TECHNISCHE UNIVERSITÄT WIEN

Diplomarbeit

Simulationsgestützte Produktionsfeinplanung mittels kombinatorischer Optimierungsalgorithmen

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines

Diplom-Ingenieurs

unter der Leitung von

Univ.-Prof. Dr.-Ing. Dipl. Wirt.-Ing. Prof. eh. Dr. h.c. Wilfried Sihn
(E330 Institut für Managementwissenschaften, Bereich: Betriebstechnik und Systemplanung)

Proj.-Ass. Dipl. Wirt.-Ing. Thomas Sobottka
(E330 Institut für Managementwissenschaften, Bereich: Betriebstechnik und Systemplanung
Fraunhofer Austria Research GmbH)

eingereicht an der Technischen Universität Wien

Fakultät für Maschinenwesen und Betriebswissenschaften

von

Philipp Sumereder BSc
0928863 (066.482)
Spengergasse 27/ 2.904.1
1050 Wien

Wien, im März 2017

_____

Philipp Sumereder

TECHNISCHE
UNIVERSITÄT
WIEN

Ich habe zur Kenntnis genommen, dass ich zur Drucklegung meiner Arbeit unter der Bezeichnung

Diplomarbeit

nur mit Bewilligung der Prüfungskommission berechtigt bin.

Ich erkläre weiters Eides statt, dass ich meine Diplomarbeit nach den anerkannten Grundsätzen für wissenschaftliche Abhandlungen selbstständig ausgeführt habe und alle verwendeten Hilfsmittel, insbesondere die zugrunde gelegte Literatur, genannt habe.

Weiters erkläre ich, dass ich dieses Diplomarbeitsthema bisher weder im In- noch Ausland (einer Beurteilerin/einem Beurteiler zur Begutachtung) in irgendeiner Form als Prüfungsarbeit vorgelegt habe und dass diese Arbeit mit der vom Begutachter beurteilten Arbeit übereinstimmt.

Wien, im Monat 2016

_____

Philipp Sumereder

Danksagung

An dieser Stelle möchte ich mich bei Herrn Professor Sihn sowie bei Herrn Dipl. Wirt.-Ing. Thomas Sobottka und Herrn Dipl.-Ing. Felix Kamhuber für die hervorragende Betreuung dieser Arbeit bedanken.

Darüber hinaus möchte ich mich bei meiner Familie für Ihre permanente Unterstützung und die Ermöglichung meines Studiums bedanken.

Ein besonderer Dank gilt auch meiner Freundin Dipl.-Ing. Bianca Patru für ihre moralische und fachliche Unterstützung bei der Erstellung dieser Arbeit.

Kurzfassung

Das Betreiben von Produktionsanlagen mit der höchstmöglichen Effizienz ist für die meisten Unternehmen eine absolute Notwendigkeit, um auf den Märkten bestehen zu können. Um dies zu erreichen, werden unter anderem die Methoden und Werkzeuge des Operations Research eingesetzt. So ist die Simulation von Produktions- und Logistikprozessen seit Jahrzehnten ein wichtiges Hilfsmittel für die Entscheidungsfindung bei der Planung und Steuerung von Produktionsanlagen. Die systematische Manipulation von Stellgrößen solcher Simulation, um definierte Ziele zu erreichen, führt oft zu sehr komplexen Optimierungsproblemen, welche nicht mehr mit exakten Methoden lösbar sind. Diese Arbeit befasst sich mit Optimierungsalgorithmen, die in der Lage sind, Probleme dieser Art zu lösen.

Das Ziel des ersten Teils der Arbeit ist es, geeignete Optimierungsalgorithmen für ein Anwendungsbeispiel, das dem Projekt Balanced Manufacturing der Technischen Universität Wien entspringt, zu identifizieren. Dazu wird erst ein Überblick über Optimierungsalgorithmen für simulationsbasierte Optimierung gegeben. Anschließend werden das Simulationsmodell und das entsprechende Optimierungsproblem näher erläutert und im Rahmen des Operations Research klassifiziert. Basierend auf dieser Einordnung erfolgt eine Evaluierung und Auswahl geeigneter Algorithmen.

Im zweiten Teil der Arbeit werden zwei Algorithmen (Particle Swarm Optimization und Genetischer Algorithmus) anhand des vorliegenden Optimierungsproblems hinsichtlich Effektivität, Effizienz und Lösungsqualität verglichen. Zudem werden für ein bereits vorhandenes Optimierungsmodul, das auf einem genetischen Algorithmus basiert, diverse Einstellungen und Operatoren getestet, mit dem Ziel, dessen Leistungsfähigkeit zu erhöhen. Durch einige der getesteten Maßnahmen werden signifikante Verbesserungen erzielt.

Abstract

The operation of production facilities in the most efficient way is a necessity for most companies to be competitive on the markets. To achieve this goal, the tools and methods of operations research are applied among others. The simulation of production and logistic processes is for example an important means for decision making at planning and controlling production plants. The systematic alteration of input variables of such simulations to reach certain goals often leads to highly complex optimization problems, which cannot be solved with exact methods. This work deals with optimization algorithms which can solve such problems.

The aim of the first part of this work is to find suitable optimization algorithms for a use case originating the Technical University Vienna Balanced Manufacturing project. For this purpose, an overview of available optimization algorithms for simulation optimization is given. Afterwards the simulation model and the corresponding optimization problem are described in detail and classified in terms of operations research. Based on that classification appropriate algorithms are evaluated and selected.

In the second part of this work two algorithms (particle swarm optimization and genetic algorithm) are being compared on the problem regarding efficiency and effectivity. Furthermore, several setups and options are being tested for an existing optimization module, which is based on a genetic algorithm, to increase its performance. Some of these adaptions tested increase the algorithms performance significantly.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Simulation ist ein in Produktion und Logistik häufig eingesetztes Werkzeug für die Abbildung komplexer Prozesse, da eine mathematische Beschreibung dieser Systeme oft sehr schwierig ist. Die Durchführung einer Simulationsstudie beinhaltet in der Regel die Variation von Eingangsparametern des Simulationsmodells, um zu Handlungsempfehlungen für das reale System zu kommen [57]. Dies geschieht häufig, indem einige Varianten erstellt und anschließend die am besten scheinende Alternative ausgewählt wird. Im Gegensatz zu diesem eher erfahrungsbasierten Ansatz erfolgt bei einer Optimierung im Sinne des Operations Research erst eine genaue Definition des Ziels bzw. der Ziele der Optimierung. Im Falle von Produktionssystemen sollen oft mehrere, teils konkurrierende Ziele möglichst gut erreicht werden. Beispiele hierfür sind eine hohe Auslastung von Kapazitäten gegen eine kurze Durchlaufzeit oder niedrige Bestände gegen hohe Liefertreue. Dieses Zielsystem wird mittels einer Zielfunktion mathematisch beschrieben, welche zur Bewertung von Lösungen herangezogen wird. Die Aufgabe der Optimierung besteht in der Minimierung bzw. Maximierung dieser Zielfunktion. Die Simulation dient somit zur Bewertung von Lösungsvorschlägen der Optimierung [30].

Viele diese Optimierungsprobleme sind NP-schwer. Das bedeutet, dass die benötigte Zeit für die Lösung exponentiell mit der Problemgröße anwächst. Dies hat zur Folge, dass NP-schwere Probleme mit vertretbarem Zeitaufwand nicht exakt lösbar sind [52]. Mit Heuristiken können aber näherungsweise Lösungen in beschränkter Zeit gefunden werden, was vor allem für die Anwendbarkeit der simulationsbasierten Optimierung im täglichen Betrieb, z.B. für die Erstellung von Ablaufplänen oder die Erstellung von Maschinenbelegungsplänen, von Bedeutung ist.

Optimierungsalgorithmen für die simulationsbasierte Optimierung (SBO) kann man laut Fu [16] nach Funktionsweise in fünf Kategorien einteilen:

1. Gradientenbasierte Methoden
2. Response Surface Methoden
3. Random Search
4. Sample Path Optimierung

5. Metaheuristiken

Diese Kategorien sowie deren wichtigste Vertreter werden in Kapitel 2 näher erläutert. Ziel des ersten Teils dieser Arbeit ist die Evaluierung und Auswahl geeigneter Algorithmen aus den oben angeführten Klassen für ein Anwendungsbeispiel, welches dem Forschungsprojekt BaMa (Balanced Manufacturing) der Technischen Universität Wien entspringt. Das bereits vorhandene Simulationsmodell sowie das entsprechende Optimierungsproblem werden erst klassifiziert. Geeignete Algorithmen sollen durch eine Literaturrecherche und den Vergleich mit ähnlichen Beispielen ermittelt werden. Im zweiten Teil der Arbeit werden zwei populationsbasierte Optimierungsalgorithmen (Genetischer Algorithmus (GA) und Particle Swarm Optimization(PSO)), bezüglich Eignung für das Problem, Leistungsfähigkeit und Qualität der Lösungen, verglichen. Anschließend werden verschiedene Varianten des genetischen Algorithmus getestet und an das Anwendungsbeispiel angepasst, mit dem Ziel, eine bestmögliche Optimierungsmethode zu erhalten.

1.2 Problemstellung (Optimierungsproblem)

Diese Arbeit baut auf einem bereits vorhandenem, hybriden Simulationsmodell einer Produktionslinie für Backwaren auf. Das vereinfachte Modell einer realen Produktionsanlage bildet, neben Materialflüssen, auch den Energieverbrauch und das thermisch-physikalische Verhalten der Aggregate ab. Es beinhaltet neben diskreten auch kontinuierliche Zustandsvariablen und ist daher ein hybrides Modell.

Ziel dieser Arbeit ist die Optimierung der Produktionsplanung basierend auf dem Simulationsmodell und in diesem Zusammenhang die Auswahl und Anpassung geeigneter Optimierungsalgorithmen. Zur Bewertung von Lösungsvorschlägen der Optimierung wird eine Zielfunktion verwendet, welche aus mehreren gewichteten Teilzielen (Gesamtdurchlaufzeit, Energiekosten, Lieferverzug und Lagerhaltungskosten sowie der Differenz zwischen Soll- und Ist-Produktionsmenge) besteht. Das Ziel der Optimierung ist die Minimierung dieser Funktion. Die Stellgrößen dafür sind die Einsteuerungszeitpunkte sowie die Reihenfolge der Produktionslose und die An- und Abschaltzeitpunkte der Aggregate. Das Beispiel wird in Kapitel 2.2 ausführlicher beschrieben.

1.3 Forschungsfragen

Aus der oben beschriebenen Problemstellung leiten sich folgende Forschungsfragen ab:

1. Wie ist das Optimierungsproblem im Rahmen des Operations Management einzuordnen?
2. Welche Algorithmen sind für das vorliegende Optimierungsproblem am besten geeignet?
3. Welcher der beiden populationsbasierten Optimierungsalgorithmen GA und PSO eignet sich hinsichtlich Leistungsfähigkeit (Schnelligkeit der Lösungsfindung und Lösungsqualität) besser für das Problem?
4. Wie kann der genetische Algorithmus parametrisiert und konfiguriert werden, damit er für die vorliegende Problemkategorie bestmöglich optimierte Lösungen erzeugt?
    - Welche Kombination von GA-Operatoren ist am leistungsfähigsten?
    - Wie wirken sich unterschiedliche Parameter auf die Leistungsfähigkeit des GA aus?
    - Durch welche zusätzlichen Nebenbedingungen der Optimierung kann der Suchraum eingeschränkt werden, um die Effizienz der Lösungsfindung zu erhöhen?
    - Führt eine Hybridisierung mit einer lokalen Suchheuristik zu einer Verbesserung der Performance?

2 Simulationsgestützte Optimierung von Produktionsprozessen

2.1 Grundbegriffe

2.1.1 Systeme, Modelle und Simulation

System. Law und Kelton definieren ein System folgendermaßen:

"A system is defined to be a collection of entities, e.g., people or machines, that act and interact together toward the accomplishment of some logical end" [33]

Ein System besteht also aus Elementen, die interagieren um einen gewissen Zweck zu erfüllen. Weiters ist es durch seine Systemgrenze von der Umwelt abgegrenzt. Besteht eine Interaktion des Systems mit seiner Umwelt spricht man von einem offenen System, andernfalls von einem geschlossenem [6]. Ein Beispiel für ein System ist eine Fabrik. Die Maschinen und Personen interagieren miteinander, um Güter zu produzieren. Das Fabrik-System muss außerdem mit Lieferanten, Kunden und vielen weiteren Elementen außerhalb der Fabrik (also seiner Umwelt) interagieren, um zu funktionieren.

Modell eines Systems. Meist ist es unzweckmäßig oder unmöglich Experimente an realen Systemen durchzuführen, daher wird ein Modell als Abbild der Wirklichkeit erstellt. Ein Modell ist daher definiert als Abbildung eines Systems zum Zwecke der Studie des Systems [6]. Die Möglichkeiten dafür sind in Abbildung 2.1 dargestellt.

Physikalische vs. Mathematische Modelle. Gegenstand von Untersuchungen im Rahmen des Operations Research sind meist mathematische Modelle, die das zugrundeliegende System mittels logischer und quantitativer Zusammenhänge abbilden. Diese werden anschließend manipuliert, um zu untersuchen, wie sich das System verhalten würde. Physikalische Modelle kommen in diesem Zusammenhang eher selten zum Einsatz.

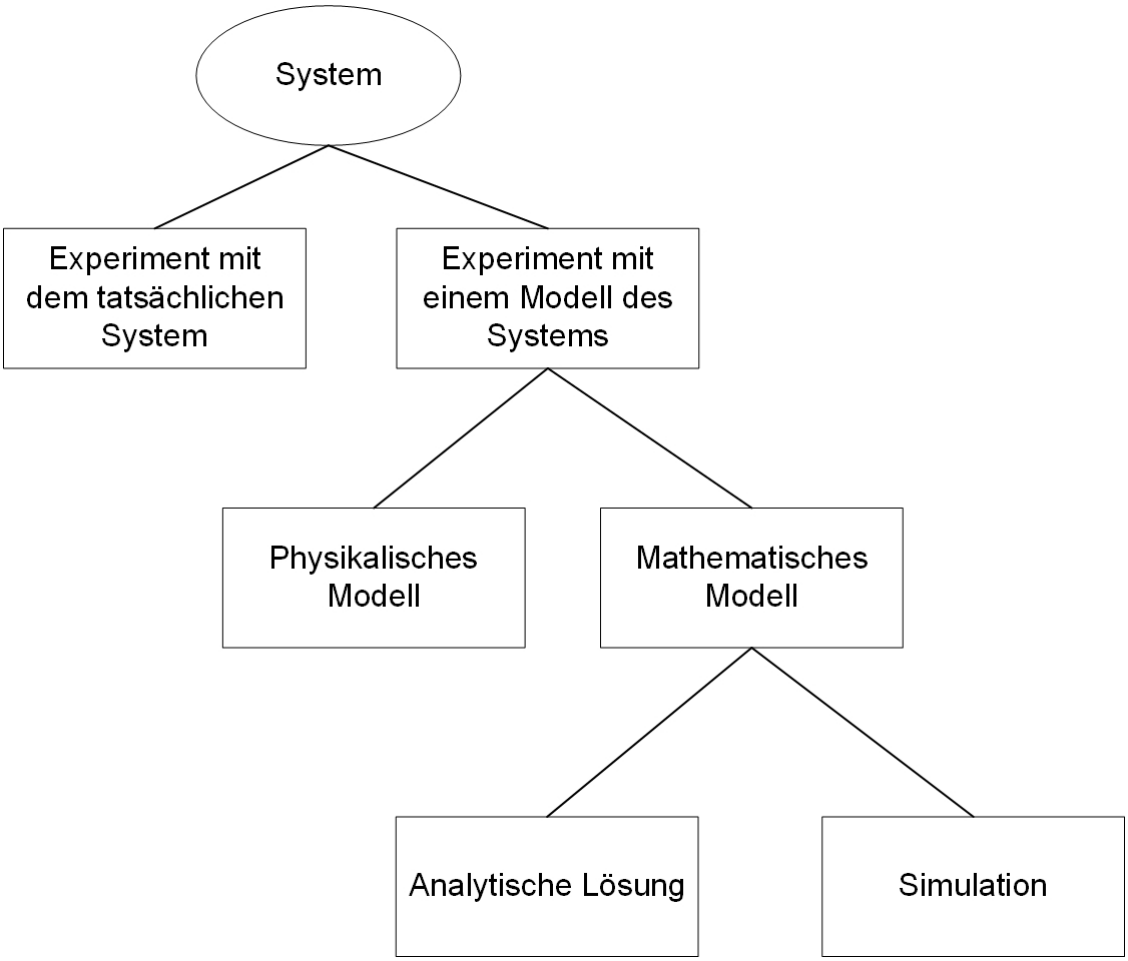

Abbildung 2.1: Möglichkeiten zur Untersuchung eines Systems [28]

Analytische Lösung vs. Simulation. Ist das untersuchte mathematische Modell einfach genug, kann man exakte, analytische Lösungen bestimmen. Oft sind die betrachteten Modelle aber sehr komplex und eine analytische Lösung ist nicht in akzeptabler Rechenzeit möglich, oder gar nicht zu bestimmen. In diesem Fall ist es zweckmäßig das System mittels Simulation zu untersuchen. Das bedeutet, dass das mathematische Modell mit den Eingangsparametern numerisch ausgeführt wird, um den Einfluss auf Ausgangsparameter zu untersuchen [6].

Modellarten. Die Klassifizierung eines Simulationsmodells kann in drei verschiedenen Dimensionen erfolgen [33]:

- Statische vs. Dynamische Modelle. Ein statisches Simulationsmodell repräsentiert ein System zu einem bestimmten Zeitpunkt oder ein System, in welchem die Zeit keine Rolle spielt. Beispiele für statische Modelle sind Monte Carlo Modelle. Dynamische Modelle repräsentieren Systeme mit zeitabhängigen Zuständen z.B.

das Modell eines Förderbandes in einer Fabrik.

- Deterministische vs. Stochastische Modelle. Deterministische Modelle sind Modelle ohne Zufallseinfluss. Der Output eines deterministischen Systems ist eindeutig definiert, sobald alle Eingangsgrößen definiert sind. Stochastische Modelle haben zumindest eine vom Zufall abhängige Eingangsgröße. Somit ist auch der Output des Modells eine Zufallsgröße.
- Kontinuierliche vs. Diskrete Modelle. In einem kontinuierlichen Modell ändern sich die Zustandsvariablen kontinuierlich mit der Zeit. Ein Beispiel dafür ist die Temperatur eines Ofens beim Aufheizvorgang. In einem diskreten Modell ändern sich die Zustandsvariablen ohne Verzögerung zu bestimmten Zeitpunkten. Ein Beispiel dafür ist die Anzahl von Produkten auf einem Förderband.

2.1.2 Optimierung

Alle mathematischen Verfahren der Optimierung zielen darauf ab, neue Suchschritte in Richtung des Maximums oder Minimums eines vorgegebenen analytischen Ziels unter den gegebenen Nebenbedingungen zu finden. Das globale Optimum (bei exakten Verfahren) bzw. dessen Näherung (bei Heuristiken) sollte dabei mit hinreichender Genauigkeit nach möglichst wenigen Suchschritten gefunden werden. Bei der Optimierung von Simulationsmodellen geht es darum, jene Kombination von Eingangsparametern aus einer Vielzahl von Kombinationen zu finden, welche zu einer optimalen Lösung führt. Die Eingangsparameter sind also Entscheidungsvariablen. Die Ausgangsparameter der Simulation werden verwendet, um die Güte von Lösungen mithilfe von Zielfunktionen zu ermitteln. Das Ziel der simulationsbasierten Optimierung stellt die Minimierung bzw. Maximierung dieser Funktion dar [30].

Im Kontext der simulationsbasierten Optimierung kann ein Simulationsmodell also als eine (unbekannte) Funktion betrachtet werden, welche Eingangsparameter in Ausgangsgrößen transformiert. Diese ist oft nichtlinear, nicht differenzierbar und weist multiple lokale Minima auf, die von Optimierungsalgorithmen überwunden werden müssen, um (näherungsweise) optimale Lösungen zu finden. Ein Beispiel einer Funktion mit mehreren lokalen Extremwerten und einem globalen Minimum (rote Markierung) ist in Abbildung 2.2 dargestellt.

