sortiert, der zum jeweiligen Zeitpunkt neu berechnet wird. Bei diesem Dekoder werden die Jobs am Anfang nach dem Random-Key sortiert, die Reihenfolge bleibt dann unverändert. In der abgeleiteten, konkreten Klasse `Decoder2` ist die im Abschnitt 3.3.2 beschriebene Repräsentation für simultane Zuordnungen-Reihenfolge-Entscheidungen implementiert.

In der Abbildung 4.1 ist ein Klassendiagramm mit wichtigsten beteiligte Klassen gezeichnet. Die abstrakte Klasse *Decoder* und die beiden von der Klasse *Decoder* abgeleiteten Klassen *Decoder1* und *Decoer2*. Die Klasse *Decoder* hat einen Verweis

auf die Klasse *ProblemInstance*. In der Klasse *ProblemInstance* beinhaltet unter anderen einen Vektor von Elementen der Klasse *JobData* zur Speicherung der Daten der Jobs.

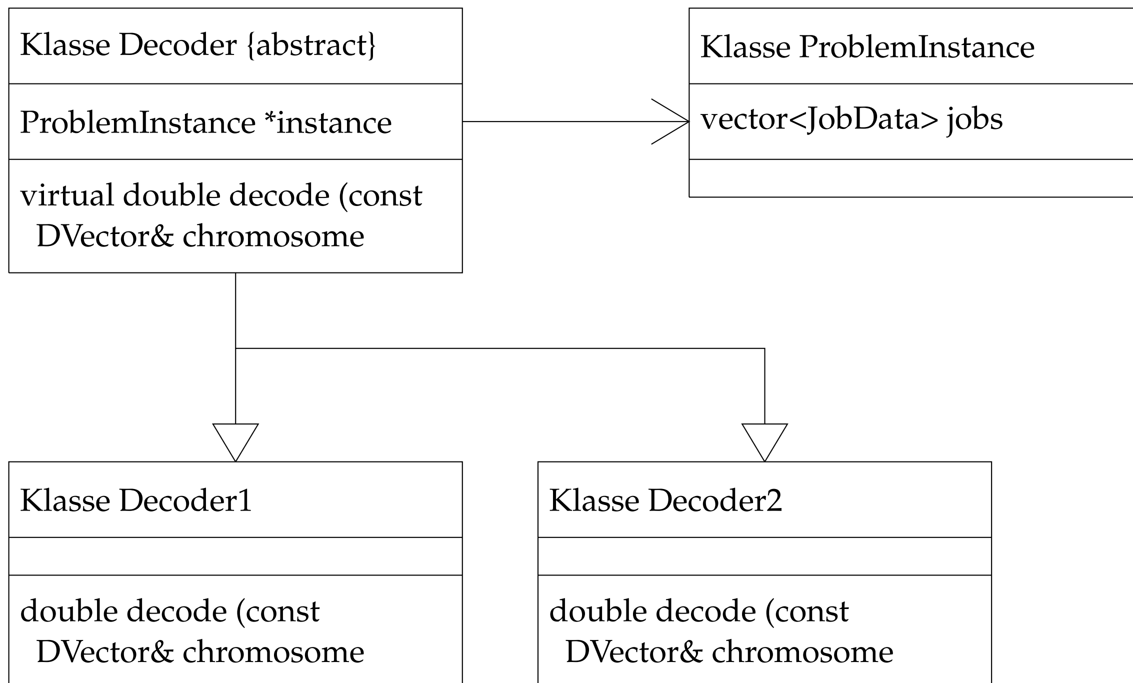

Abbildung 4.1.: Klassendiagramm Dekoder

Weitere Dekoder können als Ableitungen der Klasse `Decoder` gebildet werden. Wichtig ist, dass diese Klassen dann jeweils die Methode `decode` implementieren und für ein Chromosom einen Fitnesswert zurück geben.

4.2.2. Ablaufplan

Der Ablaufplan wird als Liste von Batches implementiert. In der Klasse `Batch` (`Batch.h / Batch.cpp`) wird die Datenstruktur für einen Batch definiert:

Datenfeld	Beschreibung
`int maxJobs;`	maximale Anzahl der Jobs eines Batches
`std::vector<int> jobs;`	Liste der Jobs des Batches
`double value;`	berechneter Wert zum Batch
`int processTime;`	Prozesszeit für den Batch
`int completion;`	Fertigstellungstermin
`int machine;`	Nummer der Maschine
`int family;`	Nummer der Familie

Im Datenfeld `jobs` wird die Liste der Jobs, in diesem Batch zugeordnet sind gespeichert. Es wird nur die Nummer des Jobs gespeichert. Über diese Nummer

werden die benötigten Daten für den Job in der Klasse *ProblemInstance* angesprochen.

Im Datenfeld `value` werden unterschiedliche Werte gespeichert. Im Rahmen der *ATC-BATC*-Heuristik wird Anfang der *BATC*-Wert eine Batches gespeichert, in späteren Teilen ist in diesem Feld der wT-Wert hinterlegt.

In `scheduling.h` ist die Datenstruktur für einen Ablaufplan, also eine Liste von Batches definiert

```
typedef std::vector<Batch> VBatch; .
```

Diese Datenstruktur wird verwendet, wenn der Ablaufplan gespeichert wird. Bei der *ATC-BATC*-Heuristik wird ein Ablaufplan erstellt. Bei den anderen Heuristiken (Dekompositionsheuristik, Swap-Heuristik, Balancing-Heuristik) wird jeweils ein Ablaufplan als Eingabe verwendet und dieser Ablaufplan gemäß der jeweiligen Heuristik modifiziert.

Kapitel 5.

Numerische Experimente

In diesem Kapitel wird die Durchführung der Tests beschrieben. Zuerst (Abschnitt 5.1 wird das Design der Tests erstellt. Im Abschnitt 5.2 werden Untersuchungen für die Auswahl der Parameter der Algorithmen durchgeführt. Im Abschnitt 5.3 werden die Ergebnisse der numerischen Experimente mit den gewählten Parametern erstellt. Abschließend werden im Abschnitt 5.4 die Ergebnisse analysiert und interpretiert.

5.1. Versuchsplanung

5.1.1. Probleminstanzen

Für die Durchführung des numerischen Test wurden Instanzen herangezogen, die auch bei Almeder und Mönch [1] verwendet wurden. In der Tabelle 5.1 sind die Parameter für die Erstellung der Testdaten aufgeführt.

Es gibt drei Varianten von inkompatiblen Familien. Für die Anzahl der Jobs gibt es ebenfalls drei Varianten. Hierbei ist immer die Anzahl der Jobs ein Vielfaches des Anzahl der Familien. Für die maximale Größe Batches gibt es zwei Varianten. Nur bei der Variante mit 180 Jobs, drei inkompatiblen Familien und einer maximalen Batchgröße von vier kann ein Ablaufplan gestaltet werden, so dass alle Batches voll gefüllt sind. Bei allen anderen Varianten gibt es nicht gefüllte Batches.

Für die Bearbeitungszeit gibt es fünf Varianten, die mit unterschiedlicher Wahrscheinlichkeit gewählt werden. Die Werte für das Gewicht eines Jobs sind Realisierungen einer Zufallsvariablen, welche gleichverteilte Zufallszahlen im Intervall $[0, 1]$ liefert. Die Werte für die Fälligkeitstermine der Jobs sind ebenso Realisierungen einer Zufallsvariablen, welche gleichverteilte Zufallszahlen liefert. Das Intervall, in dem die Werte sind, ist $[\mu(T) \cdot (1 - R/2), \mu(T) \cdot (1 + R/2)]$ mit den beiden Parametern R (*range*) für die Spanne der Werte und T (*tightness*) für die Dichte der Ablaufpläne. Es sind

$$\hat{C}_{max} = \frac{n \cdot E(p)}{m \cdot B} \tag{5.1}$$

Tabelle 5.1.: Gestaltung der Versuchsdaten

Parameter	Ausprägungen	Anzahl
Anzahl Familien f	3; 6; 12	3
Anzahl Jobs je Familie	180/f; 240/f; 300/f	3
maximale Batchgröße B	4; 8	2
Bearbeitungszeit p_s je Familie	2 mit Wahrscheinlichkeit 0,2 4 mit Wahrscheinlichkeit 0,2 10 mit Wahrscheinlichkeit 0,3 16 mit Wahrscheinlichkeit 0,2 20 mit Wahrscheinlichkeit 0,1	1
Gewicht	$w_j \sim U[0,1]$	1
gewünschter Fertigstellungstermin	$d_j \sim U[\mu(T) \cdot (1 - \frac{R}{2}), \mu(T) \cdot (1 + \frac{R}{2})]$ $R \in \{0,5; 2,5\}$ Spanne gewünschte Fertigstellungstermine	2
	$T \in \{0,3; 0,6\}$ Dichte Ablaufpläne	2
Anzahl Maschinen m	3; 4; 5; 6	4
Anzahl Kombinationen der Parameter		288
Anzahl unabhängige Testinstanzen je Kombination		5
Anzahl Testinstanzen		1.440

und

$$\mu(T) = \hat{C}_{max}(1 - T) . \tag{5.2}$$

Dabei ist $\hat{C}_{max}$ ein Schätzer für die Zykluszeit (*makespan*), wobei $E(p)$ der Erwartungswert für die Bearbeitungszeit ist. Die n Jobs benötigen in etwa $n \cdot E(p)$ Zeit für die Bearbeitung. Da es m Maschinen gibt und die maximale Batchgröße B beträgt, wird somit ungefähr $\hat{C}_{max}$ Zeit für die Bearbeitung benötigt.