Neben Optimierungsmethoden, die sich Informationen in den Ausgangsgrößen zunutze

machen wie z.B. durch die Erstellung von Response Surfaces, Metamodellen oder durch das Approximieren von Gradienten, gibt es auch sogenannte Black-Box Methoden, welche ohne Wissen über die zugrundeliegende Funktion auskommen [5].

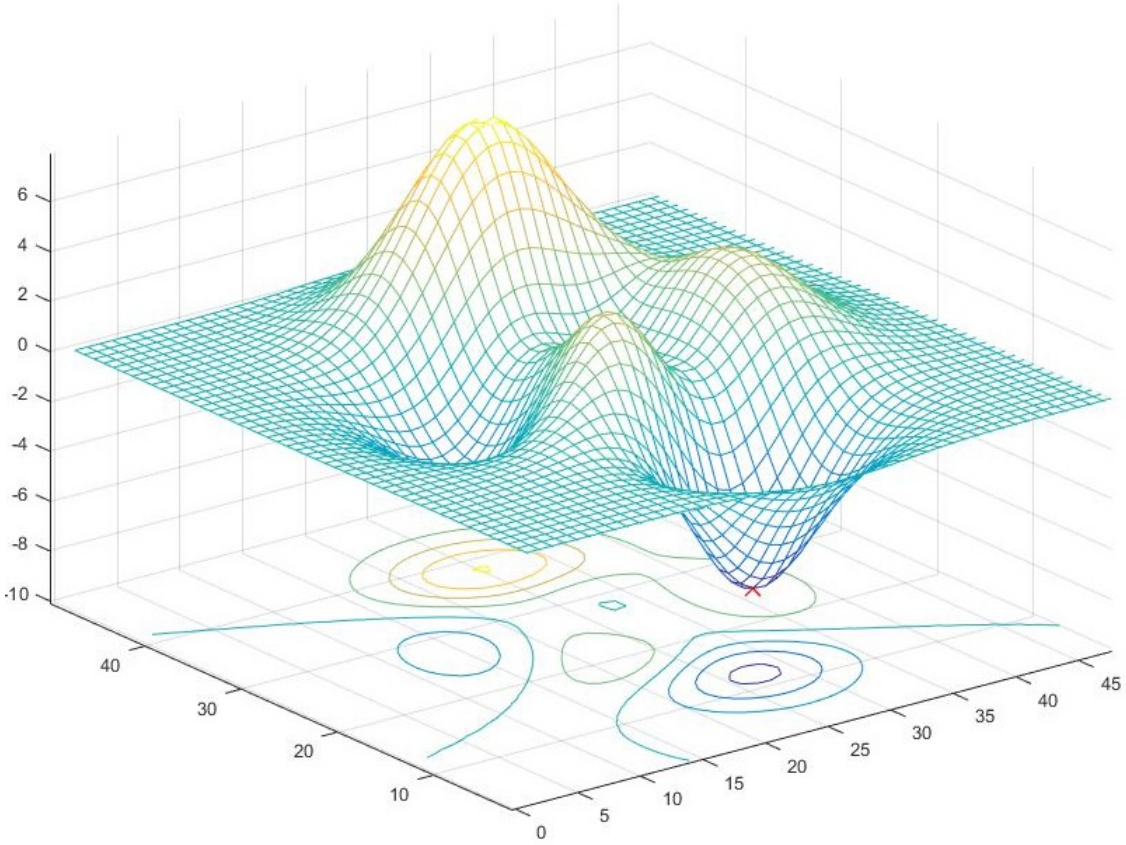

Abbildung 2.2: Funktion mit mehreren lokalen Extremwerten[1]

2.1.3 Stell- und Zielgrößen

Stellgrößen sind im Kontext der simulationsbasierten Optimierung jene Größen, welche die Optimierung verändern kann, um Einfluss auf die Lösung zu nehmen. Diese Variablen werden bei jedem Simulationslauf verändert, um eine optimale Konfiguration des Simulationsmodells zu erreichen und somit das Optimierungsziel (gemessen am Zielfunktionswert), unter Einhaltung von Nebenbedingungen, zu erreichen. Grundsätzlich werden drei Typen von Einflussgrößen unterschieden [30]:

1. Parametervariationen (z.B. die Kapazität von Maschinen).
2. Zuordnungen (z.B. von Aufträgen zu Maschinen).

[1] Quelle: Eigene Darstellung nach https://de.mathworks.com/help/matlab/ref/peaks.html (Gelesen am 27.02.2017)

3. Permutationen (z.B. Auftragsreihenfolgen)

Bei der Auswahl geeigneter Einflussgrößen muss darauf geachtet werden, dass nur jene Stellgrößen ausgewählt werden, welche die Zielgröße(n) des Systems wesentlich beeinflussen können. Zusätzliche Stellgrößen mit vernachlässigbarem Einfluss auf den Fitness-Wert von Lösungen erhöhen nur unnötig den Rechenaufwand.

Zielgrößen sind jene Größen, die das Optimierungsziel durch die Auswertung von Simulationsläufen messbar machen. Ziel jeder (simulationsbasierten) Optimierung ist das Auffinden von Extremwerten der Zielgrößen. Grundsätzlich wird zwischen einfachen Zielgrößen, welche nur ein Optimierungskriterium beinhalten, und Mehrfachzielen unterschieden. Häufige Ziele von Optimierungen in Produktion und Logistik sind die Einhaltung von Lieferterminen, die Reduktion von Beständen, die Minimierung von Durchlaufzeiten, Stillstandszeiten und Wartezeiten sowie die Auslastung von Maschinen [30].

Bei vielen Anwendungen (vor allem bei der Lösung von realen Optimierungsproblemen) werden Mehrfachziele verfolgt. Üblicherweise wird dann aus mehreren Teilzielen ein Ersatzziel ermittelt und dieses zur Bewertung von Lösungen herangezogen. Dies kann realisiert werden, indem man durch die Gewichtung mit den Faktoren λ_i und Addition von Teil-Zielfunktionswerten c_i den Fitness-Wert C errechnet [30].

$$C = \sum_{i=1}^{n} \lambda_i . c_i \tag{2.2}$$

Die Gewichtungen spiegeln die (subjektive) Wichtigkeit der Teilziele wider. Bei dieser Art der Ermittlung des Fitness-Werts müssen zwei Dinge berücksichtigt werden:

1. Alle Teilziele müssen gleich ausgerichtet sein (Minimierung oder Maximierung), andernfalls muss dies mittels Vorzeichen berücksichtigt werden.
2. Die Teilziele müssen die gleiche Größenordnung aufweisen. Ist das nicht der Fall, müssen die Teilziele (z.B. auf Maximalwerte oder Schranken) normiert werden.

Diese Ersatzzielgröße hat den Vorteil, dass man beliebig viele Teilziele in die Zielfunktion aufnehmen kann, diese aber wie eine einzelne Zielgröße behandelt werden kann. Optimierungsalgorithmen müssen also nicht zusätzlich angepasst werden. Wichtig ist jedoch, dass man die Wechselwirkungen zwischen den Zielgrößen sowie die Auswirkungen der Gewichtungen versteht.

Eine andere Vorgehensweise bei der Verfolgung von Mehrfachzielen ist, dass man erst das wichtigste Teilziel optimiert und das gefundene Optimum als eine Bedingung für die Optimierung nach anderen Kriterien definiert. Dadurch werden keine Gewichtungen benötigt, die Zielgrößen werden aber dennoch der Wichtigkeit nach gereiht [30].

2.2 Algorithmen für die simulationsbasierte Optimierung

Simulationsbasierte Optimierung ist die Optimierung einer Zielfunktion unter Nebenbedingungen, wobei die Zielfunktionswerte durch eine Simulation ermittelt werden. Diese Definition ist sehr allgemein und umfasst sowohl Optimierungsprobleme mit kontinuierlichen oder diskreten Stellgrößen als auch Probleme mit stochastischen oder deterministischen Ausgangsgrößen (siehe Modellarten in Kapitel 2.1.1). Außerdem wird zwischen lokaler und globaler Optimierung unterschieden. Für die unterschiedlichen Problemarten existiert daher eine Vielzahl an Algorithmen [2]. Diese lassen sich nach [16] in fünf Hauptkategorien unterteilen, welche im Folgenden erklärt werden.

2.2.1 Stochastische Approximation (gradientenbasierte Methoden)

Diese Algorithmen versuchen die Gradienten-Suchmethode der deterministischen Optimierung zu imitieren. Im $\mathbb{R}^n$ mit euklidischem Skalarprodukt ist der Gradient einer Funktion definiert als:

$$grad(f) = \nabla f = \frac{\partial f}{\partial x_1}\hat{\mathrm{e}}_1 + \cdots + \frac{\partial f}{\partial x_n}\hat{\mathrm{e}}_n \tag{2.3}$$

Der Gradient an einer Stelle $\vec{x}$ zeigt in die Richtung des stärksten Anstiegs. In Abb. 2.3 ist das Gradientenfeld einer Funktion in zwei Variablen in der Nähe von zwei lokalen Minima dargestellt. Die Richtung der Pfeile entspricht der Richtung des Gradienten, die Pfeillänge ist proportional zur Stärke des Anstiegs.

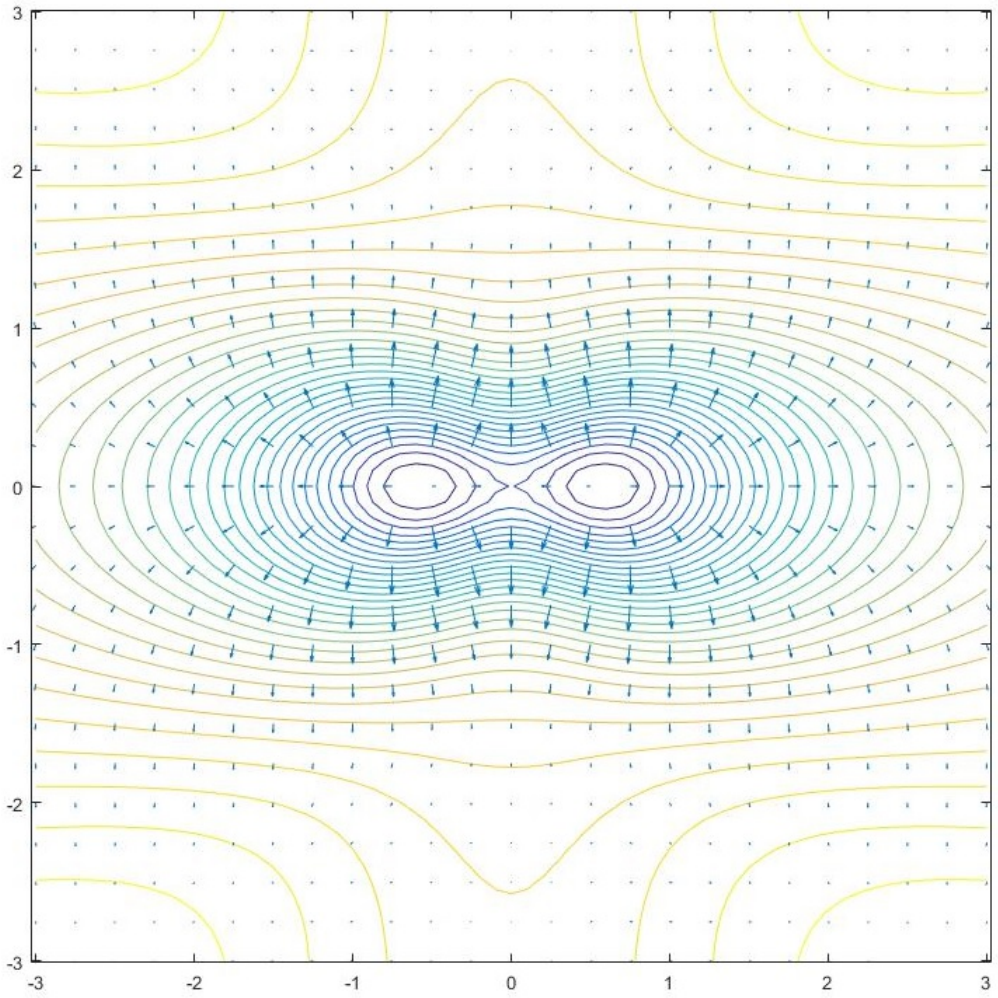

Abbildung 2.3: Gradientenfeld einer Funktion in zwei Variablen in der Nähe zweier (lokaler) Minima[2]

Ausgehend von einer Startlösung werden beim deterministischen Gradientenverfahren Suchschritte entgegen der Richtung des Gradienten, also in Richtung des steilsten Abstiegs, durchgeführt. Die Voraussetzung dafür ist allerdings die Differenzierbarkeit der Zielfunktion. Bei simulationsbasierter Optimierung wird diese Voraussetzung insbesondere bei diskreten Simulationen nicht erfüllt. Der Gradient muss dann approximiert werden, was sehr viel Rechenleistung erfordern kann. Ein Beispiel für ein stochastisches Approximationsverfahren ist Simultaneous Perturbation Stochastic Approximation (SPSA). Bei SPSA wird der Gradient, unabhängig von der Dimension des Lösungsraums, durch jeweils zwei Zielfunktionsbewertungen geschätzt, was ein Vorteil gegenüber anderen Verfahren ist, bei denen meist mehrere Bewertungen nötig sind [51]. Stochastische Approximation ist vor allem für Probleme mit kontinuierlichen Variablen

[2]Quelle: Eigene Darstellung nach https://de.mathworks.com/help/matlab/ref/gradient.html (Gelesen am 23.2.2017)

geeignet, es existieren aber auch Ansätze für diskrete Lösungsvektoren [21].

2.2.2 (Sequentielle) Response Surface Methode

Basiert auf der Erstellung von Metamodellen des Simulationsmodells auf lokaler Ebene, um den Gradienten für die Suchrichtung zu bestimmen. Eine Response Surface ist eine numerische Repräsentation der Zielfunktion des Simulationsmodells. Sie wird erstellt, indem die Simulation für eine definierte Liste von Eingangsparametern ausgeführt wird und die Zielfunktionswerte aufgezeichnet werden. Ein Metamodell ist ein algebraisches Modell der Simulation, das durch die Approximierung der Response Surface erstellt wird. Der Optimierer verwendet dann das Metamodell anstatt des Simulationsmodells zur Bewertung von Lösungen. Für die Aproximierung der Response Surface wird meist eine gewöhnliche lineare Regression verwendet [5]. Dieser Prozess läuft wiederholt ab. Es wird also nicht versucht die Zielfunktion im gesamten Lösungsraum zu approximieren, sondern nur im aktuellen Suchbereich.

2.2.3 Random Search (RS) und Direct Search (DS)

RS-Methoden wählen für Suchschritte zufällig einen Punkt aus der Nachbarschaft der aktuellen Lösung aus. Dies setzt eine Definition der Nachbarschaft voraus. Wird dadurch eine bessere Lösung gefunden, wird diese als aktuelle übernommen. RS-Methoden werden hauptsächlich bei diskreten Problemen angewendet. Für diese Methoden wurde Konvergenz, d.h. das Erreichen des tatsächlichen Optimums, nachgewiesen. Dieser Vorteil bedeutet in der Praxis aber nicht viel, da es dort eher darum geht eine gute Lösung in möglichst kurzer Zeit zu finden, als das garantierte Optimum nach einer unendlichen Anzahl an Suchschritten. Random Search Algorithmen sind verwandt mit Direct Search Algorithmen, deren Hauptmerkmal ebenfalls ein Suchvorgang mit wiederholtem, direkten Vergleich von Lösungen ist [26]. Es wird also wie bei Random Search Algorithmen kein Gradient der Zielfunktion benötigt, die Auswahl des nächsten Lösungskandidaten erfolgt aber ohne Zufallseinfluss. Ein Algorithmus dieser Kategorie ist Pattern Search (PS). Dessen grundlegende Funktionsweise ist folgende [26]:

1. Ausgehend von einer Startlösung wird ein Netz definierter Größe auf den Achsen des Lösungsraums aufgespannt. Dies geschieht durch das koordinatenweise

Addieren und Subtrahieren der Netzgröße. So erhält man $2n$ neue Lösungskandidaten, wobei n die Anzahl der Variablen des Lösungsvektors ist.

2. Die Zielfunktionswerte der Lösungskandidaten werden ermittelt (Polling). Sobald ein besserer Wert als der aktuelle gefunden wird, ersetzt dessen Position die aktuelle Position.

3. Update des Netzes. Wurde in Schritt 2 eine bessere Lösung gefunden (erfolgreicher Poll), wird die Netzgröße um einen definierten Faktor vergrößert. Bei einem nicht erfolgreichen Poll wird das Netz verkleinert. Die Suche stoppt, wenn ein Abbruchkriterium (z.B. eine zu kleine Änderung des Zielfunktionswerts) erfüllt wird.

Randomized Pattern Search (RPS) ist eine erweiterte Version des ursprünglichen PS und gehört zu den Random Search Algorithmen [34]. Anstatt den Suchraum entlang den Achsen mit variierenden Schrittweiten abzusuchen, wird ausgehend vom aktuellen Punkt einer Achse ein Zufallsvektor generiert und die resultierende Lösung mit der aktuellen verglichen. Ist die Lösung besser, wird ihre Position als aktuelle Position übernommen. Ansonsten ist der Ablauf gleich wie beim PS. RPS gleicht einige Nachteile des PS, insbesondere bei Optimierung unter Nebenbedingungen (Steckenbleiben an Schranken), aus.

2.2.4 Sample Path Optimierung

Die Idee hinter dieser Methode stellt die Optimierung einer deterministischen Funktion dar, welche auf n Simulationsläufen basiert. Die Simulationsläufe werden dabei mit verschiedenen Werten der Eingangsvariablen durchgeführt, und die Ergebnisse gemittelt, um die Eigenschaften des Simulationsmodells anzunähern [1]. Auf die ermittelte Funktion werden anschließend Methoden der deterministischen, kontinuierlichen Optimierung angewandt.

2.2.5 Metaheuristiken

Obwohl die oben genannten Ansätze einen großen Teil der Literatur zur simulationsbasierten Optimierung ausmachen, kommen in kommerzieller Optimierungs-Software hauptsächlich Metaheuristiken zur Anwendung [5]. Das liegt laut Andradottir [4] in vielen Fällen an der hohen benötigten Rechenleistung. Die Erstellung von hinreichend

genauen Response Surfaces oder Metamodellen wird bei zunehmender Komplexität des Optimierungsproblems sehr schwierig oder sogar unmöglich. Die in Produktion und Logistik mittels SBO behandelten Probleme sind meist NP-schwer und daher nur mittels Heuristiken näherungsweise lösbar [30]. Die Zielfunktion weist dann oft multiple lokale Minima auf, die durch bestimmte Strategien überwunden werden müssen, um zu einer globalen (näherungsweise) optimalen Lösung zu kommen. Viele Algorithmen der oben genannten Klassen sind nicht oder nur bedingt für die Optimierung solcher Zielfunktionen geeignet. Die Fähigkeit zur Überwindung lokaler Optima und die Fähigkeit, selbst bei sehr schweren Problemen in akzeptabler Rechenzeit gute Lösungen zu finden, zeichnen Metaheuristiken aus [18].

Eine Metaheuristik ist ein übergeordneter Prozess, der die Arbeitsweise untergeordneter Heuristiken dirigiert, um effizient gute Lösungen zu finden [54]. In komplexen Lösungsräumen müssen sie dafür sorgen, dass lokale Optima überwunden werden, um zu einer global optimalen Lösung zu kommen. Metaheuristiken betrachten das Simulationsmodell als Black-Box zur Auswertung von Eingangsparametern (siehe Abb. 2.4). Dabei wird vom Optimierer eine Kombination von Eingangsparametern ausgewählt. Anschließend wird die vom Simulationsmodell erhaltene Antwort in Form einer Bewertung des Simulationslaufs mittels Zielfunktion verwendet, um das nächste Set von Eingangsparametern zu bestimmen. Dies geschieht ohne zusätzliche Informationen wie beispielsweise dem Gradienten der Zielfunktion. Die Hauptmerkmale von Metaheuristiken sind daher die meist nicht nachweisbare Konvergenz gegen lokale oder globale Optima und die universelle Anwendbarkeit auf komplexe Optimierungsprobleme [7].

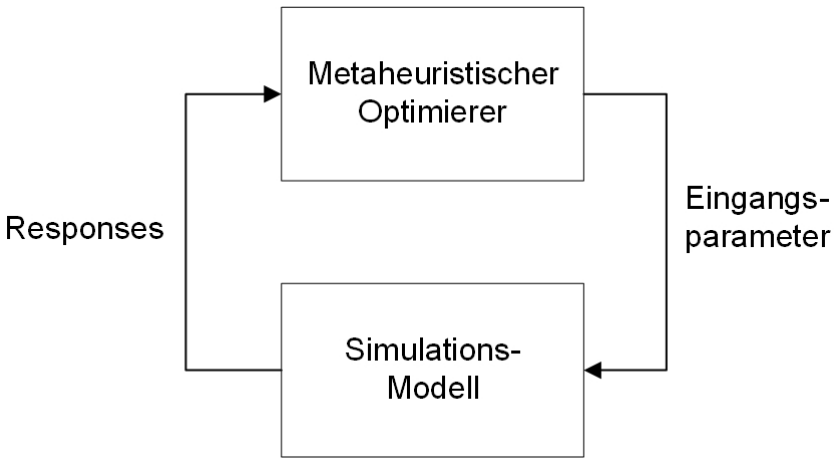

Abbildung 2.4: Black-Box Ansatz von Metaheuristiken [5]

Tabelle 2.1 enthält eine Auswahl kommerzieller Optimierungspakete sowie die unter-

stützte Simulationssoftware und die von den Optimierern verwendeten Suchprozeduren. Manche Simulationssoftwares haben bereits inkludierte Optimierer. So verfügt z.B. Plant Simulation über einen Experiment Manager, der automatisch Szenarios unter der Verwendung von genetischen Algorithmen generiert.

Optimization package	**Vendor**	**Simulation software supportet**	**Optimization methodology**
AutoStat	Applied Materials, Inc.	AutoMod	Evolutionary strategy
Evolutionary Optimizer	Imagine That, Inc.	ExtendSim	Evolutionary strategy
OptQuest	OptTek Systems, Inc.	FlexSim, @Risk, Simul8, Simio, SIMPROCESS, AnyLogic, Arena, Crystal Ball, Enterprise Dynamics, ModelRisk	Scatter search, tabu search,neural networks, integer programming
SimRunner	ProModel Corp.	ProModel, MedModel, ServiceModel	Genetic algorithms and evolutionary strategies
RISKOptimizer	Palisade Corp.	@Risk	Genetic algorithm
WITNESS Optimizer	Lanner Group, Inc.	WITNESS	Simulated annealing, tabu search, hill climbing
Plant Simulation Optimizer	Siemens AG	Siemens PLM software	Genetic algorithm
ChaStrobeGA	N/A	Stroboscope	Genetic algorithm
Global Optimization toolbox	The Math Works	SimEvents (Matlab)	Genetic algorithms, simulated annealing, pattern search, GlobalSearch, MultiStart, particle swarm optimization

Tabelle 2.1: Kommerzielle Optimierungssofware [2]

MATLAB enthält eine Global Optimization Toolbox, welche Methoden zur Lösung von globalen Optimierungsproblemen (darunter auch die Metaheuristiken Simulated Annealing, GA und PSO) anbietet[3].

[3]Vgl. http://de.mathworks.com/products/global-optimization (Gelesen am: 22.08.2016)

Die meisten Optimierungspakete verwenden Evolutionäre Strategien oder beschränken sich auf eine kleine Anzahl verschiedener Metaheuristiken [5].

Eine Klassifizierung von Metaheuristiken sowie die Beschreibung wichtiger Algorithmen der verschiedenen Klassen befindet sich in Kapitel 2.3.

2.3 Metaheuristiken (Klassifizierung und Algorithmen)

Die Klassifizierung von Metaheuristiken kann anhand verschiedener Eigenschaften der Algorithmen wie z.B. von der Natur inspiriert vs. nicht von der Natur inspiriert oder nach Methoden mit Speicher vs. Methoden ohne Speicher erfolgen [7]. In dieser Arbeit werden Metaheuristiken unterschieden in populationsbasierte Algortihmen und Algorithmen, die auf lokaler Suche basieren, da dadurch für das Anwendungsbeispiel wichtige Eigenschaften hervorgehoben werden.

2.3.1 Metheuristiken basierend auf lokaler Suche (LS)

LS-Metaheuristiken arbeiten mit einer einzelnen Lösung im Suchraum und beschreiben dort eine Trajektorie während des Suchprozesses. Daher werden sie auch als Trajektorienmethoden bezeichnet [7].

Im folgenden Abschnitt wird erst ein grundlegender LS-Algorithmus und danach komplexere Verfahren vorgestellt.

Iterative Improvement (Basic Local Search) vollführt nur eine Bewegung zur nächsten Lösung, wenn diese besser ist als die aktuelle. Dies geschieht entweder sobald eine erste Verbesserung gefunden wurde oder wenn die beste mögliche Lösung für den aktuellen Suchschritt gefunden wurde. Die konkrete Umsetzung des Algorithmus ist problemspezifisch aber alle iterativen Improvement Algorithmen stoppen, sobald ein lokales Minimum erreicht ist. Dies liefert bei Optimierungsproblemen mit multiplen lokalen Minima nicht zufriedenstellende Ergebnisse, weshalb Strategien für die Überwindung lokaler Minima benötigt werden [7].

Iterated Local Search (ILS) und Random Restart Local Search (Multistart) ILS wendet ein lokales Suchverfahren auf eine Startlösung an, bis ein lokales Optimum

gefunden wird. Anschließend wird die Lösung verändert (perturbiert) und der Vorgang wiederholt, bis eine (näherungsweise) optimale Lösung gefunden wird. Die Veränderung der Lösung darf nicht zu klein sein, da ansonsten das lokale Optimum nicht überwunden wird. Eine zu große Veränderung würde bewirken, dass sich das Verfahren wie Multistart verhält. Multistart erzeugt zufällige Startlösungen, von denen aus lokale Suchverfahren angewandt werden [7]. Als LS-Algorithmen können unter anderem lokale Metaheuristiken, Pattern Search oder das oben beschriebene Basic Local Search verwendet werden.

Simulated Annealing (SA) ist eine Metaheuristik, die auf dem Metropolisalgorithmus basiert (siehe [41]). Das Hauptmerkmal von SA ist ein Mechanismus, der dem Algorithmus durch Schritte, die den Zielfunktionswert verschlechtern erlaubt, lokalen Optima zu entkommen. SA (dt. simulierte Abkühlung) verdankt seinen Namen seiner Analogie zur kontrollierten Abkühlung von kristallinen Feststoffen. Durch die langsame Abkühlung von erhitzten, kristallinen Stoffen wird versucht ein möglichst regelmäßiges Kristallgitter zu erreichen (z.B. beim Anlassen von gehärtetem Stahl).

Beim SA werden die Zielfunktionswerte zweier Lösungen verglichen (die aktuelle und eine neu gewählte Lösung). Bessere Lösungen werden immer akzeptiert, während schlechtere Lösungen nur teilweise akzeptiert werden, mit dem Ziel lokalen Optima zu entkommen. Die Wahrscheinlichkeit, dass eine schlechtere Lösung akzeptiert wird, wird von einem Abkühlungsparameter bestimmt, welcher in der Regel monoton abnimmt und einer Boltzmann-Verteilung entspricht. Das führt dazu, dass SA bei unendlich langer Abkühlungszeit gegen ein globales Optimum konvergiert. Das ist bei der Anwendung auf praktische Beispiele zwar kaum von Bedeutung, ist aber dennoch ein Vorteil gegenüber anderen Metaheuristiken, deren Konvergenz meist nicht beweisbar ist [42]. SA wird hauptsächlich für diskrete Optimierungsprobleme angewandt

Simulated Annealing ist in Algorithmus 1 als Pseudocode dargestellt.

Algorithmus 1 Simulated Annealing [42]

1: Select an initial Solution $\omega \in \Omega$
2: Select the temperature change counter $k = 0$
3: Select a temperature cooling schedule, t_k
4: Select an initial temperature $T = t_0 \geq 0$
5: Select a repetition schedule, M_k, that defines the number of iterations executed at each temperature, t_k
6: **repeat**
7: Set repetition counter $m = 0$
8: **repeat**
9: Generate a solution $\omega` \in N(\omega)$
10: Calculate $\Delta_{\omega,\omega`} = f(\omega`) - f(\omega)$
11: **if** $\Delta_{\omega,\omega`} \leq 0$ **then**
12: $\omega \leftarrow \omega`$
13: **else**
14: $\omega \leftarrow \omega`$ with probability $exp(-\Delta_{\omega,\omega`}/t_k)$ (corresponding a Boltzmann
15: distribution)
16: **end if**
17: $m \leftarrow m + 1$
18: **until** $m = M_k$
19: $k \leftarrow k + 1$
20: **until** stopping criterion is met

Tabu Search ist wie SA eine Erweiterung klassischer LS Methoden. Im Prinzip kann TS als eine Kombination von LS mit einem Kurzzeitspeicher betrachtet werden. Zur weiteren Erklärung müssen erst die Begriffe Suchraum und Nachbarschaftsstruktur definiert werden:

- Der Suchraum einer TS-Heuristik ist der Raum aller möglichen Lösungen, die während des Suchprozesses bewertet werden.
- Die Nachbarschaftsstruktur ist eine Teilmenge des Lösungsraums, die aus (machbaren) Lösungen besteht, welche durch eine einzige Transformation der momentanen Lösung möglich sind. Anders ausgedrückt sind es jene Lösungen, welche beim nächsten Suchschritt besucht werden können.

Die Definition des Suchraums und der Nachbarschaftsstruktur ist der kritischste Teil beim Entwurf einer TS-Heuristik und erfordert eine genaue Kenntnis des zugrundeliegen-

den Optimierungsproblems da hier die Effektivität und Effizienz des Lösungsverfahres maßgeblich beeinflusst werden.

Der vorher erwähnte Kurzzeitspeicher (die Tabu Liste) ist das eigentliche Kernelement dieser Methode. Tabus werden verwendet, um Schleifen beim Entkommen lokaler Optima durch verschlechternde Suchschritte zu vermeiden. Dabei muss verhindert werden, dass die Suche die Schritte zu ihren letzten Positionen wiederholt. Dies kann z.B. durch die Definition von Tabu-Schritten, welche die letzten Suchschritte rückgängig machen würden, erreicht werden. Diese Schritte werden meist für eine bestimmte Anzahl an Iterationen gespeichert und anschließend wieder gelöscht.

Manchmal sind Tabus zu restriktiv und verhindern attraktive Suchschritte, auch wenn keine Gefahr von Zyklen besteht. Dies kann zu einer Stagnation des Suchprozesses führen. Daher ist es notwendig sogenannte Aspirationskriterien zu verwenden. Wenn diese erfüllt sind, wird der Suche erlaubt Tabu-Schritte auszuführen. Ein einfaches aber häufig angewandtes Aspirationskriterium ist einen Tabu-Schritt zuzulassen, wenn er einen besseren Zielfunktionswert als den bisher besten bekannten Zielfunktionswert liefert [20].

Der Pseudocode eines einfachen Tabu Search Algorithmus ist mit folgender Notation in Algorithmus 2 angegeben:

- S : Die aktuelle Lösung
- $S^{\text{`}}$: Die beste bekannte Lösung
- f^* : Der Wert von S^*
- $N(S)$: Die Nachbarschaft von S
- $\tilde{N}(S)$: Die zulässige Teilmenge von $N(S)$ (nicht auf Tabu Liste oder erlaubt durch Aspiration)
- T : Die Tabu Liste

Algorithmus 2 Tabu Search [18]

1: Construct an initial solution S_0
2: $S \leftarrow S_0$
3: $f^* \leftarrow f(S_0)$
4: $S^* \leftarrow S_0$
5: $T \leftarrow \emptyset$
6: **while** termination criterion is not satisfied **do**
7: select S in $argmin_{S' \in \tilde{N}(S)}[f(S')]$
8: **if** $f(S) < f^*$ **then**
9: $f^* \leftarrow f(S)$
10: $S^* \leftarrow S$
11: record tabu for the current move in T (delete oldest entry if necessary)
12: **end if**
13: **end while**

2.3.2 Populationsbasierte Metaheuristiken

Diese Metaheuristiken gehen von einer größeren Menge an Lösungen aus (einer Population). Bei den einzelnen Iterationen werden durch verschiedene Operationen neue Lösungen erzeugt, welche anschließend die nächste Population bilden. Aus der gleichzeitigen Betrachtung mehrerer Lösungen folgt meist eine bessere Erkundung des Lösungsraums, als bei Metaheuristiken, die auf lokaler Suche basieren [7]. Im folgenden Abschnitt werden einige, in der Literatur häufig behandelte, populationsbasierte Metaheuristiken vorgestellt.

Evolutionäre Algorithmen (EAs) sind von der Anpassung und Evolution der Arten, basierend auf Darwin's Prinzip der natürlichen Selektion, inspiriert. EAs wenden bei jeder Iteration eine Reihe von Operatoren auf die jeweils aktuelle Population, bestehend aus Lösungskandidaten, an. Üblicherweise sind das Rekombination, um durch Kombination zweier oder mehrerer Individuen neue Lösungen zu erzeugen und Mutation, die eine Selbstanpassung der Individuen bewirkt. Eine Selektionsmethode bestimmt, basierend auf der Fitness der erzeugten Lösungen, welche Individuen in die nächste Population aufgenommen werden [56].

Genetische Algorithmen (GAs) [19], [11], [47] [56] sind die bekannteste Form evolutionärer Algorithmen. Dabei wird eine Population von Startlösungen durch die Anwendung von Selektion, Rekombination und Mutation von einer Generation zur nächsten entwickelt, bis ein Abbruchkriterium erfüllt ist. Die probabilistische Selektion der Elternindividuen sowie die Rekombination sind dabei die primären Operatoren. Eine Mutation wird nur mit geringer Wahrscheinlichkeit durchgeführt und dient dazu alle Lösungen im Lösungsraum erreichbar zu machen. Die Umsetzung dieser drei Operatoren kann in verschiedenen Varianten erfolgen, einige davon werden in Kapitel 3 genauer beschrieben. Lösungsvektoren können binäre Zeichenketten, Permutationen oder reelwertige Vektoren sein. Im Folgenden wird die grundlegende Funktionsweise der Operatoren erklärt.

1. Eine Selektion erreicht man, indem nur den besten Lösungen einer Population (messbar durch den Fitnesswert) erlaubt wird Abkömmlinge zu erzeugen. Dies wird realisiert, indem den Lösungen Wahrscheinlichkeiten entsprechend ihrer Zielfunktionswerte zugeordnet werden und die Selektion entsprechend dieser Wahrscheinlichkeiten durchgeführt wird. Verschiedene Varianten der Selektion werden im praktischen Teil dieser Arbeit näher beschrieben und getestet.

2. Bei der Rekombination werden zwei oder mehrere, vom Selektions-Operator ausgewählte, Lösungen kombiniert, indem Teile von ihnen ausgetauscht werden. Dies kann auf mehrere Arten geschehen. Bei der n-point Rekombination werden beispielsweise zwei Elternindividuen an n (zufällig ausgewählten) Punkten in Abschnitte unterteilt. Neue Lösungen werden anschließend durch die Kombination von Abschnitten beider Eltern erstellt. In (2.4) ist ein 2-point-crossover von zwei 5-dimensionalen Lösungsvektoren dargestellt.

$$\begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{pmatrix} und \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{pmatrix} \xrightarrow{crossover} \begin{pmatrix} a_1 \\ a_2 \\ b_3 \\ b_4 \\ a_5 \end{pmatrix} und \begin{pmatrix} b_1 \\ b_2 \\ a_3 \\ a_4 \\ b_5 \end{pmatrix} \tag{2.4}$$

3. Eine Mutation wird meist nur an einem kleinen Teil der Elternindividuen durchgeführt. Dabei werden die Lösungen meist nur geringfügig verändert. Mutation erhöht den Einfluss des Zufalls bei der Suche und es können Lösungen erzeugt

werden, die durch die Rekombination alleine nicht realisierbar sind. Dies verhindert einen Abbruch des Optimierungsprozesses beim Erreichen lokaler Optima.

Algorithmus 4 zeigt den Pseudocode eines (sehr allgemein gehaltenen) Genetischen Algorithmus.

Algorithmus 3 Genetischer Algorithmus [18]

```
1: Choose an initial population of chromosomes.
2: while termination condition not satisfied do
3:     repeat
4:         if crossover condition satisfied then
5:             select parent chromosomes.
6:             choose crossover parameters.
7:             perform crossover
8:         end if
9:         if mutation condition satisfied then
10:            choose mutation points.
11:            perform mutation.
12:        end if
13:        evaluate fitness of offspring
14:    until sufficient offspring created.
15:    select new population.
16: end while
```

Evolutionsstrategien [56] gehören wie GAs ebenfalls zu den evolutionären Algorithmen. Bei Evolutionsstrategien werden aus μ Elternindividuen λ Kinder erzeugt. Ein typisches Verhältnis von μ/λ ist $1/5$. Anschließend werden aus den erzeugten Lösungskandidaten die besten μ Lösungen für die nächste Generation ausgewählt. Erfolgt die Selektion nur aus den erzeugten Kindern, spricht man von einer Komma-Selektion. Andernfalls von einer Plus-Selektion. Die Auswahl der Elternindividuen für die Erzeugung neuer Lösungen erfolgt ohne Selektionsdruck also zufällig und gleichverteilt.

Im Gegensatz zu GAs ist bei Evolutionsstrategien die Mutation der primäre Operator. Oftmals wird ganz auf eine Rekombination verzichtet, da durch die reelwertige Repräsentation der Lösungen nicht immer sichergestellt ist, dass gültige Lösungskandidaten erzeugt werden. Der Mutations-Operator muss daher zusätzlich zur Feinabstimmung eine ausreichende Erforschung des Lösungsraums sicherstellen. Dafür wird meist eine

normalverteilte Änderung von Lösungsvektoreinträgen verwendet. Dadurch werden hauptsächlich kleine Änderungen vorgenommen, größere Änderungen sind mit einer kleinen Wahrscheinlichkeit aber ebenfalls möglich.

Scatter Search ist eine evolutionäre Metaheuristik, die den Lösungsraum durch die Entwicklung von Referenzpunkten (das Referenzset) durchsucht. Dabei macht Scatter Search im Vergleich zu anderen evolutionären Strategien nur begrenzten Gebrauch von Randomisierung. Scatter Search erzeugt neue Lösungen, indem konvexe und nicht konvexe Kombinationen einer variablen Anzahl von Lösungen im Referenzset erzeugt werden. Die so erzeugten Lösungskandidaten werden anschließend heuristisch verbessert oder repariert (d.h. in den machbaren Lösungsraum projiziert) [48], [19]. Der Scatter Search Ansatz besteht im Grunde aus fünf Methoden, welche auf verschiedene Arten umgesetzt werden können. Diese sind [40]:

1. Eine *Diversification Generation Methode*, welche für die Generierung von möglichen Lösungen durch die Verwendung einer per Zufall generierten Lösung zuständig ist.
2. Eine *Improvement Methode*, welche die Lösungskandidaten in eine oder mehrere verbesserte Lösungskandidaten transformiert. Die so erzeugten Lösungskandidaten sind normalerweise zulässig.
3. Eine *Referenzset Update Methode* für die Erzeugung und das Update des Referenzsets, welches aus den b besten Lösungen besteht (b ist dabei typischerweise klein). Lösungskandidaten werden in das Referenzset aufgenommen, wenn sie einen guten Zielfunktionswert oder sehr verschieden von den anderen Lösungen im Referenzset sind.
4. Eine *Subset Generation Methode*, welche mit dem Referenzset operiert und Teilmengen der dort enthaltenen Lösungen für die anschließende Kombination bildet.
5. Eine *Solution Combination Methode* für die Transformation der erstellten Teilmengen in eine oder mehrere kombinierte Lösungsvektoren.

Die Interaktion dieser fünf Methoden wird in Algorithmus 3 demonstriert.