Es gibt vier Ausprägungen für die Anzahl der Maschinen. Damit ergeben sich insgesamt 288 verschiedene Varianten. Für jede Variante gibt es fünf Instanzen. Damit gibt es insgesamt 1.440 verschiedene Testinstanzen.

Beispiel 5.1. *Mit den in der Tabelle 5.1 angegebenen Werten ist $E(p) = 9,4$. Wird die Variante mit $n = 180$ Jobs, $m = 3$ Maschinen und einer maximalen Batchgröße $B = 4$ bearbeitet, dann ergibt sich*

$$\hat{C}_{max} = \frac{180 \cdot 9,4}{3 \cdot 4} = 141 . \tag{5.3}$$

In der Tabelle 5.2 sind für die gewählten Varianten für T und R, die Intervalle $[\mu(T) \cdot (1 - \frac{R}{2}), \mu(T) \cdot (1 + \frac{R}{2})]$ der Fälligkeitstermine angegeben.

Tabelle 5.2.: Beispiele für Bereich Fälligkeitstermine

T	$\mu(T)$	$R = 0,5$	$R = 2,5$
$0,3$	$98,7$	$[74,025\ ,123,375]$	$[-24,675\ ,222,075]$
$0,6$	$56,4$	$[42,3\ ,70,5]$	$[-14,1\ ,126,9]$

5.1.2. Fragestellungen

In den ersten Testreihen werden verschiedene Fragestellungen untersucht. Hierbei werden die nachfolgenden Fragen untersucht:

1. Es ist zu untersuchen, ob *BRKGA* besser als *RKGA* ist. Hiermit soll das Ergebnis, welches bei Gonçalves *e.a.* [11] bereits ausgeführt wurde, bestätigt werden.
2. Es ist zu untersuchen, welcher der beiden Dekoder, die im Rahmen dieser Arbeit entwickelt wurden, die besseren Ergebnisse erzielt.
3. Es ist zu untersuchen, ob es sinnvoll ist, beim Dekodieren jedes mal die Swap-Heuristik anzuwenden?
4. Beim Dekoder 2, also dem Dekoder für die Repräsentation für simultane Zuordnungen-Reihenfolge-Entscheidungen, wurde im Rahmen dieser Arbeit eine Balancing-Heuristik entwickelt. Es ist zu untersuchen, welchen Einfluss die Anwendung dieser Heuristik auf die Ergebnisse hat.

Mit Hilfe dieser Fragestellungen soll die gemäß den Untersuchungen beste Variante ausgewählt werden. Mit dieser besten Variante werden dann Untersuchungen bezüglich der Güte der Lösung nach bestimmten maximalen Rechenzeiten (beispielsweise 15 Sekunden, 30 Sekunden, 45 Sekunden, 60 Sekunden) durchgeführt. Es werden die Ergebnisse nach diesen Rechenzeiten mit den Referenzwerten, also den Werten, die mit Hilfe der *ATC-BATC*-Heuristik erstellt werden, verglichen. Hierbei wird untersucht, ob es Verbesserungen gibt und welchen Einfluss die Anzahl der Familien, die maximale Batchgröße, die Anzahl der Jobs, die Anzahl der Maschinen oder die Parameter R und T auf die Ergebnisse haben.

5.1.3. Vergleich der Werte

Für den Vergleich der Ergebnisse der verschiedenen Versuchen wird die Verbesserung oder Verschlechterung des *TWT*-Wertes gegenüber dem Referenzwert berechnet. Der Referenzwert $TWT_{Referenz}$ ist dabei der *TWT*-Wert, der mittels der in Abschnitt 2.4 beschriebenen *ATC-BATC*-Heuristik ermittelt wird. Ist der *TWT*-Wert gemäß der getesteten Heuristik $TWT_{Heuristik}$, dann wird durch

$$\frac{TWT_{Referenz} - TWT_{Heuristik}}{TWT_{Referenz}} \tag{5.4}$$

die relative Veränderung der Versuchswerte gegenüber dem Referenzwert dargestellt. Ein positiver Prozentwert bedeutet dabei eine Verbesserung, ein negativer Wert eine Verschlechterung gegenüber dem Referenzwert.

5.1.4. Parameter der Rechner

Die numerische Experimente wurden meistens mit einem Intel Core i5 Prozessor mit einer Taktfrequenz von 2,50 GHz, mit 4 GB Hauptspeicher und dem Betriebssystem Windows 7 durchgeführt (im Nachfolgenden kurz $i5$-Maschine genannt). Einige Experimente wurden mit einem Intel Core i3 Prozessor mit einer Taktfrequenz von 1,40 GHz, mit 4 GB Hauptspeicher und dem Betriebssystem Windows 7 durchgeführt (im Nachfolgenden kurz $i3$-Maschine genannt). Wenn nichts anderes ausgesagt wird, dann sind die Experimente mit der $i5$-Maschine durchgeführt worden.

5.2. Parameterauswahl für die Heuristiken

Für die Untersuchungen und Wahl der Parameter wurden einige numerische Experimente durchgeführt. Die Zusammenfassungen der Ergebnisse sind in den nachfolgenden Abschnitten hinterlegt. Ausführlichere Ergebnisse sind auf der beigefügten CD-ROM (siehe Kapitel C im Anhang) hinterlegt. Diese Untersuchungen sind in der Regen mit den ersten 99 Testinstanzen durchgeführt worden, statt mit dem gesamten Umfang von 1.440 Testinstanzen. Damit ist die benötigte Rechenzeit geringer.

5.2.1. Basisparameter

Die Auswahl der Parameter für die ATC-$BATC$-Heuristik, die Dekompositionsheuristik (DH) und die Swap-Heuritik orientiert sich an Almeder und Mönch [1], da dessen Werte als Referenz gelten. Damit kann die Veränderungen dieser Ausarbeitung mit der Ergebnissen, die dort ermittelt wurden, verglichen werden. Die Wahl der Parameter ist im Rahmen der Beschreibung der Heuristiken, im Abschnitt 2.4 bereits diskutiert. Für die ATC-$BATC$-Heuristik ist der entscheidende Parameter der Vorschauparameter κ. Es wird $\kappa = 0,5 \cdot k$ mit $k = 1, \ldots, 10$ gesetzt. Die Dekompositionsheuristik wird durch die Parameter λ, α und $iter$ gesteuert. Der Parameter λ beschriebt, wie viele Objekte in jedem Iterationslauf untersucht werden. Mit dem Parameter α wird festgelegt, dass die ersten α Objekte der untersuchten λ Objekte nach einem Iterationslauf fixiert werden. Der Parameter $iter$ gibt an, wie viele Iterationen maximal bearbeitet werden. In Anlehnung an Almeder und Mönch [1] wird $\lambda = 5$, $\alpha = 2$ und $iter = 15$ gesetzt.

Für $RKGA$ und $BRKGA$ gibt es einige Parameter, die bei Gonçalves und Resende [10] dargestellt und erläutert sind. Diese sind in Tabelle 5.3 zusammen gefasst.

Tabelle 5.3.: Parameter für *BRKGA*

Parameter	Beschreibung	Empfehlung
p	Größe der Population	$p = an$, mit $a \geq 1$, wobei n die Länge eines Chromosoms ist.
p_e	Größe der Elite	$0,10p \leq p_e \leq 0,25p$
p_m	Größe der Mutation	$0,10p \leq p_m \leq 0,30p$
ρ_e	Wahrscheinlichkeit, dass beim Crossover das Allel des ersten Elternteils gewählt wird	$0,5 < \rho_e \leq 0,8$

Für die Untersuchung der Parameter p_e, p_m und ρ_e wurden einige Experimente durchgeführt, um zu sehen, welche Auswirkungen die Veränderungen haben. In der Tabelle 5.4 sind Ergebnisse mit für die ersten 99 Testinstanzen mit den Parametern $p = 1000$ und der Methode *BRKGA* für einige Varianten der Parameter p_e,p_m und ρ_e dargestellt. Die Vergleichbasis ist hierbei jeweils $p_e = 0,2$, $p_m = 0,1$ und $\rho_e = 0,7$

Tabelle 5.4.: Vergleichsergebnisse p_e, p_m und ρ_e

p_e	p_m	ρ_e	00 sec	10 sec	20 sec	30 sec
0,15	0,10	0,70	-391,03%	-218,39%	-115,51%	-65,13%
0,20	0,10	0,70	-391,03%	-227,05%	-129,41%	-76,12%
0,25	0,10	0,70	-391,03%	-240,29%	-139,91%	-85,08%
0,20	**0,05**	0,70	-391,03%	-221,69%	-120,73%	-67,46%
0,20	**0,10**	0,70	-391,03%	-227,05%	-129,41%	-76,12%
0,20	**0,15**	0,70	-391,03%	-236,93%	-138,29%	-87,78%
0,20	0,10	**0,60**	-391,03%	-240,66%	-144,61%	-91,18%
0,20	0,10	**0,70**	-391,03%	-227,05%	-129,41%	-76,12%
0,20	0,10	**0,80**	-391,03%	-223,64%	-124,89%	-72,06%

In einem Experiment mit der Laufzeit 120 Sekunden wurden dann anschließend die Parameterkombination $p_e = 0,2$, $p_m = 0,1$ und $\rho_e = 0,7$ mit der Parameterkombination $p_e = 0,15$, $p_m = 0,05$ und $\rho_e = 0,8$, welche nach der Tabelle 5.4 die beste Wahl erscheint, verglichen. Die Ergebnisse sind in der Tabelle 5.5 hinterlegt.

Tabelle 5.5.: Vergleichsergebnisse p_e, p_m und ρ_e bei 120 Sekunden

p_e	p_m	ρ_e	60 sec	80 sec	100 sec	120 sec
0,20	0,10	0,70	-17,64%	-5,71%	-0,23%	2,38%
0,15	0,05	0,80	-11,72%	-4,23%	-1,23%	1,08%

Hier zeigt sich die Parameterwahl $p_e = 0,15$, $p_m = 0,05$ und $\rho_e = 0,8$ nicht

überlegen. Ohne dies im Rahmen dieser Arbeit weiter zu untersuchen werden die Parameter p_e, p_m und ρ_e gesetzt, wie nachfolgend beschrieben.

Der Parameter p_e beschreibt, wie groß der Anteil der Elite in der Population ist. Für die Experimente in dieser Arbeit wird $p_e = 0,20$ gesetzt, also 20% der Population gehört zur Elite.

Der Parameter p_m beschreibt den Anteil, der bei jeder Generation durch Mutation neu in die Generation beigesteuert wird. Diese neuen Elemente in der Generation werden zufällig bestimmt, das bedeutet, dass die Allele jeweils zufällig bestimmt werden. Für die Experimente in dieser Ausarbeitung wird $p_m = 0,10$ gesetzt, also in jeder Generation werden 10% der Population neu erstellt.

Der Parameter ρ_e bestimmt, mit welcher Wahrscheinlichkeit beim Crossover zweier Elternteile die Allele des ersten Elternteils (bei *BRKGA* also des Elternteils aus der Elite) das Gen des Kinds bestimmt. Falls $\rho_e = 0,5$ ist, dann würden beide Elternteile mit gleicher Wahrscheinlichkeit ausgewählt werden, ein Allel an das Kindindividuum zu übertragen. Wenn $\rho_e > 0.5$ ist, dann überträgt (bei *BRKGA*) das Elternteil aus der Elite durchschnittlich mehr Allele auf das Kindindividuum als das andere Elternteil. In dieser Arbeit wird mit $\rho_e = 0,7$ gearbeitet.

5.2.2. Vergleich - Methode und Dekoder

In diesem Abschnitt werden die Methode *BRKGA* und *RKGA*, sowie die beiden Dekoder verglichen. Damit soll bestätigt werden dass *BRKGA* besser als *RKGA* ist, wie dies schon von Gonçalves *et al.* [11] erkannt wurde. Des weiteren soll geprüft werden, welcher Dekoder die besseren Ergebnisse liefert.

In der Tabelle 5.6 sind die Ergebnisse für die 99 Testinstanzen für den Vergleich der Dekoder (Permutationsrepräsentation = Dekoder 1 oder simultane Repräsentation = Dekoder 2) und die Methode (*BRKGA* oder *RKGA*). Es ist die relative Verbesserung oder Verschlechterung zum Referenzwert angegeben. Es zeigt sich, dass die Permutationsrepräsentation der Dekoder ist, der im Durchschnitt bessere Ergebnisse liefert.

Tabelle 5.6.: Vergleich Methode / Dekoder

Eigenschaft	*BRKGA*-1	*RKGA*-1	*BRKGA*-2	*RKGA*-2
alle	6,03%	-5,82%	-12,01%	-44,43%
$n = 180$	8,17%	5,91%	-10,55%	-22,88%
$n = 240$	5,56%	-2,46%	-13,38%	-45,30%
$n = 300$	4,37%	-20,30%	-12,10%	-59,12%

In der Abbildung 5.1 ist für die erste Testinstanz der *TWT*-Wert nach der Zeit für den Dekoder mit der Permutationsrepräsentation für die Methoden *BRKGA* und *RKGA* exemplarisch dargestellt.

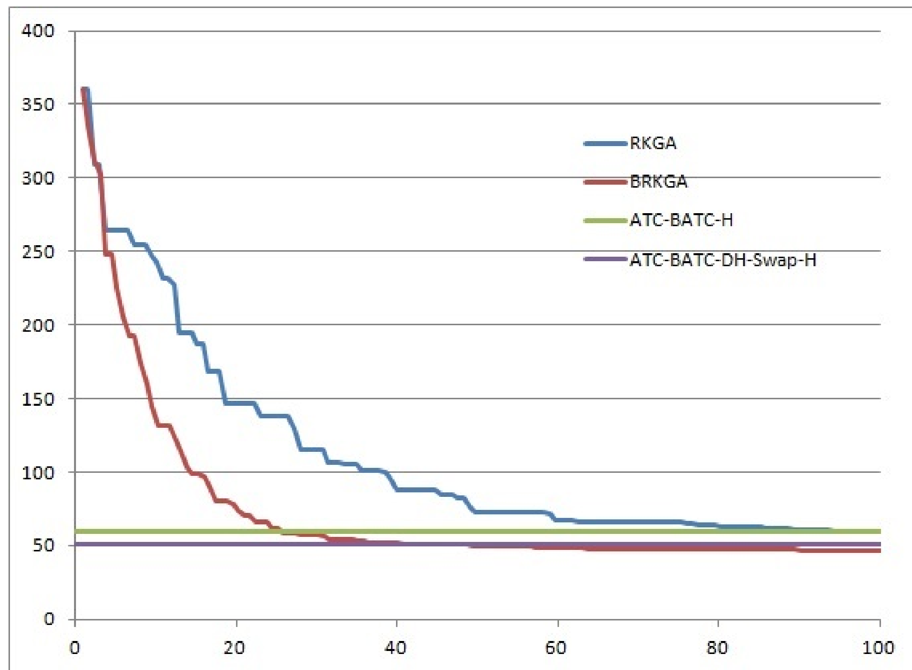

Abbildung 5.1.: Vergleich $BRKGA$ und $RKGA$, TWT-Werte nach Zeit

Es zeigt sich hier, dass die TWT-Werte für die $BRKGA$-Methode schneller kleiner werden als bei der $RKGA$-Methode. Dies ist die Bestätigung des Ergebnisses von Gonçalves *et al.* [11]. Bei den ersten 99 Testinstanzen ist es selten vorgekommen, dass in der ersten Generationen $RKGA$ einen besseren TWT-Wert produziert hat als $BRKGA$. Jedoch bereits nach maximal zehn Generationen, in der Regel schon nach fünf oder sogar zwei Generationen, lieferte $BRKGA$ bessere TWT-Werte als $RKGA$. Mit mehr Generation nähern sich die Werte an, wobei in den meisten Fälle nach 300 Sekunden, der Rechenzeit, die je Instanz und je Kombination Methode / Dekoder aufgewendet wurde $BRKGA$ weiterhin den besseren TWT-Wert als $RKGA$ geleifert hat. Nur bei den Instanzen 7 beziehungsweise 46 und 73 hatte die Methode $RKGA$ den besseren TWT-Wert als die Methode $BRKGA$ für Dekoder 1 beziehungsweise Dekoder 2.

In der Tabelle 5.7 sind Verhältnisse der TWT-Werte von $BRKGA$-Methode zu $RKGA$-Methode

$$\frac{TWT_{BRKGA-Methode}}{TWT_{RKGA-Methode}} \tag{5.5}$$

dargestellt. Es ist jeweils der Mittelwert er Verhältnisse der 99 ersten Testinstanzen für beide Dekoder dargestellt. Eine Verhältniszahl von 0,8 bedeutet dabei, dass die $BRKGA$ 20% bessere Werte liefert als die $RKGA$-Mathode.

Tabelle 5.7.: Vergleich $BRKGA$ besser als $RKGA$

Generation	20	40	60	80	100	200	300	400
Verhältnis	0,7048	0,6043	0,5868	0,6032	0,6312	0,7345	0,7770	0,7963

5.2.3. Populationsgröße

Bezüglich der Größe der Population werden Untersuchungen durchgeführt. Das bedeutet, es wird mit unterschiedlichen Populationsgrößen gearbeitet. Gemäß der Formel $p = an$ wurden für verschiedene Werte von a der TWT-Wert berechnet. Mit den 99 ersten Testinstanzen wurden Experimente durchgeführt. Dabei wurde mit der $BRKGA$-Methode und dem Dekoder für die Permutationsrepräsentation gearbeitet. Die maximale Zeit für die Berechnung pro Probleminstanz wurde dabei auf 90 Sekunden gesetzt. In der Tabelle 5.8 sind Daten der Untersuchungen aufgelistet.

Tabelle 5.8.: Vergleich Populationsgröße

Eigenschaft	$a = 1,5$	$a = 3,0$	$a = 4,5$
alle	-2,87%	-1,23%	-10,40%
$n = 180$	0,64%	5,59%	6,55%
$n = 240$	-3,40%	2,38%	-0,55%
$n = 300$	-5,84%	-11,66%	-37,20%

Die Größe der Population p soll größer sein als die Länge eines Chromosoms. Große Population bedeuten, dass weniger Generation bearbeitet werden können. Kleine Population bedeuten, dass die Vielfalt in einer Generation geringer ist.

Die jeweilige Populationsgröße ist $p = a \cdot n$. Über alle getestete 99 Instanzen ist der Faktor $a = 3,0$ der mit der geringsten Verschlechterung gegenüber dem Referenzwert. Für die Länge der Chromosomen $n = 180$ ergibt sich das beste Ergebnis für $a = 4,5$, also einer Populationsgröße von $p = 4,5 \cdot 180 = 810$. Für die Länge der Chromosomen $n = 240$ ergibt sich das beste Ergebnis für $a = 3,0$, also einer Populationsgröße von $p = 3,0 \cdot 240 = 720$. Für die Länge der Chromosomen $n = 300$ ergibt sich das bester Ergebnis für $a = 1,5$, also einer Populationsgröße $p = 1,5 \cdot 300 = 450$.