Algorithmus 4 Scatter Search [18]

1: Start with $P = \emptyset$. Use the diversification generation method to construct a solution and apply the improvement method. Let x be the resulting solution. If $x \notin P$ then add x to P, otherwise discard x. Repeat this step until $|P| = PSize$.
2: Use the reference set update method to build $RefSet = \{x^1, ..., x^b\}$ with the "best"b solutions in P. Order the solutions in $RefSet$ according to their objective function value such that x^1 is the best solution and x^b the worst. Make $NewSolutions = TRUE$
3: **while** $NewSolutions$ **do**
4: Generate $NewSubsets$ with the subset generation method.
5: Make $NewSolutions = False$
6: **while** $NewSubsets \neq \emptyset$ **do**
7: Select the next subset in $NewSubsets$.
8: Apply the solution combination method to s to obtain one or more new trial
9: solutions x. Apply the improvement method to the trial solutions.
10: Apply the reference set update method.
11: **if** $RefSet$ has changed **then**
12: Make $NewSolutions = TRUE$
13: **end if**
14: Delete s from $NewSubsets$.
15: **end while**
16: **end while**

Erst wird eine Menge mit P Startlösungen mithilfe der *Diversification Generation Methode* erzeugt. Daraus werden die b besten Lösungen ausgewählt und ins Referenzset übernommen. Die Auswahl erfolgt durch die *Referenzset Update Methode*. Typischerweise gilt dabei $P \geq 10 * b$. Die Lösungen im Referenzset werden entsprechend der Lösungsqualität in absteigender Reihenfolge geordnet. Anschließend beginnt die Suche mit der Zuordnung von $False$ zum Boolean $NewSolutions$. Danach wird $NewSubsets$ von der *Subset Generation Methode* z.B. durch die Bildung aller Paare von Lösungen im Referenzset erzeugt. Die Lösungspaare werden von der *Solution Combination Methode* verwendet, um durch Kombination neue Lösungskandidaten zu erzeugen. Diese Lösungskandidaten werden von der *Improvement Methode* verbessert und danach von der *Referenzset Update Methode* überprüft und gegebenenfalls ins Referenzset aufgenommen. Falls es dadurch verändert wurde, wird $NewSolutions$ der Wert $True$ zugewiesen, wodurch die erste while-Schleife wieder durchlaufen wird. Das Subset s,

welches gerade von der *Subset Generation Methode* bearbeitet wurde, wird danach von $NewSubsets$ gelöscht.

Particle Swarm Optimization (PSO) wurde ursprünglich als Beschreibungsmodell für das Nahrungssuchverhalten von Vogelschwärmen entwickelt. In Eberhart und Kennedy (1995) wird PSO aber erstmals als Konzept zur Optimierung nichtlinearer Funktionen beschrieben. PSO-Algorithmen erzeugen zufallsbasiert eine große Menge an Partikeln (Lösungskandidaten) im Lösungsraum. Jedem Partikel ist eine Position $\vec{x_i}$, eine Geschwindigkeit $\vec{v_i}$ und ein Speicher für die beste bisherige Position zugeordnet. Die Partikel bewegen sich im D-dimensionalen Suchraum. Der Zielfunktionswert wird für jedes Partikel am aktuellen Ort ermittelt. Die Änderung des Geschwindigkeitsvektors der Partikel wird durch die aktuelle Position, der historisch besten Position $\vec{p_i}$ und der Position der besten, im Schwarm gefundenen Lösung $\vec{p_g}$ beeinflusst. Außerdem wird die Änderung unter Zufallseinfluss durchgeführt. Nach einer Aktualisierung der Positionen beginnt der Prozess von vorne, bis ein Abbruchkriterium erfüllt ist [27], [44], [11].

In Algorithmus 5 ist der Pseudocode eines PSO-Algorithmus mit folgender Notation dargestellt:

D : Dimension des Lösungsraums
$pbest_i$: Bisher bester Zielfunktionswert von Partikel i
$\vec{p_i}$: Die Position der bisher besten Lösung von Partikel i
$\vec{x_i}$: Die aktuelle Position von Partikel i
g : Index für die bisher beste gefundene Lösung des Schwarms
$\vec{v_i}$: Geschwindigkeit von Partikel i. $\vec{v_i} \in [V_{max}, V_{min}]$
$\vec{U}(0, \phi_1)$: Ein Vektor bestehend aus in $[0, \phi_i]$ gleichverteilten Zufallszahlen
$\vec{p_g}$: Die Position des bisher besten gefundenen Lösung des Schwarms
$\otimes$: Komponentenweise Multiplikation

Algorithmus 5 Particle Swarm Optimization [44]

1: Initialize a population array of particles with random positions and velocities on D dimensions in the search space.
2: **while** termination condition not satisfied **do**
3: For each particle, evaluate the desired optimization fitness function in D
4: variables.
5: Compare particle's fitness evaluation with its $pbest_i$.
6: **if** current value is better than $pbest_i$ **then**
7: set $pbest_i$ equal to the current value.
8: set $\vec{p_i}$ equal to the current location $\vec{x_i}$ in D-dimensional space.
9: **end if**
10: Identify the particle in the neighborhood with the best success so far and assign
11: its index to the variable g.
12: Change the velocity and position of the particle according to the following
13: equations:
14:

$$\begin{cases} \vec{v_i} \leftarrow \vec{v_i} + \vec{U}(0, \phi_1) \otimes (\vec{p_i} - \vec{x_i}) + \vec{U}(0, \phi_1) \otimes (\vec{p_g} - \vec{x_i}) \\ \vec{x_i} \leftarrow \vec{x_i} + \vec{v_i} \end{cases} \tag{2.5}$$

15: **end while**

2.3.3 Hybride Metaheuristiken

In den letzten Jahren werden vermehrt sogenannte metaheuristische Hybriden zur Lösung sehr schwerer Optimierungsprobleme angewandt. Bei der Lösung von vielen realen Problemen kommen kombinierte Methoden zum Einsatz, wobei versucht wird die jeweiligen Vorteile der individuellen Methoden auszunutzen und gleichzeitig die Nachteile auszugleichen. Diese Hybriden liefern meist bessere Ergebnisse hinsichtlich Effektivität und Effizienz als deren Komponenten alleine.

Die Idee Metaheuristiken zu kombinieren ist nicht neu, sondern so alt wie die Heuristiken selbst. Anfangs waren solche Kombinationen jedoch nicht sehr populär, da die betroffenen Forschergruppen eher im Wettbewerb standen und versuchten ihre eigenen Ansätze durchzusetzen. Dies änderte sich mit den No-free-Lunch-Theoremen, welche besagen, dass keine Optimierungsstrategie existieren kann, die gemittelt über alle Probleme besser als alle anderen ist [59]. Die effiziente Lösung eines spezifischen Optimierungsproblems erfordert daher meist einen genau auf das Problem abgestimm-

ten Algorithmus, welcher oft aus Teilen verschiedener Metaheuristiken und spezifisch angepassten Methoden besteht. Diese Abstimmung sowie die Auswahl und Zusammensetzung der richtigen Komponenten gestaltet sich meist schwieriger als bei der Anwendung einzelner Metaheuristiken, da mehr Parametereinstellungen und Optionen getestet werden müssen. Beispiele für hybride Metaheuristiken sind Kombinationen populationsbasierter Methoden wie GAs mit LS-Methoden. Die populationsbasierten Methoden sind gut bei der Auffindung von Bereichen mit Lösungen hoher Qualität im Lösungsraum. Bei der Initialisierung erfassen sie ein globales Bild des Lösungsraums und fokussieren die Suche in weiterer Folge auf vielversprechende Gebiete. Jedoch sind sie nicht sehr effektiv bei der Auffindung der besten Lösung innerhalb dieser Gebiete, weshalb hier LS angewandt wird, dessen Stärke die schnelle Verbesserung von gegebenen Startlösungen ist. Evolutionäre Algorithmen, die lokale Suchmethoden verwenden werden auch als Memetische Algorithmen bezeichnet. [29]

Einige weitere erfolgreiche Ansätze wie z.B. die Kombination von Metaheuristiken mit ganzzahliger, linearer Optimierung finden sich in Blum u.a. [9].

Der Nachteil hybrider Metaheuristiken liegt in der höheren Komplexität im Vergleich zu klassischen Strategien. Dies führt meist zu einem wesentlich höheren Aufwand bei der Entwicklung und der Anpassung an ein spezifisches Problem, da neben den Parametereinstellungen und Optionen der einzelnen Algorithmen auch das Zusammenwirken der Methoden definiert und optimiert werden muss [46], [10].

2.4 Beschreibung der Optimierungsaufgabe

In diesem Kapitel wird das Anwendungsbeispiel, welches im Rahmen dieser Arbeit behandelt wird, näher beschrieben. Das Beispiel ist ein Simulations-Prototyp, d.h. ein vereinfachtes Simulationsmodell einer realen Produktionsanlage. Es entstammt dem Projekt BaMa (Balanced Manufacturing) der Technischen Universität Wien und soll die wesentlichen Konzepte der BaMa-Simulation zeigen.

2.4.1 Simulationsmodell

Das Simulationsmodell ist eine vereinfachte Abbildung einer Produktionslinie für Backwaren (siehe Abbildung 2.5). Die Produktionslinie kann tiefgekühlte und gebackene

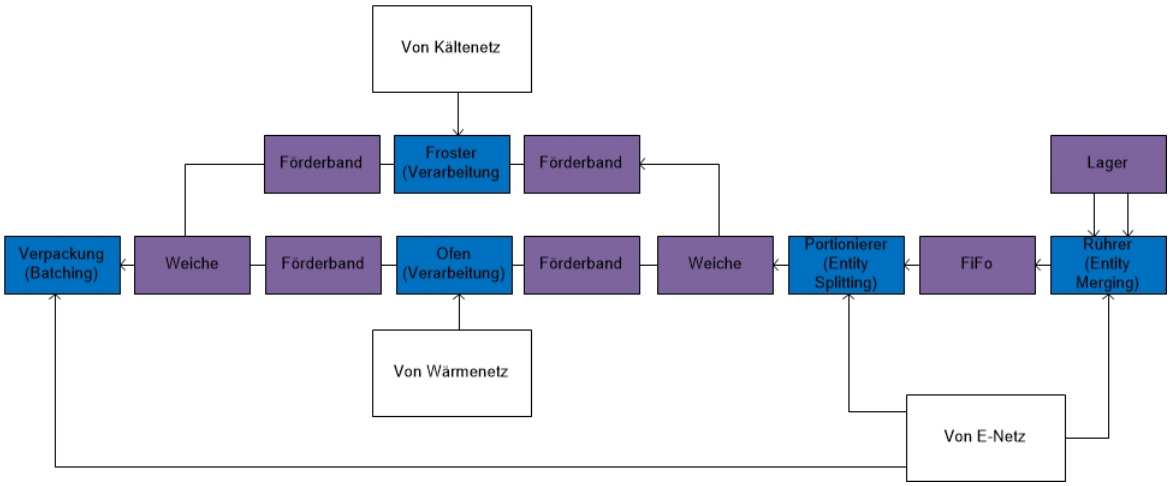

Abbildung 2.5: Produktionslinie [50]

Semmeln herstellen, wobei die Produkte jeweils verschiedene Pfade durch die Produktionslinie nehmen.

Das Modell besteht aus verschiedenen Elementen, welche unterschiedliche Eigenschaften und Funktionen aufweisen:

- Der Merging-Cube dient der Verschmelzung mehrerer Entitäten am Eingang zu einer Entität am Ausgang. In diesem Fall werden Zutaten aus dem Lager zu einem Teig verrührt. Die Umwandlung findet nicht unmittelbar statt, sondern benötigt eine vorher definierte Zeit.
- Der Splitting-Cube dient der Aufteilung einer Entität am Eingang auf mehrere Entitäten am Ausgang. Hier dient der Splitting-Cube der Aufteilung der Teigeinheiten auf mehrere Portionen (Teiglinge). Die Portionierung geschieht taktweise d.h. es wird eine Entität pro Takt ausgegeben, bis die Entität am Eingang aufgebraucht ist.
- Der Batching-Cube dient in unserem Fall der Verpackung von Semmeln in Verpackungseinheiten. Es werden also mehrere eingehende Entitäten zu einer ausgehenden Entität zusammengefasst. Im Unterschied zum Merging bleiben die eingehenden Entitäten aber als Unter-Entitäten erhalten.
- Verzweigungen dienen dazu, Entitäten auf mehrere Pfade aufzuteilen oder von mehreren Pfaden zusammenzuführen. Die erste Weiche in diesem Beispiel (vor Ofen und Froster) legt den Pfad der Produkte anhand des ankommenden Produkttyps fest. Die zweite Weiche dient der Weitergabe von Entitäten beider Pfade und benötigt keine Stellgrößen.

Die Steuerung des Simulationsmodells erfolgt nach Produkttyp und Produktionsplan. Das heißt, dass die Produkte bereits im Lager als solche in Auftrag gegeben werden. Der Produkttyp wird als Attribut der jeweiligen Entitäten geführt. Damit ist allen Cubes der Typ der Entität bekannt, sobald sie betreten werden, wodurch vom Typ abhängige Entscheidungen getroffen werden können. In diesem Beispiel erfolgt eine solche Entscheidung an der ersten Weiche. Hier wird aufgrund des Produkttyps entschieden, ob die Entität durch den Ofen oder den Froster geschickt werden soll. Der Produktionsplan dient dazu, das zeitliche Verhalten der Produktion zu definieren. Dazu werden vor dem Start der Simulation Zeitpunkte und herzustellende Produkte (Menge und Typ) geplant. Anhand des Produktionsplans können dann Cubes zeitlich auf verschiedene Betriebszustände umgestellt werden. Er steuert alle weiteren Cubes, die eine von anderen Cubes unabhängige zeitliche Steuerung haben sollen um damit eine energetische Optimierung zu ermöglichen. In diesem Beispiel werden auf diese Weise das Lager, der Merger, der Ofen sowie der Froster gesteuert. Für den Produktionsplan gelten Beschränkungen und Randbedingungen, um einen unrealistischen Produktionsplan und somit ein fehlerhaftes Modellverhalten auszuschließen.

In Abbildung 2.6 ist beispielhaft die Herstellung von jeweils einem Los von Produkttyp 0 (gebackene Semmel) und Produkttyp 1 (Tiefkühlsemmel) dargestellt. Die Lose bestehen aus 8 bzw. 16 TE.

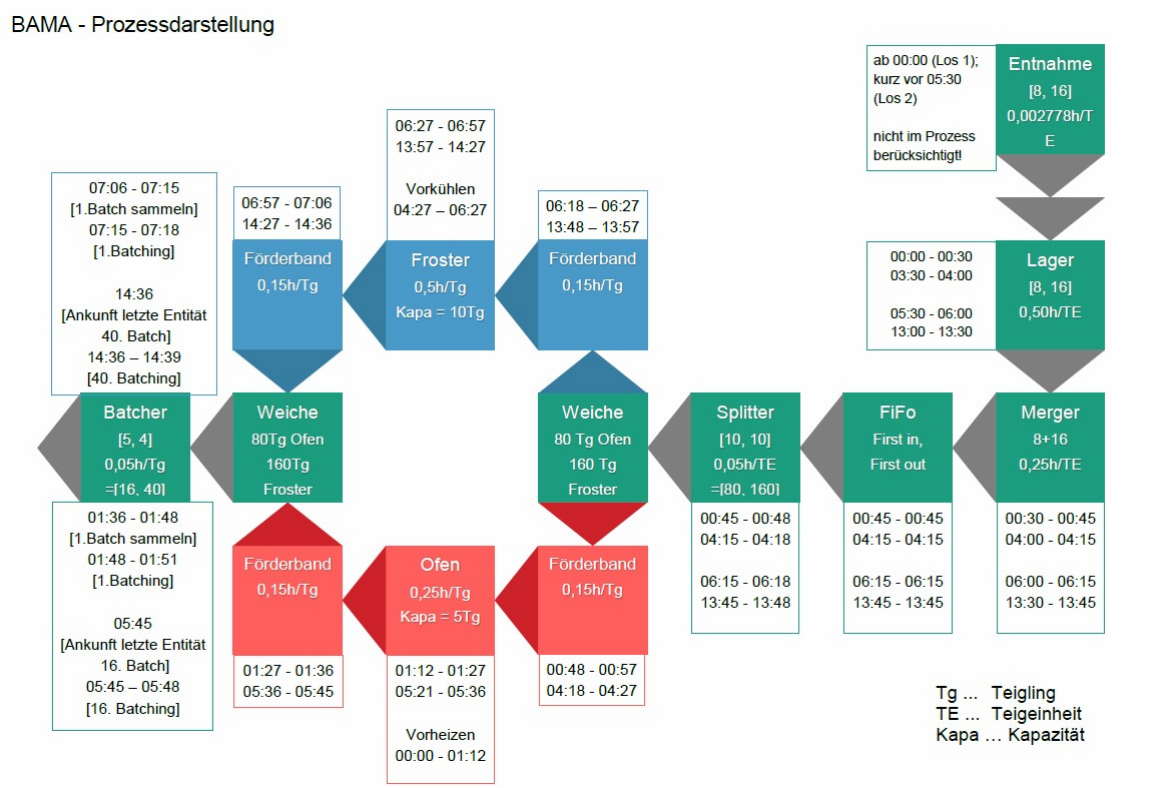

Abbildung 2.6: Prozessdarstellung (der BaMa Projektdokumentation entnommen)

Für jeden Cube sind pro Los Beginn und Ende der Aktivität für den ersten und letzten Vorgang angegeben. Der Prozess wird vom Produktionsplan im Lager um 00:00 angestoßen. Die Entnahme einer TE (bzw. deren Zutaten) aus dem Lager dauert 0,5 Stunden. Die erste TE erreicht den Merger somit um 00:30. Nach einer Bearbeitungsdauer von 0,25 Stunden wird die erste TE dem Splitter zugeführt. Jede TE wird dort in 10 Tg aufgeteilt (Bearbeitungszeit je Split-Vorgang = 3 Minuten). Anschließend wird der erste Tg um 00:48 mittels Förderband (Transportdauer = 9 Minuten) zum Ofen transportiert (Ankunft 00:57). Der Backvorgang beginnt sobald der erste Tg den Ofen erreicht, jedoch benötigt der Ofen vor dem ersten Backvorgang eine Aufheizdauer von 1,2 Stunden. Die Aufheizdauer ist zeitlich wesentlicher Bestandteil der energetischen Optimierung. Der Aufheizvorgang beginnt in diesem Beispiel mit dem Start der Simulation. Der erste Backvorgang kann somit um 01:12 beginnen. Nach 15 Minuten verlässt der erste Tg den Ofen und erreicht nach 9 Minuten den Batcher. Bevor ein Batching-Vorgang beginnen kann (Dauer = 3 Minuten), muss jedes Mal auf 5 gebackene bzw. 4 tiefgekühlte Semmeln gewartet werden. Das erste Batching erfolgt daher um 01:36. Nachdem die letzten gebackenen Semmeln um 05:48 verpackt wurden, beginnt der Prozess für Los 2 um 05:30 von vorne. Der Ablauf ist im Prinzip gleich, der Pfad führt aber gemäß Arbeitsplan durch den Froster.

2.4.2 Stellgrößen

Die Stellgrößen der Optimierung sind die Einsteuerungszeitpunkte der Lose (und somit auch deren Reihenfolge) sowie die Ein- und Ausschaltzeitpunkte von Merger, Ofen und Froster. Der Lösungsvektor x besteht grundsätzlich aus vier Teilvektoren. Für die Herstellung von z.B. zwei Losen innerhalb eines Tages wäre 2.6 eine zulässige Lösung.

$$\vec{x} = [\underbrace{0, 5.5}_{Storage}, \overbrace{0.5, 6, 24}^{Merger}, \underbrace{0, 24}_{Oven}, \overbrace{0, 24}^{Freezer}], \tag{2.6}$$

Die ersten beiden Einträge betreffen das Lager. $0, 5.5$ bedeutet, dass nach 0 Stunden das erste Los ausgelagert wird bzw. nach 5.5 Stunden das Zweite. Die nächsten drei Einträge $0.5, 6, 24$ betreffen den Merger und bedeuten, dass dieser nach 0,5 Stunden für das erste bzw. nach 6 Stunden für das zweite Los angetriggert wird sowie nach 24 Stunden abgeschaltet wird. Die nächsten Einträge $0, 24$ betreffen den Ofen und bedeuten, dass dieser von 0-24 Uhr läuft. Die letzten beiden Einträge betreffen den Froster und haben die gleiche Bedeutung wie für den Ofen.

Die Struktur der Lösungsvektoren bleibt im Prinzip immer gleich, jedoch wächst die Anzahl der Einträge (und damit die Komplexität des Optimierungsproblems) mit der Anzahl an zu fertigenden Losen.