Bei einem weiteren Experiment bezüglich der Populationsgröße p wurden verschieben Läufe mit den ersten 99 Testinstanzen durchgeführt. Dabei sind stets die Parameter Methode $BRKGA$ und Dekoder 1 (Permutationsrepräsentation) identisch.

Die Parameter, die verändert wurden, sind dabei die Populationsgröße p und die Anzahl der Generation (gen). Hierbei wurde bei allen vier Versuchsreihen

das Produkt $p \cdot gen$ konstant (bei 100.000) belassen. Bei einer großen Population werden damit weniger Generationen gebildet als bei einer kleinen Population.

In der Tabelle 5.9 sind die Ergebnisse hinterlegt. Hierbei wird die durchschnittliche relative Veränderung für alle 99 Probleminstanzen sowie für die Teilmengen bezüglich der Anzahl der Jobs in der Probleminstanz dargestellt. Je kleiner die Population, desto schlechter ist das Ergebnis. Nur für eine Populationsgröße von 1.000 werden Verbesserungen gegenüber dem Referenzwert erzielt. Die Größe der Verbesserung ist dabei abhängig von der Anzahl der Jobs. Je mehr Jobs eingeplant werden, desto schlechter ist das Ergebnis. Demgegenüber sind die Verbesserungen der *ATC-BATC-DH*-Swap-Heuristik relativ unabhängig von der Anzahl der Jobs.

Tabelle 5.9.: Vergleich Populationsgröße (2)

Eigenschaft	*ATC-BATC-DH*-Swap-*H*	$p = 1000$ $gen = 100$	$p = 400$ $gen = 250$	$p = 200$ $gen = 500$	$p = 100$ $gen = 1000$
alle	4,46%	2,82%	-1,30%	-8,45%	-17,34%
$n = 180$	4,47%	6,52%	3,21%	-2,79%	-9,12%
$n = 240$	4,12%	2,17%	-1,87%	-9,15%	-19,78%
$n = 300$	4,78%	-0,22%	-5,24%	-13,41%	-23,13%

5.2.4. Swap-Heuristik

Die Dekoder können jeweils mit oder ohne die abschließende Durchführung der Swap-Heuristik angewendet werden. Wenn dabei die Swap-Heuristik durchgeführt wird, dann bedeutet dies, dass der Ablaufplan bei der Erstellung im Dekoder gespeichert werden muss, so dass er für die Durchführung der Heuristik zur Verfügung steht, also als Eingabe übergeben werden kann.

Die numerische Experimente für diese Untersuchung wurden mit der *BRKGA*-Methode und dem Dekoder 1, also dem Dekoder für die Permutationsrepräsentation, durchgeführt. Für die Durchführung mit beziehungsweise ohne Swap-Heuristik wurde jeweils eine Populationsgröße p von 1.000 gewählt. Somit wurde je Generation 800 mal die Swap-Heuristik aufgerufen (bei der Generation bei der Initialisierung 1.000 mal). Die Laufzeit wurde dabei auf der $i5$-Maschine auf jeweils 300 Sekunden je Instanz begrenzt. Das bedeutet, dass wenn nach der Durchführung einer Generation die 300-Sekunden-Grenze überschritten wurde, wurde die Berechnung beendet. Für die Version mit der Swap-Heuristik wurde dabei teilweise eine Laufzeit deutlich über der 300 Sekunden benötigt, es wurde dann jedoch eventuell nur eine Generation fortgeschrieben.

In der Tabelle 5.10 sind Ergebnisse für den Vergleich der Durchführung mit beziehungsweise ohne Swap-Heuristik zu finden. Hierbei wurde mit den ersten 99 Testinstanzen gearbeitet.

Tabelle 5.10.: Test Swap-Heuritik

Kriterium	TWT_v	ohne Swap Gen	ohne Swap TWT_H	mit Swap Gen	mit Swap TWT_H
alle	4,46%	312,7	6,03%	18,3	-4,58%
n = 180	4,47%	435,6	8,17%	31,6	2,39%
n = 240	4,12%	298,9	5,56%	15,2	-5,66%
n = 300	4,78%	203,7	4,37%	8,2	-10,47%
f = 3	5,11%	298,3	5,55%	3,4	-10,65%
f = 6	4,49%	317,8	6,25%	13,2	-4,63%
f = 12	3,77%	322,1	6,29%	38,3	1,53%

In der ersten Spalte der Tabelle ist aufgeführt, für welches Kriterium die Daten in der Zeile beschreiben. In der Spalte TWT_v ist die relative Verbesserung der Ablaufplanung mit der ATC-$BATC$-DH-Swap-Heursitik gegenüber der einfachen ATC-$BATC$-Heuristik, der Referenz. In der dritten und vierten Spalte sind die Daten für die Durchführung ohne Swap-Heuristik. In der dritten Spalte ist die Anzahl der Generationen aufgeführt, in der vierten Spalte der TWT-Wert mit dieser Heuristik. Die fünfte und sechste Spalte beinhalten die entsprechenden Werte (Anzahl der Generationen und TWT-Wert dieser Heuristik) für das Verfahren, wenn die Swap-Heuristik angewendet wird.

Ohne die Anwendung der Swap-Heuristik sind die Werte besser, als wenn die Swap-Heuristik angewendet wird. Besonders zu beachten ist, dass die Anzahl der Generation bei der Verwendung der Swap-Heuristik deutlich niedriger ist als ohne Verwendung der Swap-Heuristik.

Beachtenswert sind auch die Werte bezüglich der verschiedenen Anzahl der Familien. Die Ergebnisse mit der Verwendung der Swap-Heuristik werden besser, wenn die Anzahl der Familien höher wird. Die Swap-Heuristik behandelt jede Familie für sich. Wenn viele Familien vorhanden sind, dann ist die Anzahl der Batches der Familie geringer. Damit agiert die Swap-Heuristik auf einer kleineren Menge von Batches je Familie. Sei dazu die maximale Batchgröße $B = 4$. Für $n = 180$ Jobs hat der Ablaufplan mindestens 45 Batches. Bei $f = 3$ Familien sind das dann 15 Batches je Familie, für $f = 12$ sind es 4 Batches je Familie (also insgesamt 48 Batches), wobei ein Batch je Familie nicht voll gefüllt ist. Für $n = 300$ Jobs gibt es mindestens 75 Batches. Für $f = 3$ Familien 25 Batches je Familie, bei $f = 12$ Familien sind es 6 Batches je Familie. Je weniger Batches einer Familie vorhanden sind, desto schneller kann die Swap-Heuristik eine Familie durcharbeiten.

5.2.5. Balancing-Heuristik

Beim Dekoder für die Repräsentation für simultane Zuordnungen-Reihenfolge-Entscheidungen kann es auf jeder Maschine für jede Familie unvollständige Bat-

ches geben. Die im Abschnitt 3.3.2 dargestellte Balancing-Heuristik reduziert die Anzahl der unvollständigen Batches, auf maximal einen unvollständigen Batch je Familie. Dadurch kann der TWT-Wert verbessert werden.

Bei diesen Experimenten gilt für die Populationsgröße $p = 3,0 \cdot n$ je Instanz, wobei n die Anzahl der Jobs, also auch die Länge der Chromosomen ist. Es wurde die $BRKGA$-Methode verwendet. Für die 99 Testinstanzen wurde jeweils eine Rechenzeit von 90 Sekunden gewählt. In der Tabelle 5.11 sind die relative Veränderung zum Referenzwert in Abhängigkeit von der maximal möglichen Rechenzeit dargestellt. Diese Daten wurden auf der $i3$-Maschine erstellt.

Tabelle 5.11.: Test Balancing-Heuristik

Verfahren	Anz.Gen	00 sec	30 sec	60 sec	90 sec
ohne Balancing	267,6	-507,76%	-79,66%	-41,34%	-28,73%
mit Balancing	19,4	-349,80%	-268,16%	-208,62%	-167,48%

Die Werte bei 00 sec sind die Werte der Generation 0, also die Initialwerte. Hier ist das Verfahren mit Balancing besser. Danach sind die Werte ohne Anwendung der Balancing-Heuristik besser. Die TWT-Werte werden schneller besser. In der Tabelle 5.11 ich ebenso zu entnehmen, dass ohne Balancing durchschnittlich 267,6 Generationen bearbeitet wurden. Mit Balancing wurden nur etwa 19,4 Generationen bearbeitet. Wenn die Balancing-Heuristik angewendet wird, dann speichert der Dekoder den erstellten Ablaufplan, damit der Ablaufplan für die weitere Bearbeitung, das heißt der Anwendung der Balancing-Heuristik, zur Verfügung steht. Diese Speicherung kostet Rechenzeit. Damit können in der vorgegebenen maximalen Rechenzeit nur etwa ein Zehntel der Generationen erstellt werden, die in derselben Rechenzeit ohne die Anwendung der Balancing-Heuristik erstellt werden.

In der Abbildung 5.2 sind für die erste Testinstanz der Verlauf der TWT-Werte in Abhängigkeit von der Rechenzeit dargestellt.

In der Tabelle 5.12 sind TWT-Werte nach 0, 10, 20, 30 und 40 Generation ablesbar. Ohne Balancing wurde dafür etwa 9 Sekunden Rechenzeit benötigt, mit Balancing etwa 90 Sekunden.

Tabelle 5.12.: Test Balancing, Generationen

Verfahren	Gen 0	Gen 10	Gen 20	Gen 30	Gen 40
ohne Balancing	409,618	268,281	152,377	120,957	96,4213
mit Balancing	374,576	201,876	142,331	108,414	89,3907

Mit Balancing sind die TWT-Werte je Generation besser als ohne Balancing, es fordert jedoch mehr Rechenzeit. Die Anwendung der Balancing-Heuristik zeigt sich damit als ungeeignet gegenüber der Variante ohne die Anwendung dieser Balancing-Heuristik.

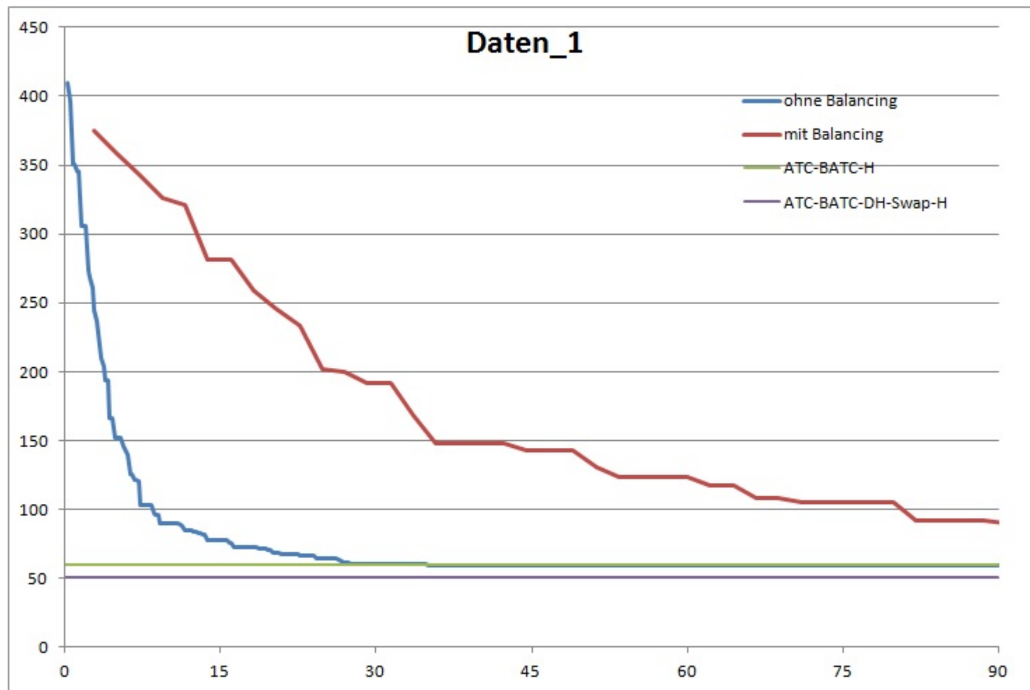

Abbildung 5.2.: Vergleich Balancing, *TWT*-Werte nach Zeit

5.3. Darstellung der numerischen Ergebnisse

Mit allen 1.440 Testinstanzen wurde ein Test mit jeweils 120 Sekunden Rechenzeit durchgeführt. Für die ersten 72 Testinstanzen wurden Tests mit einer Rechenzeit von jeweils 900 Sekunden, also 15 Minuten, durchgeführt.

5.3.1. Test mit allen Testinstanzen

Für die 1.440 Testinstanzen wurde mit dem Dekoder für die Permutationsrepräsentation das *BRKGA*-Verfahren durchgeführt. Die Populationsgröße wurde auf $p = 1.000$ gesetzt. Für jede Instanz wurde ein maximale Rechenzeit von 120 Sekunden gesetzt. Nach den 120 Sekunden wurde noch abschließend die Swap-Heuristik angewendet, um nochmals eine Verbesserung des *TWT*-Werten zu ermöglichen. Die gesamte Rechenzeit betrug damit knapp über 48 Stunden. Die Ergebnisse sind in der Tabelle 5.13 zusammen gefasst. Die Auswahl der Kriterien orientiert sich an Almeder und Mönch [1].

Als Referenzwert dient für alle Werte der *TWT*-Wert, der mit Hilfe der *ATC-BATC*-Heuristik ermittelt wurde. Es wird jeweils die relative Veränderung dargestellt. Ein positiven Prozentwert bedeutet eine Verbesserung, ein negativer Prozentwert eine Verschlechterung.

Tabelle 5.13.: Testergebnisse 1.440 Instanzen

Kriterium	Anzahl	TWT_{swap}	TWT_{Heu}	TWT_{final}	TWT_{best}	besser
alle	1.440	4,30%	4,45%	6,87%	9,50%	112
$n = 180$	480	4,10%	7,87%	8,31%	9,76%	62
$n = 240$	480	4,29%	6,09%	7,43%	9,80%	32
$n = 300$	480	4,50%	-0,62%	4,87%	8,93%	18
$B = 4$	720	4,36%	2,32%	5,96%	9,38%	38
$B = 8$	720	4,23%	6,57%	7,77%	9,62%	74
$m = 3$	360	4,38%	3,08%	5,76%	8,54%	27
$m = 4$	360	4,46%	4,22%	6,64%	9,39%	25
$m = 5$	360	4,18%	4,87%	7,20%	9,63%	32
$m = 6$	360	4,16%	5,61%	7,89%	10,44%	28
$R = 0,5, T = 0,3$	360	12,08%	15,61%	18,26%	24,08%	10
$R = 0,5, T = 0,6$	360	2,15%	2,88%	4,26%	5,53%	28
$R = 2,5, T = 0,3$	360	0,91%	0,43%	1,90%	2,97%	34
$R = 2,5, T = 0,6$	360	2,04%	-1,14%	3,07%	5,42%	40
$f = 3$	480	4,72%	3,61%	6,90%	9,71%	14
$f = 6$	480	4,40%	4,82%	7,22%	9,98%	38
$f = 12$	480	3,77%	4,90%	6,49%	8,81%	60

In der Spalte TWT_{swap} ist die Verbesserung mittels der ATC-$BATC$-DH-Swap-Heuristik (siehe Abschnitt 2.4) erzielt wurde. In der Spalte TWT_{Heu} ist die Verbesserung mittels der $BRKGA$-Methode erzielt wurde. In der Spalte TWT_{final} wurde das Ergebnis aus der $BRKGA$-Methode mit der Swap-Heuristik bearbeitet. Es wurde nochmals eine Verbesserung erzielt. Die Spalte TWT_{best} sind beste Ergebnisse für den TWT-Wert der Instanzen zum Vergleich herangezogen. Diese Werte wurden mit verschiedenen Berechnungen erzielt. In der letzten Spalte mit der Überschrift *besser* ist die Anzahl der Instanzen angegeben, bei denen eine Verbesserung des TWT-Wertes gegenüber der bisher besten Ergebnissen erzielt wurden.

Ohne die finale Anwendung der Swap-Heuristik wird im Durchschnitt bei einigen Kriterien die Verbesserung der ATC-$BATC$-DH-Swap-Heuristik nicht erreicht. Mit der Anwendung der finalen Swap-Heuristik wird dies verbessert. Im Durchschnitt werden jedoch nicht die Ergebnisse der bisher besten TWT-Werte erreicht. Allerdings werden bei 112 von 1.440 Instanzen die bisher besten Werte verbessert. Das sind bei 7,8% der Instanzen. Bei den Kriterien $n = 180$ beziehungsweise $f = 12$ sind es sogar 12,5%. Bei den 160 Instanzen mit $n = 180$ und $f = 12$ wird in 40 Fällen (also bei 25%) eine Verbesserung erzielt. Die durchschnittliche Verbesserung nach der Durchführung der finalen Swap-Heuristik von 7,94% reicht jedoch nicht ganz an die durchschnittliche Verbesserung der besten Werte (8,78%) heran.

In der Tabelle 5.14 sind die Ergebnisse nach unterschiedlichen Rechenzeiten dargestellt. In der Spalte *Gen* ist angegeben, wie viele Generationen in den 120 Se-

kunden Rechenzeit je Testinstanz erzeugt wurden. In den Spalten vier bis neun sind die Verschlechterungen beziehungsweise die Verbesserungen die unter Anwendung der $BRKGA$-Methode erzielt wurden niedergeschrieben. Hier ist die schrittweise Verbesserung zu verfolgen. In der letzten Spalte, der Spalte TWT_{final}, wurde das Ergebnis nach 120 Sekunden Rechenzeit je Testinstanz mit der finalen Swap-Heuristik dargestellt. Es wurden nochmals Verbesserungen erzielt.

Tabelle 5.14.: Testergebnisse, Generationen

Kriterium	Anzahl	Gen	15 sec	30 sec	45 sec	60 sec	90 sec	120 sec	TWT_{final}
alle	1.440	150,6	-118,16%	-50,30%	-22,15%	-9,25%	1,12%	4,45%	6,87%
$n = 180$	480	208,3	-40,18%	-4,56%	4,04%	6,45%	7,63%	7,87%	8,31%
$n = 240$	480	143,3	-110,50%	-40,34%	-13,80%	-2,96%	4,30%	6,09%	7,43%
$n = 300$	480	100,3	-203,81%	-106,00%	-56,68%	-31,25%	-8,57%	-0,62%	4,87%
$B = 4$	720	126,5	-163,30%	-75,19%	-36,59%	-18,19%	-2,81%	2,32%	5,96%
$B = 8$	720	174,7	-73,03%	-25,41%	-7,71%	-0,32%	5,05%	6,57%	7,77%
$m = 3$	360	150,8	-139,75%	-59,19%	-26,68%	-12,08%	-0,52%	3,08%	5,76%
$m = 4$	360	150,9	-121,46%	-51,66%	-22,85%	-9,56%	0,89%	4,22%	6,64%
$m = 5$	360	150,4	-114,06%	-48,79%	-21,52%	-8,75%	1,57%	4,87%	7,20%
$m = 6$	360	150,4	-97,37%	-41,56%	-17,55%	-6,63%	2,54%	5,61%	7,89%
$R = 0,5, T = 0,3$	360	150,5	-170,16%	-59,98%	-18,52%	-0,87%	12,05%	15,61%	18,26%
$R = 0,5, T = 0,6$	360	150,5	-53,36%	-23,33%	-10,25%	-4,03%	1,11%	2,88%	4,26%
$R = 2,5, T = 0,3$	360	150,7	-146,31%	-63,79%	-29,62%	-14,40%	-2,84%	0,43%	1,90%
$R = 2,5, T = 0,6$	360	150,8	-102,81%	-54,10%	-30,20%	-17,71%	-5,83%	-1,14%	3,07%
$f = 3$	480	148,6	-136,98%	-59,20%	-26,95%	-12,11%	-0,18%	3,61%	6,90%
$f = 6$	480	154,1	-117,71%	-49,64%	-21,45%	-8,70%	1,50%	4,82%	7,22%
$f = 12$	480	149,2	-99,80%	-42,06%	-18,05%	-6,95%	2,04%	4,90%	6,49%