2.4.3 Testszenarios

Für die Optimierung wird ein Testszenario benötigt, welches realen Betriebsbedingungen entsprechen soll und die in der realen Produktion zu erwartende Planungskomplexität (Stellgrößen, Anzahl an Variablen, Aggregaten und Aufträgen) bietet. Dazu wird das in Tabelle 2.2 dargestellte Szenario mit einem siebentägigen Planungszeitraum verwendet. Das Testszenario gibt die Anzahl an Produktionslosen und deren Größe für die beiden Produkttypen vor. Der zugehörige Absatzplan (siehe Tabelle 2.3) definiert die geforderten Mengen fertiger Produkte zu bestimmten Zeitpunkten. Der Absatzplan dient somit zur Bewertung von Lösungen hinsichtlich Lieferverzug und Lagerhaltungskosten. Ein Ziel der Optimierung besteht somit in der möglichst optimalen Erfüllung des Absatzplans.

Produkttyp 0 (Losgröße in TE)	Produkttyp 1 (Losgröße in TE)
6	6
4	8
7	10
4	6
6	4
4	12
5	8

Tabelle 2.2: Testszenario für $t = 7d$

Geforderte Menge fertiger Batches		Zeit [s]
Typ 0	Typ 1	
5	10	42000
12	24	85000
18	40	170000
30	60	260000
48	80	430000
60	100	520000
70	130	604800

Tabelle 2.3: Absatzplan für $t = 7d$

2.4.4 Zielfunktionssytem

Das Zielfunktionssystem (2.7) dient zur Bewertung von Lösungsvorschlägen der Optimierung. Es besteht aus vier Teilzielen, welche den Schwerpunkten des Forschungsprojekts und den Präferenzen des Produktionsplaners entsprechend gewichtet sind. Der für jede Lösung ermittelte Funktionswert (Fitnesswert) spiegelt die Qualität der Lösung wider. Ziel der Optimierung ist die Minimierung dieser Funktion.

$$f(x) = \omega_1 \sum_{i=1}^{n} (f_1(c_i)) + \omega_2 n_2 + \omega_3 n_3 + \omega_4 n_4 \tag{2.7}$$

$\omega_1 - \omega_4$: Gewichtungen

f_1 : Bewertungsfunktion für Lieferverzug und Lagerhaltungskosten

c_i : Relativer Fertigstellungszeitpunkt

n_2 : Kumulierte Gesamtlosdurchlaufzeit

n_3 : Kumulierter Gesamtenergieverbrauch

n_4 : Kumulierte Differenz zwischen Soll- und Ist-Produktionsmenge

Lieferverzug und Lagerhaltungskosten. Für die Ermittlung dieses Teil-Fitnesswerts werden die Ergebnisse der Simulation mit dem Absatzplan verglichen. Die zur Bewertung von zu früh oder zu spät fertiggestellten Batches verwendeten Faktoren sind abhängig von der simulierten Dauer und vom Produkttyp. Der Lieferverzug wird bewertet, indem die Anzahl fertiger Entitäten zu den im Absatzplan definierten Zeitpunkten mit der benötigten Anzahl verglichen wird. Die Fehlmenge wird mit dem Faktor für Liefertermintreue (siehe Tabelle 2.4) multipliziert um den Teil-Fitnesswert zu erhalten.

Der Teil-Fitnesswert für die Lagerhaltungskosten wird berechnet, indem erst die Anzahl der laut Absatzplan zu früh fertig gestellten Entitäten sowie die benötigte Lagerdauer (auf Sekundenbasis) ermittelt werden. Der Preis für die Lagerung beträgt pro Stellplatz 4,76 € pro Monat. Der Fitnesswert ergibt sich durch die Multiplikation von Lagerdauer (in Sekunden), Anzahl an zu lagernden Entitäten und den in Tabelle 2.4 dargestellten Faktoren. Da die Lagerung der tiefgekühlten Produkte teurer als die der normalen Produkte ist, wird der Wert für die Tiefkühl-Produkte mit dem Faktor 2 berücksichtigt.

	Faktor Lagerhaltung pro zusätzlichem Tag	**Faktor Liefertermintreue**
Szenarios 1T	0.5	20
Szenarios 7T	0.2	20
Szenarios 30T	0.005	20

Tabelle 2.4: Faktoren für Lieferverzug und Lagerhaltungskosten

Gesamtlosdurchlaufzeit. Die Gesamtlosdurchlaufzeit geht mit dem Gewichtungsfaktor $\omega_2 = 0,002$ (für die 1T und 7T Szenarios) bzw. mit Gewichtungsfaktor $\omega_2 = 0,005$ (für die 30T Szenarios) in die Zielfunktion ein.

Gesamtenergieverbrauch: Dieser Teil-Fitnesswert spiegelt den Gesamtenergieverbrauch sowie den CO_2-Impact wider. Für die Berechnung wird die verbrauchte Gesamtenergiemenge in kWh nach dem Simulationslauf ermittelt. In der Simulation werden drei Energie-Verbrauchsformen bewertet:

1. Elektrische Energie
2. Heizbedarf
3. Prozesskälte

Die verbrauchte Energiemenge wird mit einem zeitabhängigen Kostenfaktor gewichtet, der die Zeitabhängigkeit der Energiekosten modelliert. Die Kostenfunktion ist sinusförmig umgesetzt. Die berechneten Werte werden anschließend mit den Verbrauchsformabhängigen Faktoren (1, 0.2, 0.5) für (elektrische Energie, Heizbedarf, Prozesskälte) multipliziert und aufsummiert. Das Ergebnis bildet den ersten Anteil dieses Teil-Fitnesswerts. Der Anteil für den CO_2-Impact wird berechnet, indem die oben ermittelten Werte für die Energiekosten mit den Faktoren (1, 0.2, 3) für (Elektrische Energie, Heizbedarf, Prozesskälte) gewichtet und aufsummiert werden. Die Summe aus dem Anteil für Energiekosten und dem Anteil für den CO_2-Impact bilden den Teil-Fitnesswert für den Energieverbrauch.

$$\omega_2 n_3 = Energiekostenanteil + AnteilCO_2Impact \tag{2.8}$$

mit dem Gewichtungsfaktor $\omega_2 = 1$ für alle Szenarios.

Differenz zwischen Soll- und Ist-Produktionsmenge. Dieser Teil des Fitnesswerts wird berechnet, indem nach dem Simulationslauf die gewünschte Anzahl an Entitäten pro Produkttyp mit der tatsächlich produzierten Anzahl verglichen wird. Die Abweichung geht mit dem Gewichtungsfaktor $\omega_2 = 1000$ in die Zielfunktion ein.

Die Gewichtungen oder Bestrafungsfaktoren für die Teil-Fitnesswerte lassen die Prioritäten bei der Bewertung erkennen. Oberste Priorität ist, dass alle geforderten Produkte produziert werden müssen. Zweite Priorität ist die rechtzeitige Fertigstellung der gemäß Absatzplan geforderten Entitäten bzw. die Vermeidung von zu spät fertiggestellten Produkten. Dritte Priorität ist die Minimierung der Energiekosten. Lagerhaltungskosten und Losdurchlaufzeiten tragen nur einen kleinen Teil zum Fitnesswert der Lösung bei.

2.5 Klassifizierung des Optimierungsproblems

Allgemein kann man das behandelte Problem als Problem der simulationsbasierten Optimierung mit Mehrfachzielen unter Nebenbedingungen einordnen. Es folgen eine genauere Klassifizierung des Simulationsmodells und eine nähere Beschreibung der vorliegenden Problemart.

2.5.1 Simulationsmodell

Wie bereits in Kapitel 2.1 beschrieben, kann die Klassifizierung eines Simulationsmodells in drei verschiedenen Dimensionen erfolgen [33]:

- Statische vs. Dynamische Modelle. Das in dieser Arbeit betrachtete Modell ist ein dynamisches Modell, da die Zustandsvariablen des Modells zeitabhängig sind.
- Deterministische vs. Stochastische Modelle. Die Abhängigkeiten in diesem Modell sind alle deterministischer Natur. Das Verhalten der einzelnen Komponenten ist somit nicht von Zufallsgrößen abhängig.
- Kontinuierliche vs. Diskrete Modelle. In einem kontinuierlichen Modell ändern sich die Zustandsvariablen kontinuierlich mit der Zeit. Ein Beispiel ist die Temperatur des Ofens. In einem diskreten Modell hingegen ändern sich die Zustandsvariablen ohne Verzögerung und nur zu bestimmten Zeitpunkten. Hier ist das beispielsweise bei der Anzahl an Entitäten in den verschiedenen Blöcken der Fall. Das Modell fällt also in beide Kategorien, da es diskrete und kontinuierliche Zustandsvariablen enthält. Das Simulationsmodell ist also ein hybrides Modell.

2.5.2 Typ des Optimierungsproblems

Die Lösungsvektoren (die Einflussgröße der Optimierung) bestehen, wie in Kapitel 2.2 beschrieben, aus vier Teilvektoren. Alle Einträge der Vektoren sind Zeiten. Der erste Teilvektor betrifft das Lager und gibt die Zeitpunkte der Auslagerung der einzelnen Lose an. Der zweite Teilvektor gibt die Startzeiten des Mergers an und ist durch eine Nebenbedingung an den ersten Teilvektor geknüpft. Die letzten beiden Teilvektoren betreffen den Ofen bzw. den Froster und geben An- und Abschaltzeitpunkte an.

Der Typ des Einflusses der Optimierung kann als Parametervariation und Permutation aufgefasst werden, wobei die Einflussgröße nicht als Permutation dargestellt ist, sondern als reelwertiger Vektor. Trotzdem wird durch die Festlegung der Startzeitpunkte auch die Reihenfolge der Lose festgelegt, weshalb das Problem vor allem als Reihenfolgeproblem betrachtet werden kann. Neben der Reihenfolge der Lose sollen außerdem die Einsteuerungszeitpunkte sowie die An- und Abschaltzeitpunkte der Aggregate optimiert werden. Letztere beeinflussen vor allem den Energiekostenanteil der Zielfunktion.

Die Zuordnung der Aufträge (Produkte) zu den Maschinen ist ein Problem der Maschi-

nenbelegungsplanung (Scheduling). Für Scheduling-Probleme hat sich ein einheitliches Klassifikationschema, das auf die Arbeit von Graham [23] zurückgeht, etabliert. Die Probleme werden anhand der Maschinencharakteristik, der Auftragscharakteristik und der Zielsetzung beschrieben.

Aufgrund der Anordnung der Maschinen (Maschinencharakteristik) ähnelt das hier betrachtete Problem einem Flow-Shop-Scheduling Problem. Bei einem allgemeinem Flow-Shop-Problem müssen n Aufträge auf m Maschinen in Serie bearbeitet werden. Jeder Job besteht aus m Operationen. Die erste Operation wird auf Maschine 1 durchgeführt, die zweite Operation auf Maschine 2, usw.. Alle Aufträge folgen also der gleichen Bearbeitungssequenz. Neben dem gewöhnlichen Flow-Shop-Problem gibt es noch zahlreiche Variationen, z.B. Hybride Flow-Shops, bei denen die Anzahl an parallelen Maschinen in jeder Bearbeitungsstufe ≥ 1 ist. Ruiz und Vazquez-Rodriguez bieten beispielsweise eine Übersicht über die Literatur zu diesem Thema [49]. In diesem Anwendungsfall können sich die einzelnen Produkte nicht überholen, daher ist die Bearbeitungsreihenfolge auf jeder Maschine identisch. Dieser Spezialfall wird als Permutations-Flow-Shop-Scheduling Problem (PFSSP) bezeichnet [43].

Ein wesentlicher Unterschied zu den in der Literatur häufig behandelten, konventionellen PFSSPs ist die Miteinbeziehung der Energiekosten in die Optimierung. Ein Scheduling-Problem mit einer ähnlichen Zielsetzung wird beispielsweise von Rager [45] behandelt. Nach einer Analyse von Optimierungsmethoden aus der Literatur kommt Rager zum Schluss, dass weder Heuristiken für spezielle Problemkategorien, noch exakte Methoden anwendbar sind. Die hohe Komplexität des Problems und die andersartige Zielsetzung erfordern eine Metaheuristik (GA) als Verbesserungsverfahren.

Um den Energieverbrauch des Systems zu simulieren, wird ein hybrides Simulationsmodell verwendet, was ein weiterer Unterschied zu Beispielen aus der Literatur ist. Das Optimierungsverfahren muss neben der Lösung des Scheduling-Problems auch eine möglichst optimale Steuerung der Aggregate (Ein- und Ausschaltzeitpunkte) sicherstellen und muss daher mit der hybriden Simulation interagieren können.

Eine weitere Abweichung von konventionellen PFSSPs betrifft die Anordnung der Maschinen. Die Produktionsanlage kann zwei verschiedene Produkte herstellen, wodurch es jeweils eine Weiche vor und nach den Aggregaten Ofen und Froster gibt (siehe Abbildung 2.5). Beim PFFSP werden hingegen alle Produkte auf denselben Maschinen bearbeitet.

Die Eigenschaften des Optimierungsproblems verhindern eine Klassifierung nach dem oben beschriebenen Schema von Graham. Trotzdem können einige Überlegungen zur Komplexität des Problems angestellt werden.

Komplexität des Optimierungsproblems. Flow-Shop-Scheduling-Probleme gehören (bis auf wenige Ausnahmen) der Komplexitätsklasse NP an [35]. NP (nichtdeterministisch-polynomielle Zeit) bedeutet, dass die Zeit, die zur Lösung eines Problems benötigt wird, exponentiell mit den Inputfaktoren (z.B. Stellgrößen) anwächst. Wächst die Zeit hingegen polynomiell an, gehört das Problem der Klasse P an. Für eine genauere Definition wird z.B. auf [55] verwiesen. Probleme der Klasse P sind effizient lösbar, jene der Klasse NP hingegen nicht. Das bedeutet, dass kein Algorithmus existiert, der diese Probleme ohne exponentiell anwachsende Rechenzeit exakt lösen kann. Diese Probleme können mithilfe von Heuristiken nur näherungsweise gelöst werden [30].

Wird das Reihenfolgeproblem des Anwendungsbeispiels dieser Arbeit losgelöst vom restlichen Optimierungsproblem betrachtet, indem z.B. die Lose in fixen Intervallen eingesteuert werden und ein fixer Zeitplan für die Laufzeiten der Maschinen definiert wird, kann die Einflussgröße auf eine Permutationsvariable reduzieren werden (die Komplexität des Problems wird dadurch stark reduziert). Trotzdem existieren dann immer noch $n!$ verschiedene Lösungsmöglichkeiten (n entspricht der Anzahl an zu fertigenden Losen). Im ursprünglichen Beispiel entspricht das der Anzahl an Einträgen im ersten Teil-Lösungsvektor (Lagersteuerung). Für das Testszenario in Tabelle 2.2 (14 Lose) existieren $14! \approx 8.72 * 10^{10}$ verschiedene Lösungsmöglichkeiten. Ein Simulationslauf dauert mit einem Intel Core i7-4702MQ ca. $0.75s$. Eine sequentielle Auswertung aller Möglichkeiten würde ca. 2073 Jahre dauern. Fakultäten kann man mithilfe der Stirlingschen Formel näherungsweise berechnen gemäß Königsberger [28]:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \tag{2.9}$$

Das Problem wächst also exponentiell und nicht polynomiell mit der Losanzahl n an. Daraus und aus dem Wissen, dass bereits wesentlich einfachere Flow-Shop-Scheduling Probleme (siehe [35]) NP-schwer sind, kann gefolgert werden, dass auch das in dieser Arbeit behandelte Optimierungsproblem der Klasse NP angehört und dadurch nur mittels Metaheuristiken näherungsweise gelöst werden kann.

2.6 Vergleichbare Anwendungen simulationsbasierter Optimierung

In Kapitel 2.2 wurden fünf Algorithmenklassen für die simulationsbasierte Optimierung vorgestellt:

1. Stochstische Approximation
2. Response Surface Methoden
3. Random Search
4. Sample Path Optimierung
5. Metaheuristiken

Die meisten Algorithmen der ersten vier Kategorien setzen eine unimodale Fitnesslandschaft voraus und sind daher nur für lokale Optimierung geeignet [53]. Aufgrund der eingeschränkten Eignung der ersten vier Algorithmenklassen für komplexe (NP-schwere) Optimierungsprobleme mit Zielfunktionen, die multiple lokale Minima aufweisen, kommen in kommerzieller Optimierungssoftware eher universell einsetzbare Metaheuristiken und hier insbesondere populationsbasierte Algorithmen zum Einsatz (siehe Tabelle 2.1). Ein Nachteil der Metaheuristiken im Vergleich zu den anderen Methoden ist jedoch die nicht beweisbare Konvergenz gegen lokale oder globale Optima [7]. Zu diesen Verfahren zählen z.B. Genetische Algorithmen (GAs), Scatter Search (SS) und Particle Swarm Optimization (PSO).

Neben evolutionären Verfahren finden noch einige andere Methoden Anwendung in der simulationsbasierten Optimierung. Amaran u.a. [2] bieten eine aktuellere Übersicht über die eingesetzten Algorithmen. Dort wird ebenfalls festgestellt, dass in kommerziellen Optimierungspaketen hauptsächlich populationsbasierte Metaheuristiken eingesetzt werden. Dies hat zur Folge, dass beim Großteil aller praktischen Anwendungen ebenfalls diese Algorithmen zum Einsatz kommen.

Erfolgreiche Anwendungsbeispiele populationsbasierter Metaheuristiken, die dem Anwendungsbeispiel dieser Arbeit ähnlich sind, werden unter anderem in März u.a.[30] oder in [31] beschrieben.

2.7 Auswahl geeigneter Algorithmen

In Abbildung 2.7 sind die Anwendungsbereiche verschiedener SBO-Algorithmen dargestellt. Ein wichtiges Auswahlkriterium für die vorliegende Problemstellung ist die Fähigkeit zur globalen Optimierung. Dieses wird vorwiegend von Metaheuristiken erfüllt.

Aus diesem Grund und aus den bereits weiter oben erwähnten Gründen (universellere Einsetzbarkeit, einfachere Implementierung, häufige erfolgreiche Anwendung in der Praxis und die Anwendbarkeit bei Black-Box Optimierung) wird daher im Weiteren nur auf Metaheuristiken eingegangen.

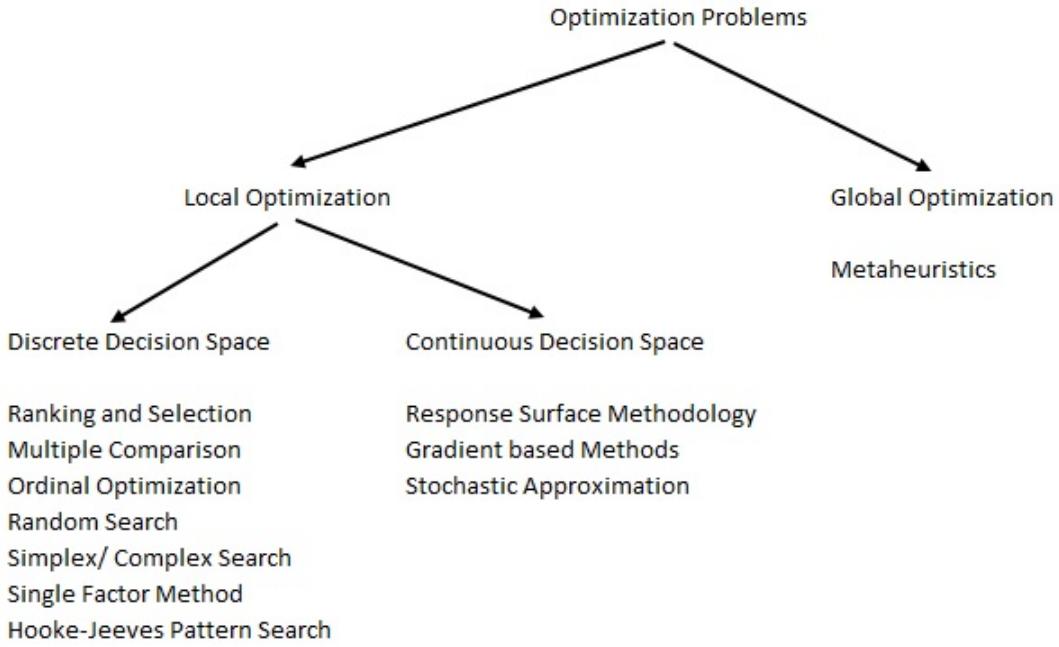

Abbildung 2.7: Anwendungsbereich verschiedener SBO Algorithmen [53]

Innerhalb der Klasse der Metaheuristiken sind vor allem populationsbasierte Verfahren gut für das Optimierungsproblem geeignet, da deren Stärke die Identifikation von Bereichen mit guten Lösungen in höherdimensionalen Lösungsräumen ist. Bei der Auffindung lokaler Optima in diesen Bereichen sind sie jedoch weniger effizient [8]. Eine Kombination mit einer Local-Search Methode ist daher sinnvoll.

Eine Entscheidung für einen konkreten Algorithmus für das vorliegende Problem gestaltet sich schwierig, da es derzeit keine umfangreichen Leistungsvergleiche diverser Algorithmen für verschiedene (simulationsbasierte) Testumgebungen gibt [2]. In der Literatur findet man eher konzeptuelle Vergleiche, wie in Eberhart [13] oder Vergleiche zwischen zwei oder einer kleinen Gruppe von Algorithmen bei der Anwendung auf ein

konkretes Problem, wie in Amodeo [3]. Diese Arbeiten lassen aber kaum einen Schluss über die Leistung bei der Anwendung auf das vorliegende Problem zu, da sich die dort behandelten Probleme zu stark von diesem Anwendungsbeispiel unterscheiden. Zusätzlich wird die Entscheidung für einen Algorithmus durch die starke Abhängigkeit der Effizienz von der konkreten Implementierung und der Parameterauswahl erschwert. Die Leistungsfähigkeit wird vor allem durch die konkrete, an das Problem angepasste, Umsetzung der Methoden und der Auswahl der Stellgrößen beeinflusst. Außerdem ist eine an das Problem angepasste Auswahl der Algorithmenparameter entscheidend. Die Laufzeit der Simulation beeinflusst die Leistungsfähigkeit der Optimierung ebenfalls wesentlich. Gerade beim Einsatzvon populationsbasierten Verfahren sollte die Laufzeit möglichst kurz sein, um ausreichend viele Bewertungen während der meist zeitlich beschränkten Optimierung zu ermöglichen.

Aufgrung der guten Anpassungsmöglichkeiten, der guten Eignung für höherdimensionale Probleme, der Anwendbarkeit für Black-Box Optimierung und aufgrund der häufigen, erfolgreichen Anwendung in der Praxis kann man die Entscheidung auf populationsbasierte Metaheuristiken eingrenzen.

3 Vergleich und Anpassung von Particle Swarm Optimization (PSO) und Genetischer Algorithmen (GAs)

In diesem Kapitel werden die populationsbasierten Metaheuristiken PSO und GA an dem in Kapitel 2.4 beschriebenen Anwendungsbeispiel hinsichtlich Leistungsfähigkeit (Dauer der Optimierung und Qualität der Lösungen) getestet. Die Simulation ist in MATLAB umgesetzt (siehe [50]). Für die Experimente wurden daher die Metaheuristiken der MATLAB Global Optimization Toolbox verwendet.

3.1 Genetischer Algorithmus in MATLAB

Der GA als Bestandteil der Global Optimization Toolbox von MATLAB R2016b funktioniert im Wesentlichen wie in Kapitel 2.1.3 beschrieben, bietet aber viele Optionen und Anpassungsmöglichkeiten. Im Folgenden wird die grundlegende Funktionsweise noch einmal beschrieben[4]:

1. Erstellung einer Startpopulation.
2. Der Algorithmus erstellt anschließend eine Reihe weiterer Populationen aus den jeweils aktuellen Individuen. Dazu werden die folgenden Schritte durchlaufen:
   (a) Berechnung der Fitnesswerte aller Individuen (durch die Simulation).
   (b) Skalierung der Fitnesswerte zur Aufbereitung für eine Auswahlprozedur.
   (c) Auswahl der Eltern mithilfe eines Selektions-Operators.
   (d) Einige Individuen der aktuellen Population mit den besten Fitnesswerten werden als Eliteindividuen in die nächste Generation übernommen.
   (e) Erzeugung von Kindern aus den Eltern-Individuen. Dies geschieht entweder durch die Kombination von zwei Eltern durch Rekombination oder durch Änderung eines Individuums durch einen Mutations-Operator.
   (f) Ersetzen der aktuellen Population durch die erzeugten Individuen und die Eliteindividuen.
3. Der Algorithmus stoppt, wenn ein Abbruchkriterium erfüllt ist.

[4] Vgl. http://de.mathworks.com/products/global-optimization (Gelesen am: 22.08.2016)

MATLAB bietet für alle erwähnten Operationen Einstellungsmöglichkeiten. Die Crossover-Funktion und die Mutations-Funktion können selbst definiert werden. Alle Optionen, die in den Experimenten verwendet wurden, werden in den entsprechenden Abschnitten erläutert. Eine vollständige Beschreibung aller standardmäßigen Einstellungsmöglichkeiten findet sich in der Global Optimization Toolbox Dokumentation.

3.2 Particle Swarm Optimization in MATLAB

Die Funktionsweise des PSO-Algorithmus der Global Optimization Toolbox ist folgende:

1. Erstellung einer Startpopulation und Zuweisung von Startgeschwindigkeiten.
2. Bewertung der Startlösungen durch die Simulation.
3. Zuweisung neuer Geschwindigkeiten basierend auf der aktuellen Partikelgeschwindigkeit, der bisher besten Position des Partikels und der bisher besten Position des Schwarms nach der Formel (komponentenweise):

$$v = W * v + y_1 * u_1 * (p - x) + y_2 * u_2 * (g - x) \tag{3.1}$$

   v : vorherige Geschwindigkeit
   W : Trägheitsparameter
   y_1 : Gewichtung für die Selbstadjustierung
   y_2 : Gewichtung für die Schwarmadjustierung
   u_1, u_2 : Gleichverteilte Zufallszahlen im Intervall [0,1]
   $p - x$: Differenz zwischen der aktuellen und besten Position des Partikels
   $g - x$: Differenz zwischen der aktuellen und besten Position des Schwarms

4. Aktualisierung der Positionen x (komponentenweise) nach:

$$x = x + v \tag{3.2}$$

5. Kontrolle der Schranken (für die Einträge des Lösungsvektors). Liegt eine Komponente außerhalb der Schranken, wird sie dem Schrankenwert gleichgesetzt.
6. Ermittlung der Fitnesswerte mittels Simulation und Zielfunktion $f = fun(x)$
   (a) Wenn $f < fun(p)$ setze $p = x$. Dadurch wird die aktuelle Position zur bisher

besten Position des Partikels.

(b) Wenn f kleiner ist als der bisher gefundene beste Zielfunktionswert b, dann setze $b = f$ und $d = x$ (Position der besten gefundenen Lösung im Schwarm).

7. Wenn b im letzten Schritt verringert wurde:

   (a) Setze den Zähler für erfolglose Iteration $c = max(0, c - 1)$.

   (b) Wenn $c < 2$ dann setze $W = 2 * W$

   (c) Wenn $c > 5$ dann setze $W = W/2$

   (d) Sicherstellen, dass W innerhalb der definierten Grenzen liegt.

   Anderenfalls setze $c = c + 1$.

Die Einstellungsmöglichkeiten beschränken sich im Wesentlichen auf die Parameter:

- Vorgabe eines Intervalls für den Trägheitsparameter W
- Gewichtung für die Selbsadjustierung y_1
- Gewichtung für die Schwarm- bzw. Nachbarschaftsadjustierung y_2
- Nachbarschaftsgröße. Wenn bei den Iterationen die Bestimmung der neuen Geschwindigkeiten nicht abhängig vom ganzen Schwarm, sondern nur von einem Teil davon sein soll, kann eine Nachbarschaftsgröße definiert werden. Die Nachbarschaft wird für jedes Partikel zufällig als Teilmenge des Schwarms gewählt.
- Populationsgröße (Schwarmgröße)

3.3 Vergleich von GA und PSO mit Standardeinstellungen

In diesem Abschnitt werden GA und PSO mit den Standardeinstellungen der MATLAB Global Optimization Toolbox hinsichtlich Leistungsfähigkeit (Optimierungsdauer, Qualität der Lösungen) verglichen. Als Testszenario dient das Szenario aus Tabelle 2.2. (7-tägiger Simulationszeitraum).

3.3.1 Einstellungen der Algorithmen

Die verwendeten Standardeinstellungen der Algorithmen sind:

GA

- Populationsgröße $P = 200$.
- Fitnesswertskalierung: Rang-proportional. Die von der Simulation erhaltenen Fitnesswerte der Lösungen werden nach ihrem Wert gereiht. Die skalierten Fitnesswerte f_s werden anschließend dem Rang r entsprechend berechnet mit $f_s = 1/\sqrt{r}$. Der skalierte Fitnesswert ist also nur abhängig von Rang der Lösung. Dadurch wird die Chance sehr schlechter Lösungen für die Rekombination oder Mutation erhöht. Außerdem wird verhindert, dass gute Lösungen unverhältnismäßig oft als Eltern gewählt werden.
- Selektionsmethode: Stochastic Universal Sampling. Dieses Verfahren ist äquivalent dem Abschreiten einer Stecke mit gleichgroßen Schritten, wobei die Strecke in Abschnitte proportional den skalierten Fitnesswerten unterteilt ist. Jene Individuen, in deren Abschnitt ein Schritt gesetzt wird, werden ausgewählt.
- Eliteindividuen: Die Anzahl der Elitelösungen wird mit $ceil(0.05 * Populationsize)$ ermittelt. Diese Lösungen werden unverändert in die nächste Generation übernommen.
- Crossoveranteil: definiert den Anteil der Population, welcher durch Crossover erzeugt wird (ohne Eliteindividuen). Standardwert $C = 0.8$
- Crossovermethode: Intermediate. Diese Crossoverfunktion erzeugt Lösungskandidaten $\vec{x}_{child}$ aus zwei Elternindividuen $\vec{x}_{parent1}$, $\vec{x}_{parent2}$ und einer Zufallszahl $rand$ im Intervall $[0, 1]$ folgendermaßen:

  $$\vec{x}_{child} = \vec{x}_{parent1} + rand * (\vec{x}_{parent2} - \vec{x}_{parent1}) \tag{3.3}$$

  Der resultierende Vektor ist eine Linearkombination (konvexe Kombination) der Elternindividuen.
- Mutationsmethode: Adaptive Feasable. Unter Rücksichtnahme auf vorherige, erfolgreiche Mutationen werden Richtungen und Schrittweiten für die Mutation ermittelt. Rand- und Nebenbedingungen werden dabei eingehalten.
- Abbruchkriterien: Beenden den Optimierungsprozess, sobald ein Kriterium erfüllt ist. Diese sind bei Verwendung der Standardoptionen:
  - Maximale Anzahl an Generationen ($100 * Variablenanzahl = 5700$ für das

Testszenario)

– Maximale Anzahl an Generationen ohne Verbesserung (50)

PSO

- Populationsgröße $P = 100$
- Intervall für den Trägheitsparameter $[0.1; 1.1]$
- Gewichtung für die Selbstadjustierung $y_1 = 1.49$
- Gewichtung für die Nachbarschaftsadjustierung $y_2 = 1.49$
- Nachbarschaftsgröße: ist als $0.25 * P = 25$ festgelegt.

3.3.2 Nebenbedingungen der Optimierung

Um den Lösungsraum einzuschränken und damit die Anzahl an Lösungen mit nicht fertiggestellten Produkten (hohe Gewichtung in der Zielfunktion) sowie nicht realisierbarer Lösungen zu reduzieren, werden folgende lineare Nebenbedingungen verwendet:

- Der Merger stoppt erst nach allen Merger-Starts.
- Der Merger startet erst 0.5h nach der Auslagerung des jeweiligen Produktionsloses.
- Ofen und Froster stoppen erst nach deren Start und einer minimalen Betriebsdauer (Bearbeitungszeit für das kleinste Los auf dem jeweiligen Aggregat). Dadurch wird eine Überlappung der Betriebszeiten auf den Aggregaten verhindert.
- Der Merger stoppt erst, nachdem alle Produktionslose im Lager gestartet sind.
- Die Summe der Betriebszeiten von Ofen und Froster muss größer gleich der kumulierten minimalen Bearbeitungszeit (Zeit die für die Bearbeitung aller Produkte) sein. Diese Beziehung gilt je Aggregat.

Diese Nebenbedingungen werden in der MATLAB Global Optimization Toolbox R2016b jedoch nur vom GA unterstützt und können daher nicht für den PSO-Test verwendet werden. Für die Lösungsvektoreinträge werden für beide Algorithmen obere (168h) und untere Schranken (0h) verwendet.

3.3.3 Startlösungen

Um die Resultate besser vergleichbar zu machen, wird für beide Algorithmen die gleiche Startpopulation verwendet, welche mit zwei gültigen Startlösungen befüllt ist.

3.3.4 Ergebnisse

In Abbildung 3.1 ist der Verlauf der Fitnesswerte für jeweils fünf Optimierungsläufe dargestellt (Testszenario für $t = 7d$ in Tabelle 2.2). Die Fitnesswerte werden auf das bisher bekannte Minimum für das Testszenario bezogen. Das bisher bekannte Mininmum wurde von bereits vorhandenen Daten für die Arbeit übernommen.

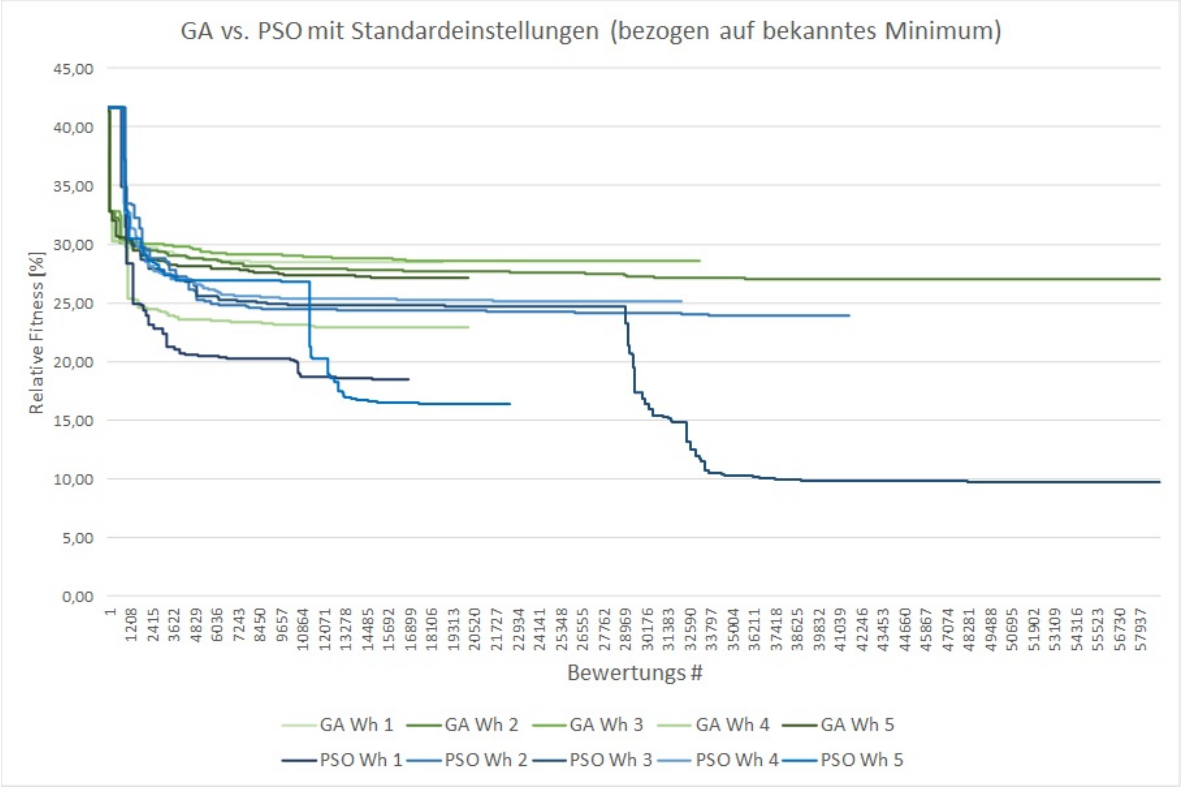

Abbildung 3.1: Relative Fitness GA vs. PSO mit Standardeinstellungen

Weder GA noch PSO erreichen das bisher bekannte Minimum für das Testszenario. Die Ergebnisse sind vor allem beim PSO von unterschiedlicher Qualität, sind aber durchschnittlich besser als die des GA. Die Optimierung stoppt aufgrund der Erfüllung von Abbruchkriterien bei beiden Algorithmen zu unterschiedlichen Zeitpunkten. Beim PSO-Algorithmus stagniert der Optimierungsprozess außerdem in drei von fünf Fällen sehr lange, bevor weitere Verbesserungen erzielt werden.

In Abbildung 3.2 ist der Verlauf der Teil-Fitnesswerte während des Optimierungsvorgangs mit dem besten Endergebnis dargestellt (PSO Wiederholung 3).

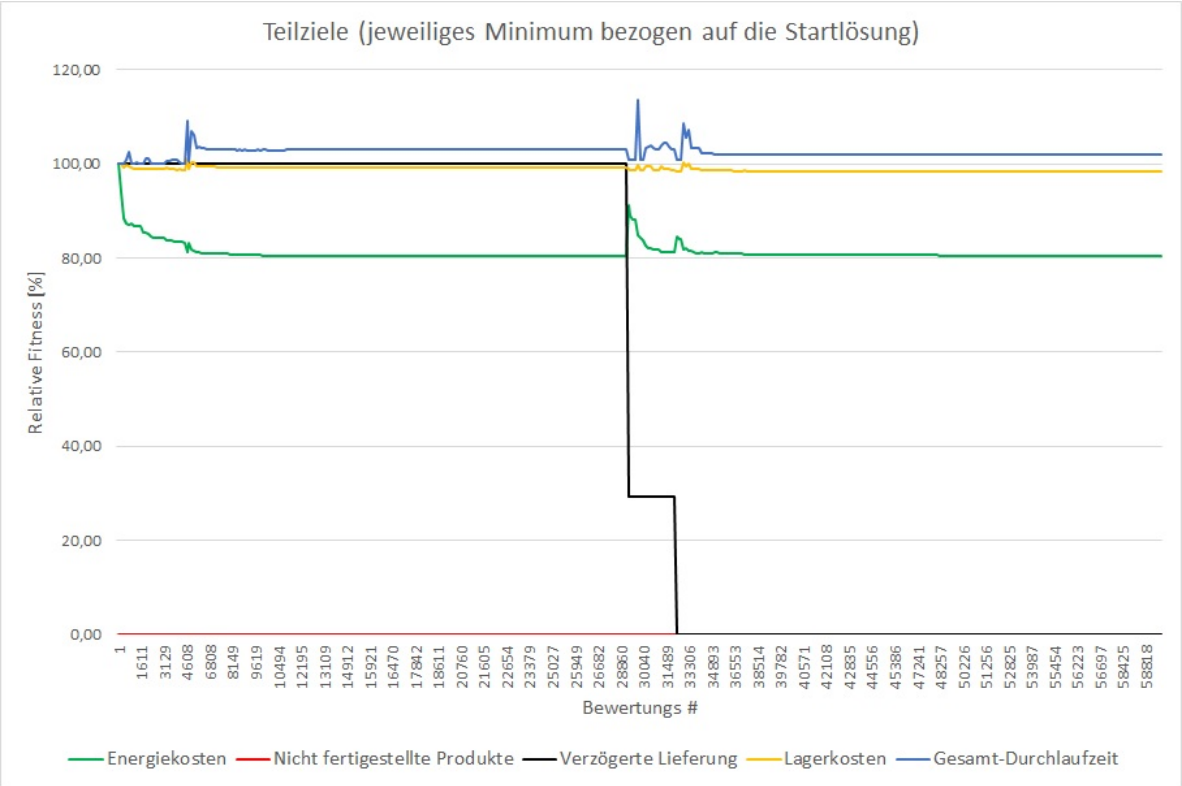

Abbildung 3.2: Teilziele PSO Wiederholung 3

In der Anfangsphase der Optimierung wird hauptsächlich der Energieverbrauch optimiert. Eine Lösung, welche den Lieferverzug minimiert, wird erst nach ca 30.000 Bewertungen gefunden. Der Zielfunktionsanteil für nicht fertiggestellte Produkte wird wegen der hohen Gewichtung stets bei 0 gehalten. Dieser Teil-Fitnesswert wird daher in weiterer Folge als Identifikationsmerkmal für nicht praktikable (bzw. schlechte) Lösungen verwendet. Die Lagerhaltungskosten und die Durchlaufzeit werden nur geringfügig verbessert, bzw. verschlechtert. Dies liegt an der geringeren Gewichtung in der Zielfunktion.