5.3.2. Langläufer

Mit den ersten 72 Testinstanzen werden Test mit einer Laufzeit von jeweils 900 Sekunden, also 15 Minuten durchgeführt. Bei diesen Testinstanzen ist die maximale Batchgröße jeweils $B = 4$, die Anzahl der Maschinen ist $m = 3$ oder $m = 4$. Es wurde wieder der Dekoder für die Permutationsrepräsentation mit dem $BRKGA$-Verfahren durchgeführt. Die Populationsgröße wurde auf $p = 1.000$ gesetzt. In der Tabelle 5.15 sind wichtige Ergebnisse der Testläufe im Vergleich zu den Ergebnissen mit einer Rechenzeit von 120 Sekunden dargelegt. In der Spalte *Gen* ist die durchschnittliche Anzahl der Generationen, die in der Rechenzeit behandelt wurde aufgeführt. Die Spalte TWT_{Heu} stellt die durchschnittliche relative Veränderung aus dem Ergebniss der Heuristik gegenüber dem Referenzwert, dem TWT-Wert mit dem List-Scheduling-Verfahren dar. In der Spalte TWT_{final} ist die Veränderung aufgeführt, wenn abschließend der Ablaufplan mit der Swap-Heuristik bearbeitet wird. In der Spalte TWT_{best} ist die durchschnittliche Veränderung gegenüber dem Referenzwert für die bisher besten Ergebnisse für die Ablaufplanung dargestellt. In der abschließenden Spalte *besser* ist dargestellt, für wie viele Instanzen die Langläufer ein besseres Ergebnis erzielt haben als das bisher beste Ergebnis.

Tabelle 5.15.: Testergebnisse Langläufer

Rechenzeit	Gen	TWT_{Heu}	TWT_{final}	TWT_{best}	besser
120 sec	126,7	0,93%	4,96%	8,37%	2
900 sec	946,7	5,75%	6,56%	8,37%	9

Durch die Verlängerung der Laufzeit und dadurch eine Erhöhung der Anzahl der Generation wird das Ergebnis verbessert. Bei etwa 950 Generationen und einer Populationsgröße von $p = 1000$ werde etwa 750.000 Chromosome per Zufall erzeugt und ausgewertet.

5.4. Analyse und Interpretation der Ergebnisse

Die Ergebnisse in Tabelle 5.13 zeigen dass das $BRKGA$-Verfahren mit dem Dekoder für die Permutationsrepräsentation ohne die permanente Anwender der Swap-Heuristik sondern nur mit der finalen Anwendung der Swap-Heuristik in den Bereich der Werte der GA-Heuristik in der Arbeit von Almeder und Mönch [1] herankommt. Die Werte erreichen jedoch nicht ganz die Werte der ACO-Heuristik oder VNS-Heuristik, die in der Arbeit von Almeder und Mönch [1] dargestellt wurden. Es werden jedoch (nach einer Rechenzeit von 120 Sekunden je Testinstanz) für einige Instanzen bessere TWT-Werte geliefert als die bisher besten Ergebnisse.

Die guten Ergebnisse werden jedoch erst nach einer langen Rechenzeit erzielt. Das Verfahren startet von schlechten Startwerten und arbeitet sich dann langsam

in die Region vor, in welcher der TWT-Wert gemäß der ATC-$BATC$-Heuristik liegt. Die Ergebnisse in Unterabschnitt 5.2.2 zeigen, dass die Verbesserungen am Anfang sind, jedoch nach einer gewissen Zeitspanne die Veränderungen sehr träge sind, das heißt kaum noch Verbesserungen eintreten.

In Unterabschnitt 5.2.3 wurden Untersuchungen zur Populationsgröße durchgeführt. Eine große Populationsgröße führt zu einem großen Reservoir von Chromosomen. Es hat jedoch zur Folge, dass in der gleichen Zeitspanne weniger Generationen berechnet werden können.

Die Verwendung der Swap-Heuristik und der Balancing-Heuristik führt zu guten Ergebnissen, die permanente Anwendung kostet jedoch viel Rechenzeit, was in den Unterabschnitten 5.2.4 und 5.2.5 dargestellt wurde. Wenn mit der Swap-Heuristik oder der Balancing-Heuristik gute TWT-Werte erzielt werden, dann wird mit dem Chromosom vor der Anwendung der Heuristik weiter gearbeitet, jedoch nicht mit einem Chromosom welches zu dem aktualisierten Ablaufplan gehört.

Es gibt viele Parameter, die verändert werden können, die zu anderen Ergebnissen führen können. Im Rahmen dieser Arbeit wurden nur einige Werte variiert, andere Werte wurde bei allen Experimenten konstant belassen (Parameter p, p_e, p_m, ρ_e) (siehe dazu Unterabschnitt 5.2.1) .

Bei den obigen Experimenten hat sich nach einiger Zeit die Änderungsrate deutlich reduziert. Das heißt, dass die Änderungen selten kamen und gering ausfielen, obwohl es noch Potenzial für Verbesserungen gibt, da die bisher besten TWT-Werte noch nicht erreicht wurden. Hier ist zu überlegen, wie das Vorgehen dynamischer gestaltet werden kann. Einige Ideen dazu werden im Kapitel 6 angerissen.

Kapitel 6.

Zusammenfassung und Ausblick

Die in der Einleitung aufgestellten Ziele wurden erreicht. Es wurde unter Anpassung des *brkgaAPI*-Frameworks von Toso und Resende [26] ein *GA* zur Erstellung eines Ablaufplanes auf parallelen Batchmaschinen erstellt. Hierbei wurde zwei verschiedene Random-Key-Repräsentationen entwickelt und getestet. Es wurden umfangreiche numerische Experimente durchgeführt und damit verschiedene Konstellationen der Methoden und Heuristiken getestet. Damit konnten die Ergebnisse der Berechnungen mit den Ergebnisse mit einem konventionellen Verfahren, des List-Scheduling-Verfahren, wie bei Almeder und Mönch [1] verglichen werden. Daher konnten die Ergebnisse mit den Ergebnissen anderer Metaheuristiken, wie dies ebenfalls bei Almeder und Mönch [1] durchgeführt wurde, verglichen werden.

Es wurde jedoch auch gezeigt, dass es noch Potenzial gibt, um weitere Verbesserungen zu erzielen.

- Veränderung der Dekoder: Die Dekoder können so verändert werden, dass bei der Bildung der Batches nicht der erste Job, der noch nicht verplant ist die Familie bestimmt. Wie bei *ATC-BATC*-Verfahren kann je Familie der jeweils erste Batch gebildet werden und dann derjenige Batch gewählt werden, der den höchsten Wert für die gewichtete Verspätung hat. Dadurch sind die *TWT*-Werte der Ablaufpläne am Anfang besser.
- In die Startgeneration kann ein Chromosom mit einem guten *TWT*-Wert verwendet werden. Hierzu ist es notwendig, aus einem gegebenen Ablaufplan ein dazugehöriges Chromosom zu ermittelt und dieses Chromosom anschließend in die Population einzupflanzen. Für den ersten Teil sind die Dekoder dahingehend zu ergänzen, dass aus dem Ablaufplan ein Chromosom erstellt wird. Derzeit wird aus einem Chromosom ein Ablaufplan erstellt und bewertet. Aus der *ATC-BATC*-Heuristik oder der *ATC-BATC-DH*-Swap-Heuristik können gute Ablaufpläne erstellt werden. Wenn diese Ablaufpläne in die Population eingepflanzt werden werden, startet der *GA* bereits mit guten Werten starten.
- Wenn mittels der Swap-Heuristik oder der Balancing-Heuristik Ablaufpläne verbessert werden, dann kann ein Chromosom, welches der Ausgangspunkte für eine Verbesserung war, ersetzt werden durch das Chromosom,

das dem verbesserten Ablaufplan entspricht. Auch hier werden die beiden Anpassungen benötigt, die im vorherigen Punkt angesprochen wurde.

- Bei jeder Generation können einige wenige Chromosome die dazugehörigen Ablaufpläne durch die Anwendung der Swap-Heuristik verbessert werden. Das Chromosom, welches der Ausgangspunkt der Verbesserung war kann dann durch das Chromosom, welches zur Verbesserung gehört, ersetzt werden.
- Nach bestimmten Zeitabständen kann die Population geleert werden, bis auf einige wenige gute Chromosomen. Damit kann der GA wieder neu starten, mit einem guten Startwert.
- Mit den Basisparameters, insbesondere p_e, p_m und ρ_e wurden nur wenige Variationen geprüft. Welche Möglichkeiten sind hierbei noch möglich?

Diese Maßnahmen, die im Rahmen dieser Arbeit nicht umgesetzt wurden, dienen dazu, das Verfahren mit besseren Werten zu starten oder die Dynamik der Veränderungen zu stärken. Mit der Umsetzung dieser und ähnlicher Maßnahmen kann das Ziel verfolgt werden, schneller zu guten Werten und damit eventuell zu Verbesserungen der Werte gegenüber der anderen Heuristiken und Metaheuristiken erzielt werden.

Anhang A.

Verwendete Bezeichnungen

Für die Beschreibung des Problems werden die Notationen verwendet, die in der Tabelle A.1 aufgelistet sind.

Tabelle A.1.: Notationen

n	Anzahl der Jobs, die eingeplant werden sollen
j	Laufvariable für die Jobs
f	Anzahl der (inkompatiblen) Familien
s	Laufvariable für die Familien
$f(j)$	Funktion, welches die Familie von Job j bestimmt.
n_s	Anzahl der Jobs in der Familie s, die eingeplant werden sollen
	$\sum_{s=1}^{f} n_s = n$
m	Anzahl der Maschinen
i	Laufvariable für die Maschinen
w_j	Gewicht (*weight*) vom Job j
d_j	gewünschter Fertigstellungstermin (*due date*) für Job j
p_s	Bearbeitungszeit (*processing time*) für Jobs aus der Familie s
B	maximale Kapazität eines Batches (in Anzahl Jobs)
B_{ks}	Batch k der Familie s
C_j	Fertigstellungstermin (*completion time*) für den Job j
T_j	Verspätung (*tardiness*) vom Job j
	$T_j = (C_j - d_j)^+ = \max(0, C_j - d_j)$
w_jT_j	gewichtete Verspätung (*weighted tardiness*) von Job j
TWT	total gewichtete Verspätung (*total weighted tardiness*)
	$TWT = \sum_{j=1}^{n} w_jT_j$

Hierbei ist $x^+ = max(0, x)$ für alle $x \in \mathbb{R}$.

Anhang B.

Übersicht Programme

Auf der beigefügten CD (siehe auch Anhang C) sind die Quellcodes der Implementierung enthalten. Die Quellcodes sind in drei Verzeichnissen und dem Hauptprogramm aufgeteilt.

Die Datei `mainframe.cpp` enthält das Programm, das aufgerufen wird, das die Steuerung übernimmt.

B.1. Verzeichnis `util`

Im Verzeichnis `util` sind problemunabhängige Hilfsroutinen implementiert.

- `Frame.h / Frame.cpp`
  Dieser Teil stellt Routinen für einen Programmrahmen zur Verfügung.
- `Logger.h / Logger.cpp`
  Dieser Teil ermöglicht die Ausgabe von Informationen auf eine Logging-Datei.
- `Parameters.h / Parameters.cpp`
  Dieser Teil steuert das Einlesen von Parametern aus einer externen Datei. Die Daten werden dabei als Key-Value-Paare als Zeichenketten dargestellt. Zwischen der Zeichenkette für den *Key* und der Zeichenkette für den *Value* ist ein Gleichheitszeichen. Die Routine kann eine Zeichenkette auch in eine ganze Zahl (*int*-Wert), eine reelle Zahl (*double*-Wert) oder in einen Wahrheitswert (*bool*-Wert) umwandeln.
- `StopWatch.h / StopWatch.cpp`
  Es wird eine Klasse für eine Stoppuhr implementiert. Damit können Zeitabstände gemessen werden. Es können mehrere Stoppuhren aktiviert werden.
- `Util.h / Util.cpp`
  In diesem Modul sind verschiedene Hilfswerkzeuge implementiert, insbesondere für die Konvertierung von Zahlen in Zeichenketten und umgekehrt. Ebenso werden Routinen zur Bearbeitung von Zeichenketten bereits gestellt.

B.2. Verzeichnis `brkgaAPI`

Im Verzeichnis `brkgaAPI` sind die angepassten Programmroutinen des *brkgaAPI*-Framework gemäß Toso und Resende [26] hinterlegt. Details hierzu sind im Abschnitt 4.1 beschrieben.

B.3. Verzeichnis `scheduling`

Im Verzeichnis `scheduling` sind die problemabhängigen Routinen implementiert. Es können dabei vier Gruppen von Routinen unterschieden werden.

B.3.1. Datenstruktur

- `JobData.h / JobData.cpp`
  Im Teil *JobData* ist eine Klasse implementiert, welche die Daten für einen Job aufnimmt.
- `JobDValue.h / JobDValue.cpp`
  Im Teil *JobDValue* ist ein Paar definiert. Es besteht aus einem *int*-Wert, welches den Verweis auf einen Index oder eine Jobnummer sein kann, und einem *doube*-Wert :

  ```
  std::pair<int,double> .
  ```

  Innerhalb der Anwendung werden an einigen Stellen Vektoren mit solchen Paaren verwendet. Beispielsweise werden Jobs (Ident *int*-Wert) nach einem *double*-Wert sortiert verwendet.
- `batch.h / batch.cpp` Im Teil *Batch* ist die Datenstruktur für einen Batch implementiert. Die Ablaufplanung ist eine Ablaufplanung für Batches, in denen die Jobs in den Batchen enthalten sind.

B.3.2. Decoder

- `Decoder.h / Decoder.cpp`
  Abstrakte Klasse für die Dekoder. Details hierzu sind in Abschnitt 4.2.1 dargestellt
- `Decoder1.h / Decoder1.cpp`
  Konkrete Klasse, abgeleitet von `Decoder` für den Dekoder der Permutationsrepräsentatation.

- `Decoder2.h / Decoder2.cpp`
  Konkrete Klasse, abgeleitet von `Decoder` für den Dekoder der Repräsentation für simultane Zuordnungen-Reihenfolge-Entscheidungen.

B.3.3. Heuristiken

- `AtcBatcH.h / AtcBatcH.cpp`
  Implementierung der *ATC-BATC*-Heuristik, siehe Abschnitt 2.4.1.
- `BalancingH.h / BalancingH.cpp`
  Implementierung der Balancing-Heuristik, siehe Abschnitt 3.3.2.
- `DH.h / DH.cpp`
  Implementierung der Dekompositions-Heuristik, siehe Abschnitt 2.4.2.
- `SwapH.h / Swaph.cpp`
  Implementierung der Swap-Heuristik, siehe Abschnitt 2.4.3.

B.3.4. Sonstiges

- `ProblemInstanz.h / ProblemInstanz.cpp`
  In dieser Klasse wird das Einlesen der Daten einer Instanz aus einer Textdatei gesteuert. Die Daten werden gespeichert und für den weiteren Verlauf des Programmes bereit gestellt.
- `SchedulingParallel.h / SchedulingParallel.cpp`
  Steuerung des Ablaufes für die Erstellung eines Ablaufplanes auf parallelen Batchmaschinen mittels der *ATC-BATC-DH*-Swap-Heuristik.
- `Scheduling.h / Scheduling.cpp`
  Verschiedene Definition von Datenstrukturen und Steuerung von verschiedenen Abläufen.

Anhang C.

Inhalte der CD

Auf der CD sind folgende Daten vorhanden

C.1. Ausarbeitung

Im Verzeichnis `Ausarbeitung` ist die Ausarbeitung der Masterarbeit als PDF-Datei enthalten.

C.2. Quellcode

In diesem Verzeichnis sind die Quellcodes der Implementierung enthalten. Die Übersicht der Quellcodes ist im Anhang B) aufgeführt.

C.3. Ergebnisse

In diesem Verzeichnis sind einige Ergebnisdaten enthalten. Sowohl die Ausgaben der Programme, als auch Auswertungen mit Hilfe eines Tabellenkalkulationsprogrammes. Dabei sind folgende Unterverzeichnisse enthalten

- `01_Basisparameter`
  Auswertungen für die Untersuchungen der Wahl der Basisparameter p_e, p_m und ρ_e (siehe Unterabschnitt 5.2.1).
- `02_MethodeDekoder`
  Auswertungen für die Untersuchungen der Wahl der Methode ($RKGA$ oder $BRKGA$) und des Dekoders (siehe Unterabschnitt 5.2.2).
- `03_Populationsgröße`
  Auswertungen für die Untersuchungen der Populationsgröße (siehe Unterabschnitt 5.2.3).

- `04_Swap`
  Auswertungen für die Untersuchungen, des Unterschiedes, ob die Swap-Heuristik (permanent) angewendet wird oder nicht (siehe Unterabschnitt 5.2.4).
- `05_Balancing`
  Auswertungen für die Untersuchungen der Auswirkung der Balancing-Heuristik (siehe Unterabschnitt 5.2.5).
- `11_Testergebnisse`
  In diesem Verzeichnis sind Auswertungen enthalten, die im Unterabschnitt 5.3.1 komprimiert dargestellt werden. Es sind die Testergebnisse für alle 1.440 gegebenen Testinstanzen.
- `12_Langläufer`
  In diesem Verzeichnis sind Auswertungen enthalten, die im Unterabschnitt 5.3.2 beschrieben werden. Hier sind 72 Testinstanzen mit jeweils 15 Minuten Rechenzeit bearbeitet worden.

C.4. Visual Studio C++

In der Datei `GAExperiment_2015-12-27.zip` ist die Entwicklungsumgebung als Visual C++ 2005 Express Projekt hinterlegt.