Der Anteil an Lösungen mit nicht fertiggestellten Produkten ist beim PSO-Algorithmus wesentlich größer als beim GA (siehe Tabelle 3.1). Dies ist darauf zurückzuführen, dass PSO im Gegensatz zum GA keine linearen Nebenbedingungen akzeptiert. Die Nebenbedingungen verhindern impraktikable Lösungen.

	Bester relativer Fitnesswert [%]	Bewertungen (Simulationsläufe)	Lösungen mit nicht fertiggestellten Produkten [%]
GA (Wh. 1)	28.49	18800	2.55
GA (Wh. 2)	27.03	59100	11.91
GA (Wh. 3)	28.53	33200	0.11
GA (Wh. 4)	22.88	20200	0.94
GA (Wh. 5)	27.15	20200	0.12
PSO (Wh. 1)	18.51	16900	26.87
PSO (Wh. 2)	23.91	41600	12.89
PSO (Wh. 3)	9.72	59100	31.21
PSO (Wh. 4)	25.13	32200	16.13
PSO (Wh. 5)	16.36	22600	17.46

Tabelle 3.1: Ergebnisse GA vs. PSO mit Standardeinstellungen

Abbildung 3.3 zeigt die jeweils besten gefundenen Lösungsvektoren beider Algorithmen, sowie die beste Startlösung für das Testszenario aus Tabelle 2.2. Der GA hat die Reihenfolge der Produktionslose und die Betriebszeiten im Vergleich zur Startlösung nur geringfügig verändert. Die Lösung entspricht den oben beschriebenen Nebenbedingungen. Der PSO-Algorithmus hat die Ausgangslösung in stärkerem Maße verändert. Sowohl die Losreihenfolge als auch die Betriebszeiten der Aggregate wurden besser angepasst als vom GA. Die Lösung verletzt aber Nebenbedingungen, wodurch es zu Überlappungen von Betriebszeiten auf den Aggregaten kommt. Solche Lösungen werden zwar von der Simulation ohne Fehlermeldung akzeptiert, liefern aber in weiterer Folge keine sinnvollen Produktionspläne.

Aufgrund der Unterstützung von linearen Nebenbedingungen, sowie selbst zu definierender Operatoren (Mutations- und Crossoverfunktion) wird der GA für weitere Anpassungen ausgewählt.

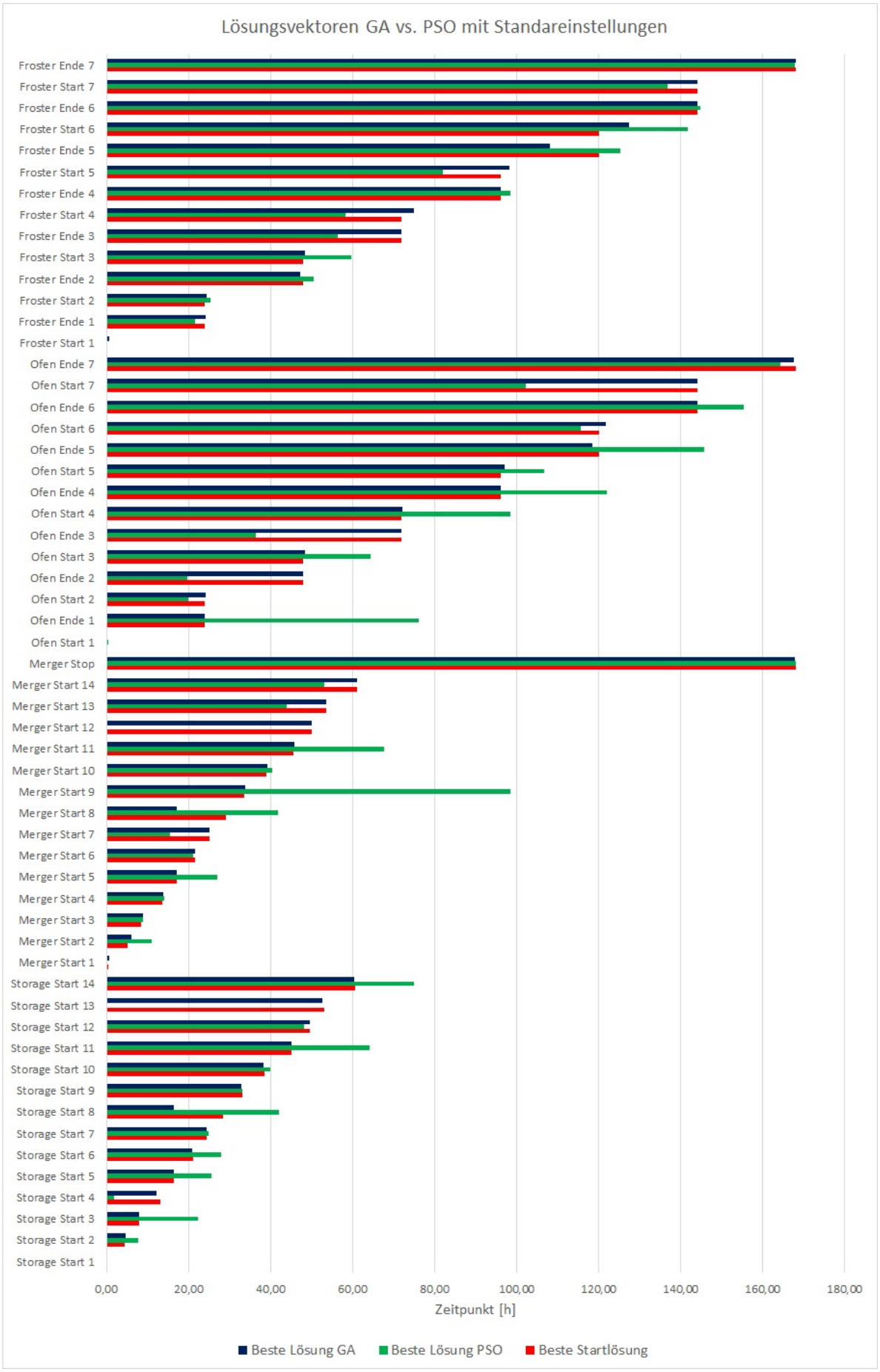

Abbildung 3.3: Beste Lösungsvektoren GA vs. PSO mit Standardeinstellungen (für das Testszenario aus Tabelle 2.2)

3.4 Vergleich verschiedener GA Varianten

Die Ergebnisse in Kapitel 3.3.4 zeigen, das der GA mit Standardeinstellungen nicht in der Lage ist, die Reihenfolge der Produktionslose und die Betriebszeiten der Aggregate in ausreichendem Maße anzupassen, was eine Adaption der Algorithmus erforderlich macht. Viele standardmäßige Einstellungsmöglichkeiten des GA der MATLAB Global Optimization Toolbox sind jedoch nicht für die Verwendung mit Nebenbedingungen geeignet oder würden durch die spezielle Struktur des Lösungsvektors zu vielen ungültigen Lösungen führen. Die Unterteilung des Lösungsvektors in vier Abschnitte ist vor allem bei Anwendung des Crossover-Operators problematisch. Ein unkontrollierter Austausch von Lösungsvektoreinträgen (vor allem außerhalb der definierten Abschnitte) ist nicht sinnvoll, daher wird auf dessen Anwendung verzichtet. Stattdessen wird (angelehnt an die Methoden aus [50]) ein selbst definierter Mutations-Operator verwendet, der eine zielgerichtete Suche unter Einhaltung von Nebenbedingungen ermöglicht. Die Mutation umfasst:

- Die Vertauschung der Produktionslosreihenfolge im ersten Teillösungsvektor (Lager) und Übertragung des Tauschs auf den Merger.
- Die Veränderung der Einsteuerungszeitpunkte von Produktionslosen im ersten Teillösungsvektor durch zufällige Mutation.
- Die gezielte Verkürzung der Betriebszeiten von Ofen und Froster. Dazu werden Startzeitpunkte verzögert und Maschinenstopps vorverlegt.
- Das zufällige Verschieben der Betriebszeiten von Ofen und Froster (paarweises Verschieben von Start- und Stoppzeitpunkten).

Die Umsetzung dieser Operationen erfolgt in mehreren Varianten, welche im nächsten Abschnitt erklärt werden.

3.4.1 Getestete Optionen

Neben mehreren Varianten der Mutationsfunktion werden auch einige, in der Literatur häufig beschriebene, Optionen für die Skalierung der Fitnesswerte und die Elternselektion getestet (siehe z.B. [56], [58], [25], [37]).

Skalierungsmethoden Die Skalierung dient zur Aufbereitung der Fitnesswerte für die Elternselektion. Dadurch kann z.B. beeinflusst werden, wie mit sehr schlechten Lösungen umgegangen wird. Ergebnis der Skalierung ist eine Zuweisung von Wahrscheinlichkeiten für die Selektion zu den Individuen einer Population. Die getesteten Optionen sind:

- Skalierung proportional dem Rang des Fitnesswerts in der Population (Beschreibung in Kapitel 3.3.1).
- Fitness-proportionale Skalierung. Hier wird den Individuen eine Wahrscheinlichkeit proportional ihrem ursprünglichen Fitnesswert zugewiesen. Die Wahrscheinlichkeit einer Selektion für sehr schlechte Lösungen ist hier geringer als bei einer Skalierung nach Rang.
- Top-Skalierung. Diese Methode weist den besten n Individuen der Population die gleiche Wahrscheinlichkeit für die Selektion zu, während alle anderen Individuen ausscheiden (Wahrscheinlichkeit 0). Für den Test wurde $n = 10$ gesetzt (Matlab Standardeinstellung).

Selektionsmethoden Die Selektionsmethode wählt jene Individuen aus der aktuellen Population (basierend auf deren skalierten Fitnesswerten), die für die Erzeugung von Nachkommen (neuen Lösungen) verwendet werden. Getestete Optionen sind:

- Fitness-proportionale Selektion (Roulette Wheel Selection). Funktioniert wie ein Roulette-Rad, das in Abschnitte proportional der skalierten Fitnesswerte aufgeteilt ist. Jener Abschnitt in dem die (fiktive) Kugel landet, wird ausgewählt. Dies wird wiederholt, bis die gewünschte Anzahl an Elternindividuen erreicht ist.
- Stochastic Universal Sampling (Beschreibung in Kapitel 3.3.1).
- Turnierselektion. Hier werden zufällig n Individuen aus der Population gewählt. Anschließend werden ihre skalierten Fitnesswerte verglichen und die beste Lösung ausgewählt. Der Vorgang wird wiederholt, bis die benötigte Elternanzahl erreicht wird. Für den Test wurde $n = 4$ gewählt (Matlab Standardeinstellung).

Mutationsmethoden für die Reihenfolge der Produktionslose Die Reihenfolge der Produktionslose wird durch die Startzeitpunkte des ersten Teillösungsvektors (Lager) bestimmt. Die Ergebnisse aus Kapitel 3.3 haben gezeigt, dass die Reihenfolge

während des Optimierungsvorgangs des GA nur geringfügig verändert wurde und der Lösungsraum daher nicht ausreichend erforscht wurde. Einer der Operationen der Mutationsfunktion soll daher die Produktionslosreihenfolge gezielt verändern. Dazu werden einige Mutationsoperatoren aus der Literatur für Reihenfolgeprobleme (siehe [56], [32]) getestet.

- Displacement Mutation wählt einen Teil des Lösungsvektors (per Zufallsgenerator) aus und verschiebt diesen an einen zufälligen Punkt des verbleibenden Vektors.

$$1\ \ 2\ \overbrace{3\ \ 4\ \ 5}^{shift\ section}\ \ 6\ \ 7 \xrightarrow{Mutation} 1\ \ 2\ \ 6\ \overbrace{3\ \ 4\ \ 5}^{shift\ section}\ \ 7$$

- Exchange Mutation vertauscht zwei zufällig ausgewählte Einträge des Lösungsvektors.

$$1\ \ 2\ \ 3\ \ 4\ \ 5\ \ 6\ \ 7 \xrightarrow{Mutation} 1\ \ 2\ \ 5\ \ 4\ \ 3\ \ 6\ \ 7$$

- Scramble Mutation wählt einen Teil des Lösungsvektors (per Zufallsgenerator) und reiht die enthaltenen Elemente nach einer zufälligen, neuen Reihenfolge.

$$1\ \ 2\ \overbrace{3\ \ 4\ \ 5\ \ 6}^{scramble\ section}\ \ 7 \xrightarrow{Mutation} 1\ \ 2\ \overbrace{5\ \ 6\ \ 3\ \ 4}^{scramble\ section}\ \ 7$$

- Insertion Mutation funktioniert wie Displacement Mutation mit dem Unterschied, dass nur ein Element verschoben wird.
- Inversion Mutation wählt einen Teil des Lösungsvektors (per Zufallsgenerator) und invertiert die Reihenfolge der enthaltenen Einträge.

$$1\ \ 2\ \overbrace{3\ \ 4\ \ 5}^{section}\ \ 6\ \ 7 \xrightarrow{Mutation} 1\ \ 2\ \overbrace{5\ \ 4\ \ 3}^{inverted\ section}\ \ 6\ \ 7$$

- Displaced Inversion Mutation funktioniert wie Displacement Mutation, der verschobene Abschnitt wird aber vor dem Einfügen invertiert.

Mutationsmethoden für die Lösungsvektoreinträge beeinflussen die Änderung der Startzeiten der Produktionslose im Lager sowie die Kontraktion der Betriebsintervalle von Ofen und Froster. Folgende Varianten werden getestet:

- Gleichverteilt mit einer maximalen Änderung von 25% des Vektoreintrags.
- Normalverteilt mit dem Mittelwert $\mu = 0$ und einer Standardabweichung $\sigma = 25\%$. Dadurch werden hauptsächlich kleinere Änderungen durchgeführt, große Änderungen sind aber mit einer kleinen Wahrscheinlichkeit ebenfalls möglich, wodurch eine bessere Erkundung des Lösungsraums gewährleistet wird. Die Standardabweichung wird nach jeder Generation um einen kleinen Faktor reduziert. Dadurch wird erreicht, dass im fortgeschrittenen Optimierungsvorgang hauptsächlich kleine Veränderungen gemacht werden. Dadurch wird ein verbessertes lokales Suchverhalten erreicht.

Sonstige Einstellungen. Neben den beschriebenen Optionen werden folgende Einstellungen für den GA verwendet:

- Die Populationsgröße wird reduziert auf $P = 25$. Der Einfluss der Populationsgröße auf die Performance des Algorithmus wird in Kapitel 3.6.1 untersucht.
- Als Abbruchkriterium wird eine maximale Anzahl an Generationen von 500 festgelegt. Insgesamt werden also 12525 Bewertungen durch die Simulation durchgeführt, was auf einem Intel Core i7-4702MQ etwa $3.5h$ in Anspruch nimmt.
- Um die Darstellung der Ergebnisse zu vereinfachen, werden die getesteten GA-Varianten nach Tabelle 3.2 codiert.

Code	Skalierungs-Methode	Selektions-Methode	Mutationsmethode (Reihenfolge)	Mutationsmethode (Wert)
1	Rang-proportional	Fitness-proportional	Displacement-Mutation	Gleichverteilt
2	Fitness-proportional	Stochastic Universal	Exchange-Mutaion	Normalverteilt
3	Top-Skalierung	Tournament-Selection	Scramble-Mutation	
4			Insertion-Mutation	
5			Inversion-Mutation	
6			Displaced Inversion	

Tabelle 3.2: Codierung der GA-Varianten

Die Bezeichnung GA-3142 steht beispielsweise für Top-Skalierung, Fitness-proportionale Selektion, Insertion-Mutation und normalverteilte Wertmutation.

- Aufgrund von Zufallseinflüssen unterliegen die Ergebnisse statistischen Fluktuationen. Abbildung 3.4 zeigt den Verlauf des Mittelwerts sowie Minimal- und Maximalwerte während des Optimierungsvorgangs von GA 1111 nach fünf Wiederholungen. Der Mittelwert bewegt sich vor allem im fortgeschrittenen Optimierungsverlauf in einem schmalen Korridor (Maximale Abweichung 2.08 Prozentpunkte nach ca. 9500 Bewertungen). Aus diesem Grund und aufgrund des hohen Zeitaufwands pro Experiment wird eine Wiederholungszahl von zwei für alle weiteren Tests festgelegt.

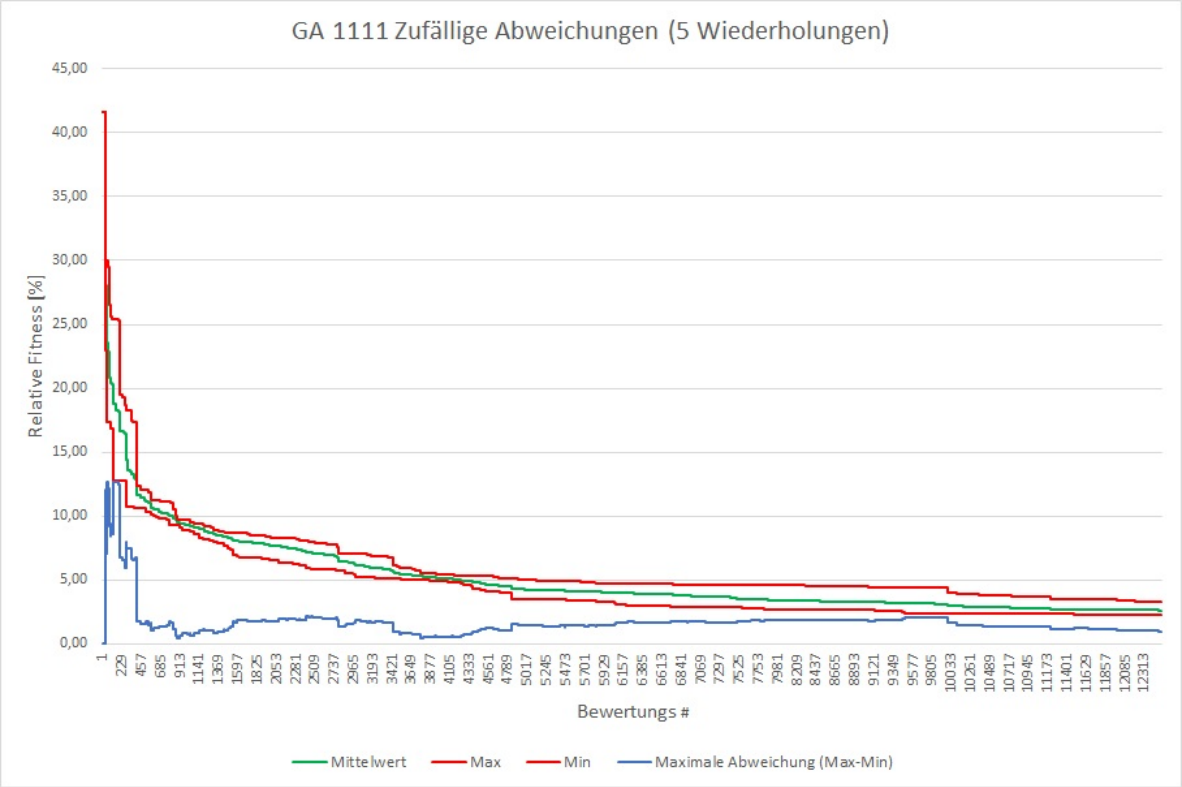

Abbildung 3.4: GA 1111 Zufällige Abweichungen

3.4.2 Ergebnisse

In den folgenden Diagrammen sind die Ergebnisse zu den Tests der verschiedenen Optionen abgebildet. Die Verläufe der relativen Fitnesswerte sind Mittelwerte. Bezugswert ist das beste bekannte Minimum für das Testszenario (0 auf der Ordinate). Neben den eigenen Ergebnissen ist jeweils der (mittlere) Verlauf des Hybrid Tuned Int(60) GA aus [50] zum Vergleich abgebildet. Ausgangspunkt für Verbesserungen ist der GA-1111.

Einfluss der Skalierungsmethode Abbildung 3.5 zeigt den mittleren Verlauf der relativen Fitnesswerte für unterschiedliche Skalierungsmethoden. Die Ergebnisse unterscheiden sich bis 500 Bewertungen kaum. Bis 3.000 Bewertungen liefert die Rangproportionale Skalierung (GA-1111) die besten Ergebnisse. Während des restlichen Optimierungsvorgangs schlägt die Top-Skalierung (GA-3111) die restlichen Methoden. Da diese während des gesamten Verlaufs gute Ergebnisse und das beste Endergebnis liefert, wird sie für weitere Verbesserungen ausgewählt.