Abkürzungen

ACO	Ant Colony Optimization (Ameinsenkolonieoptimierung)
ATC	Apparent Tardiness Cost (Prioritätsregel)
BATC	Batching Apparent Tardiness Cost (Prioritätsregel für die Bildung von Batches)
BRKGA	Biased Random-Key Genetic Algorithm
CD	Compact Disc
DH	Decomposition Heuristic (Dekompositionsheuristik)
IC	Integrated Circuit (integrierter Schaltkreis)
GA	Genetic Algorithm (Genetischer Algorithmus)
kGA	kanonischer GA
PDF	Portable Document Format
TSP	Travelling Salesman Problem
TWT	Total Weighted Tardiness (totale gewichtete Verspätung)
RKGA	Random-Key Genetic Algorithm
VNS	Variable Neighbourhood Search (Variable Nachbarschafts-suche)
wT	weighted Tardiness (gewichtete Verspätung)

Liste der Algorithmen

2.1. Gesamtablauf Referenz . . . 9
2.2. ATC-BATC-Heuristik . . . 10
2.3. Dekompositionsheuristik . . . 12
2.4. Swap-Heuristik . . . 13
2.5. Swap-Heuristik, überprüfe Batch . . . 14
2.6. Swap-Heuristik, überprüfe Job . . . 15

3.1. Kanonischer GA . . . 19
3.2. Crossover . . . 20
3.3. führe Mutation durch . . . 21
3.4. BRKGA . . . 26
3.5. Dekoder Permutationsrepräsentation . . . 28
3.6. Dekoder simultane Entscheidungen . . . 31
3.7. Balancing-Heuristik . . . 35

Abbildungsverzeichnis

3.1. Übergang von einer Generation zur nächsten 24
3.2. Ablaufdiagram BRKGA 27
3.3. Ablaufplan Beispiel Dekoder 1 30
3.4. Ablaufplan Beispiel Dekoder 2 33
3.5. Ablaufplan Beispiel Dekoder 2 nach Balancing-Heuristik 35

4.1. Klassendiagramm Dekoder 43

5.1. Vergleich *BRKGA* und *RKGA*, *TWT*-Werte nach Zeit 51
5.2. Vergleich Balancing, *TWT*-Werte nach Zeit 56

Tabellenverzeichnis

3.1. Beispiel Crossover 24
3.2. Probleminstanz für Beispiel 28
3.3. Modifiziertes Chromosom 33

4.1. Komponenten des *brkgaAPI*-Frameworks 37

5.1. Gestaltung der Versuchsdaten 46
5.2. Beispiele für Bereich Fälligkeitstermine 47
5.3. Parameter für *BRKGA* 49
5.4. Vergleichsergebnisse p_e, p_m und ρ_e 49
5.5. Vergleichsergebnisse p_e, p_m und ρ_e bei 120 Sekunden 49
5.6. Vergleich Methode / Dekoder 50
5.7. Vergleich *BRKGA* besser als *RKGA* 52
5.8. Vergleich Populationsgröße 52
5.9. Vergleich Populationsgröße (2) 53
5.10. Test Swap-Heuritik 54
5.11. Test Balancing-Heuristik 55
5.12. Test Balancing, Generationen 55
5.13. Testergebnisse 1.440 Instanzen 57
5.14. Testergebnisse, Generationen 59
5.15. Testergebnisse Langläufer 60

A.1. Notationen 64

Literatur

[1] Christian Almeder und Lars Mönch. »Metaheuristics for scheduling jobs with incompatible families on parallel batching machines«. In: *Journal of the Operational Research Society* 62.12 (2010), S. 2083–2096.

[2] Hari Balasubramanian, Lars Mönch, John Fowler und Michele Pfund. »Genetic algorithm based scheduling of parallel batch machines with incompatible job families to minimize total weighted tardiness«. In: *International Journal of Production Research* 42.8 (2004), S. 1621–1638.

[3] James C. Bean. »Genetic Algorithms and Random Keys for Sequencing and Optimization«. In: *ORSA Journal on Computing* 6.2 (1994), S. 154–160.

[4] Peter Brucker. *Scheduling algorithms*. 5th ed. Berlin und New York: Springer, 2007.

[5] Peter Brucker, Andrei Gladky, Han Hoogeveen, Mikhail Y. Kovalyov, Chris N. Potts, Thomas Tautenhahn und van de Velde, Steef L. »Scheduling a batching machine«. In: *Journal of Scheduling* 1.1 (1998), S. 31–54.

[6] Stéphane Dauzère-Pérès und Lars Mönch. »Scheduling jobs on a single batch processing machine with incompatible job families and weighted number of tardy jobs objective«. In: *Computers & Operations Research* 40.5 (2013), S. 1224–1233.

[7] Amit Devpura, John W. Fowler, W. Matthew Carlyle und Imelda C. Perez. »Minimizing Total Weighted Tardiness on Single Batch Process Machine with Incompatible Job Families«. In: *Operations Research Proceedings 2000*. Hrsg. von Bernhard Fleischmann, Rainer Lasch, Ulrich Derigs, Wolfgang Domschke und Ulrich Rieder. Bd. 2000. Operations Research Proceedings. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, S. 366–371.

[8] Ingrid Gerdes, Frank Klawonn und Rudolf Kruse. *Evolutionäre Algorithmen: Genetische Algorithmen - Strategien und Optimierungsverfahren - Beispielanwendungen ; [mit Online-Service zum Buch]*. 1. Aufl. Computational intelligence. Wiesbaden: Vieweg, 2004.

[9] David E. Goldberg. *Genetic algorithms in search, optimization and machine learning*. New Delhi u.a.: Pearson, 2006.

[10] José F. Gonçalves und Mauricio G. C. Resende. »Biased random-key genetic algorithms for combinatorial optimization«. In: *Journal of Heuristics* 17.5 (2011), S. 487–525.

[11] José F. Gonçalves, Mauricio G. C. Resende und Rodrigo F. Toso. »Biased and unbiased random key genetic algorithms: An experimental analysis«. In: *Proceedings of the 10th Metaheuristics International Conference*. Hrsg. von H. C. Lau, P. Van Hentenryck und Günther R. Raidl. Springer, 2013.

[12] Ronald L. Graham, Eugene L. Lawler, Lenstra, Jan Karel und Alexander H. G. Rinnooy Kan. »Optimization and approximation in deterministic sequencing and scheduling«. In: *Ann. Dis. Math.* 5 (1979), S. 287–326.

[13] John H. Holland. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence*. 1st MIT Press ed. Complex adaptive systems. Cambridge, Mass.: MIT Press, 1992.

[14] Andreas Klemmt. *Ablaufplanung in der Halbleiter- und Elektronikproduktion: Hybride Optimierungsverfahren und Dekompositionstechniken*. SpringerLink : Bücher. Wiesbaden: Vieweg+Teubner Verlag, 2012.

[15] Eugene L. Lawler. »A "Pseudopolynomial" Algorithm for Sequencing Jobs to Minimize Total Tardiness«. In: *Studies in Integer Programming*. Bd. 1. Annals of Discrete Mathematics. 1977, S. 331–342.

[16] Muthu Mathirajan und Appa Iyer Sivakumar. »A literature review, classification and simple meta-analysis on scheduling of batch processors in semiconductor«. In: *The International Journal of Advanced Manufacturing Technology* 29.9-10 (2006), S. 990–1001.

[17] Makoto Matsumoto und Takuji Nishimura. »Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator«. In: *ACM Transactions on Modeling and Computer Simulation* 8.1 (1998), S. 3–30.

[18] Sanjay V. Mehta und Reha Uzsoy. »Minimizing total tardiness on a batch processing machine with incompatible job families«. In: *IIE Transactions* 30.2 (1998), S. 165–178.

[19] Zbigniew Michalewicz. *Genetic algorithms + data structures: Evolution programs*. 3rd rev. and extended ed. Berlin und New York: Springer-Verlag, 1996.

[20] Zbigniew Michalewicz und David B. Fogel. *How to solve it: modern heuristics: With 7 tables*. 2., rev. and ext. ed. Berlin und Heidelberg [u.a.]: Springer, 2004.

[21] Lars Mönch, Hari Balasubramanian, John W. Fowler und Michele E. Pfund. »Heuristic scheduling of jobs on parallel batch machines with incompatible job families and unequal ready times«. In: *Computers & Operations Research* 32.11 (2005), S. 2731–2750.

[22] Lars Mönch, John W. Fowler, Stéphane Dauzère-Pérès, Scott J. Mason und Oliver Rose. »A survey of problems, solution techniques, and future challenges in scheduling semiconductor manufacturing operations«. In: *Journal of Scheduling* 14.6 (2011), S. 583–599.

[23] Imelda C. Perez, John W. Fowler und W. Matthew Carleyle. »Minimizing total weighted tardiness on a single batch process machine with incompatible job families.« In: *Computer & Operations Research* 32 (2005), S. 327–341.

[24] Michael Pinedo. *Scheduling: Theory, algorithms, and systems*. 4th ed. New York: Springer, 2012.

[25] Chris N. Potts und Mikhail Y. Kovalyov. »Scheduling with batching: A review«. In: *European Journal of Operational Research* 120.2 (2000), S. 228–249.

[26] Rodrigo F. Toso und Mauricio G. C. Resende. »A C++application programming interface for biased random-key genetic algorithms«. In: *Optimization Methods and Software* 30.1 (2014), S. 81–93.

[27] Reha Uzsoy, Chung-Yee Lee und Louis A. Martin-Vega. »A Review of Production Planning and Scheduling Models in the Semiconductor Industry Part II: Shop-Floor Control«. In: *IIE Transactions* 26.5 (1994), S. 44–55.

Eidesstattliche Erklärung

Name des Studierenden:	Andreas Zeh-Marschke
Matrikelnummer:	3016641

Hiermit erkläre ich, dass ich die vorliegende Masterarbeit selbständig verfasst und noch nicht anderweitig für Prüfungszwecke vorgelegt habe. Andere als die angegebenen Quellen und Hilfsmittel habe ich nicht benutzt, wörtliche und sinngemäße Zitate wurden als solche gekennzeichnet.

Eggenstein-Leopoldshafen, den 28.12.2015