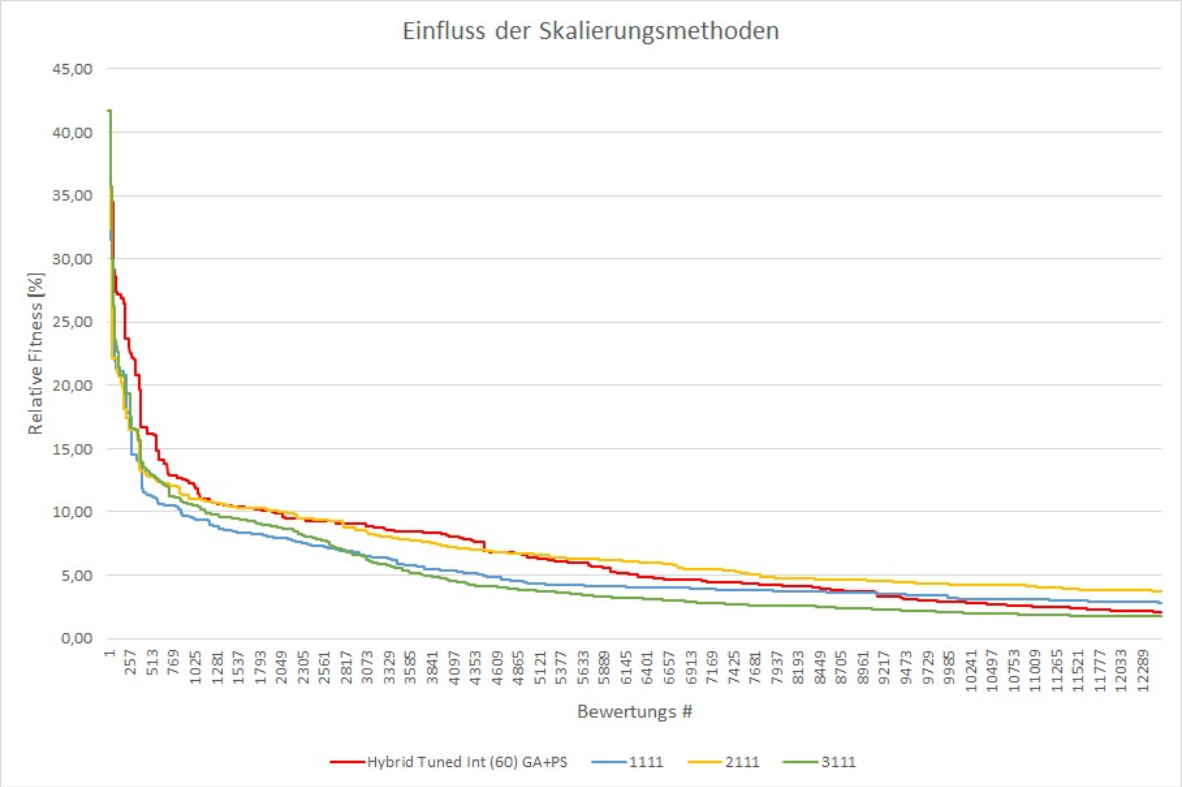

Abbildung 3.5: Einfluss der Skalierungsmethode (Mittelwert aus zwei Wiederholungen)

Einfluss der Selektionsmethode Abbildung 3.6 zeigt den Einfluss unterschiedlicher Selektionsmethoden auf den Verlauf der Fitnesswerte. Bis 3.000 liefert die Turnierselektion die besten Ergebnisse, der Vorteil gegenüber den anderen Selektionsmethoden ist aber nur gering. Während des restlichen Optimierungsvorgangs liefert die Fitnessproportionale Selektion (GA-3111) die besten Ergebnisse und wird daher für weitere Verbesserungen ausgewählt.

Einfluss der Reihenfolgemutation Die Mutationsmethode für die Reihenfolge der Produktionslose hat nur einen geringen Einfluss auf die Leistungsfähigkeit des Algorithmus (siehe Abbildung 3.7). Der GA-3151 (Scramble Mutation) liefert zu den meisten

Zeitpunkten die besten Ergebnisse und wird daher für die weitere Verbesserung ausgewählt.

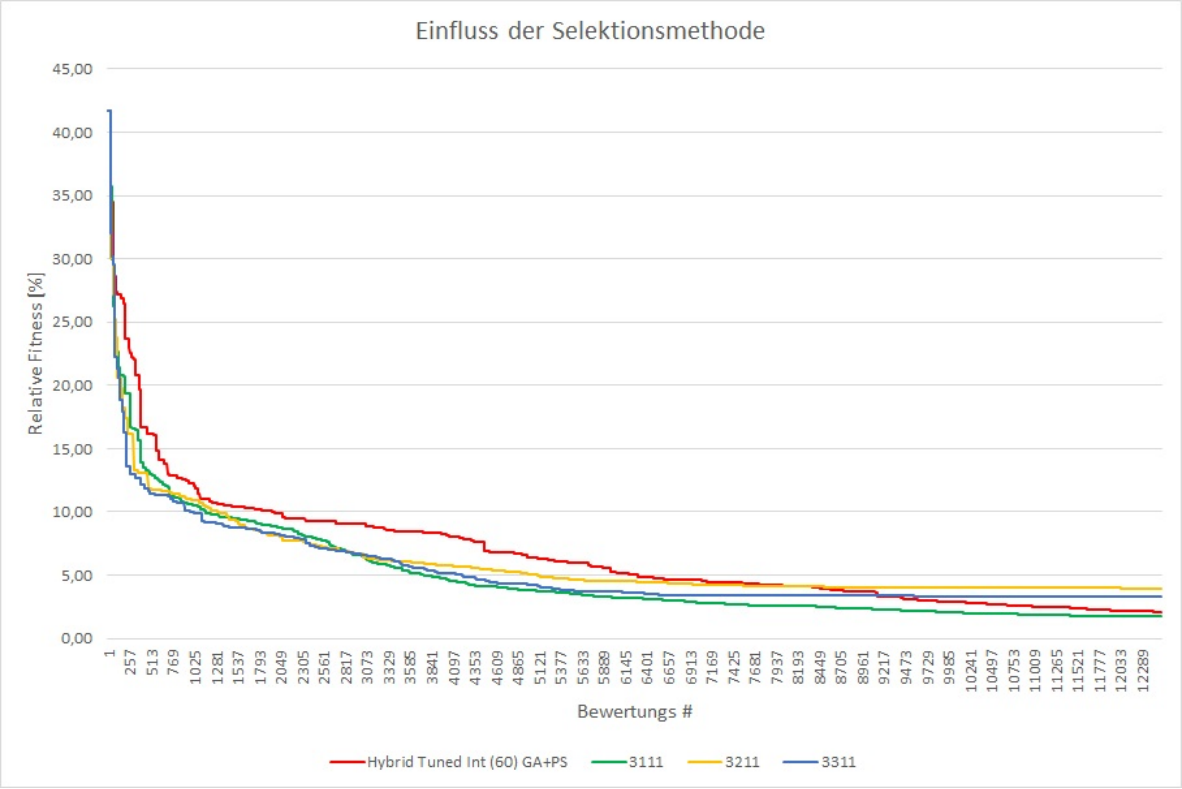

Abbildung 3.6: Einfluss der Selektionsmethode (Mittelwert aus zwei Wiederholungen)

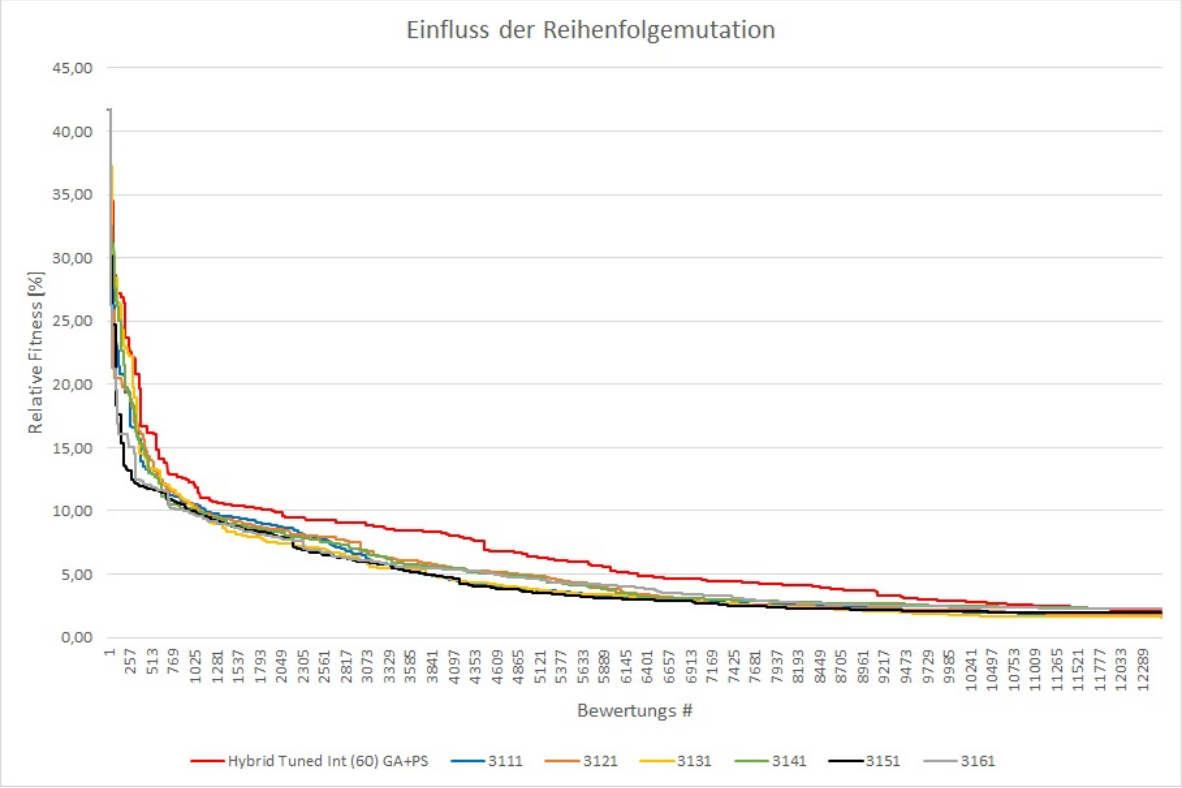

Abbildung 3.7: Einfluss der Reihenfolgemutation (Mittelwert aus zwei Wiederholungen)

Einfluss der Wertmutationsmethode Abbildung 3.8 zeigt den Einfluss von normal- oder gleichverteilter Mutation auf die Performance. Neben den beiden GA-315-Varianten sind zum Vergleich auch beide GA-311-Varianten abgebildet. Die Algorithmen mit normalverteilter Mutation liefern vor allem im ersten Drittel des Optimierungsvorgangs die besseren Ergebnisse. Aufgrund der besseren Endergebnisse wird der GA-3112 für alle weiteren Betrachtungen ausgewählt.

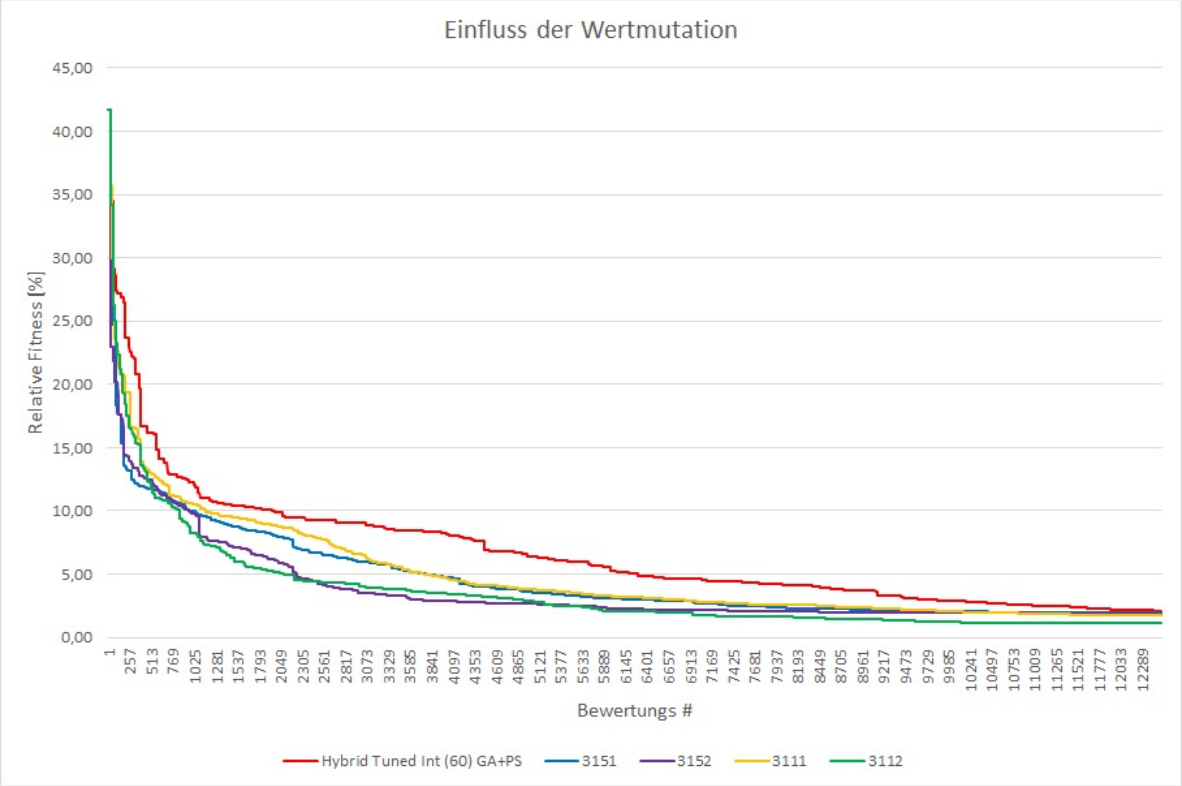

Abbildung 3.8: Einfluss der Wertmutation (Mittelwert aus zwei Wiederholungen)

Signifikanztests Um zu untersuchen, ob die Verbesserungen gegenüber der Ausgangssituation (GA-1111) statistisch signifikant sind, werden die durchschnittlichen Fitnesswerte in mehreren Punkten verglichen und ein Zweistichproben-t-Test durchgeführt. Die Null- und Alternativhypothesen lauten in diesem Fall:

$$1.\, H_0 :\ \mu_{1111} - \mu_{3112} \leq 0$$

$$2.\, H_1 :\ \mu_{1111} - \mu_{3112} > 0$$

Wobei μ die arithmetischen Mittelwerte an den Messpunkten sind. Die Teststatistik errechnet sich unter Annahme einer Normalverteilung und gleicher Varianzen zu (t ist

t-verteilt mit $m + n - 2$ Freiheitsgraden):

$$t = \sqrt{\frac{nm}{n+m}} \frac{\bar{X}_{1111} - \bar{X}_{3112}}{S} \tag{3.4}$$

Dabei sind n und m die Stichprobengrößen. Für die Signifikanztests wurde $n = m = 10$ verwendet (statt zwei Wiederholungen wie bei den vorherigen Experimenten). $\bar{X}_{1111}$ und $\bar{X}_{3112}$ sind die Mittelwerte der Stichproben an den Messpunkten und S die gewichtete empirische Standardabweichung. Diese berechnet sich zu:

$$S^2 = \frac{(n-1)S_{1111}^2 + (m-1)S_{3112}^2}{n+m-2} \tag{3.5}$$

Der zugehörige p-Wert (Signifikanzwert) gibt die Wahrscheinlichkeit an, unter H_0 den berechneten Wert der Teststatistik oder einen extremeren Wert in Richtung der Alternative zu erhalten. Er entspricht der Fläche unter der Dichtefunktion der t-Verteilung mit den angegeben Freiheitsgraden für den berechneten t-Wert (siehe Abbildung 3.9).

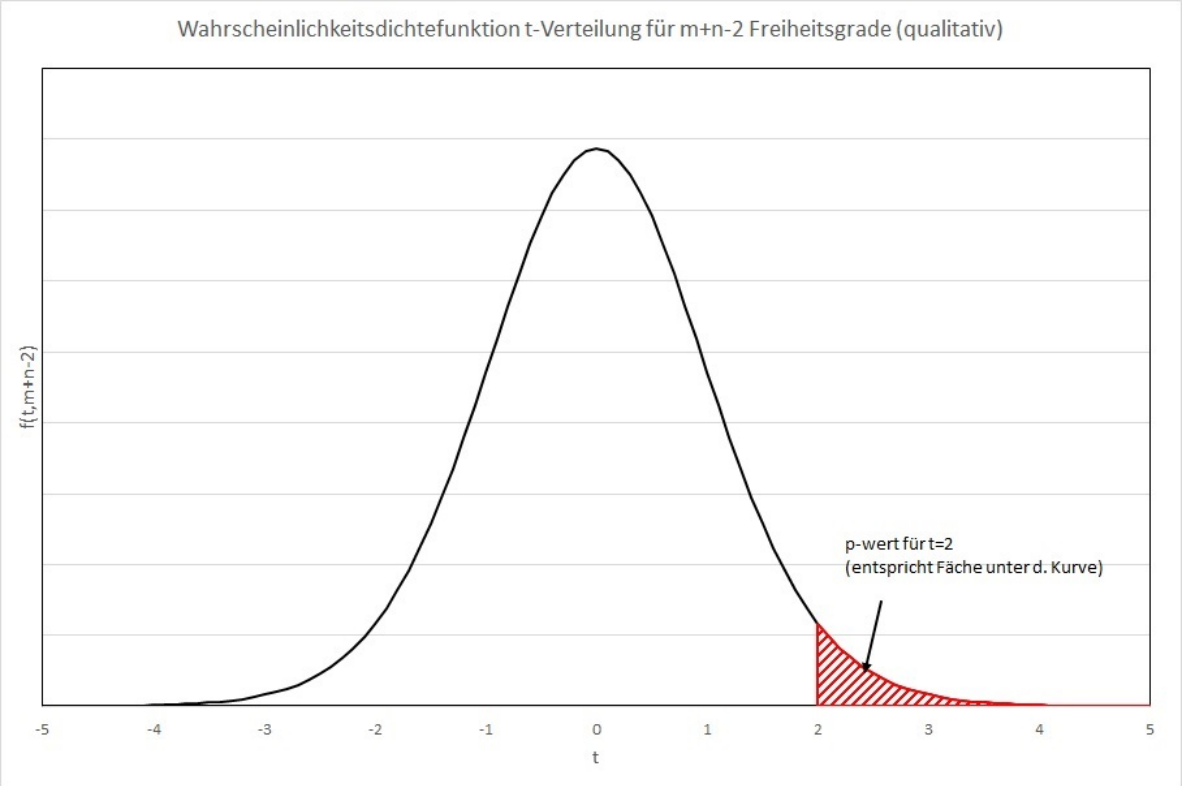

Abbildung 3.9: Veranschaulichung des Signifikanzwerts

Unterschreitet der p-Wert ein definiertes Signifikanzniveau ist die Nullhypothese zugunsten der Alternativhypothese abzulehnen. Das Signifikanzniveau ist die maximal zulässige Irrtumswahrscheinlichkeit (falsche Ablehnung der Nullhypothese).

Die Ergebnisse sind in Tabelle 3.3 dargestellt.

Bewertungs #	**Rel. Fitnesswert GA 1111**	**Rel. Fitnesswert GA 3112**	**Differenz (p-Wert)**
1000			
Mittelwert	9.24	8.51	0.73***
(Varianz)	(0.13)	(0.23)	(0.01)
2000			
Mittelwert	7.48	5.36	2.12***
(Varianz)	(0.71)	(0.55)	(<0.01)
5000			
Mittelwert	4.36	3.12	1.24***
(Varianz)	(0.43)	(0.19)	(<0.01)
12525 (Endergebnis)			
Mittelwert	2.56	1.56	1.00***
(Varianz)	(0.17)	(0.36)	(<0.01)

Tabelle 3.3: Ergebnisse der Signifikanztests. *** entspricht einem Signifikanzniveau von 1%.

Dar GA-3112 liefert an allen Messpunkten ein signifikant besseres Ergebnis als der GA-1111. Es wird daher angenommen, dass der GA-3112 leistungsfähiger hinsichtlich Optimierungsgeschwindigkeit und Qualität der Ergebnisse ist.

Optimierung der Teil-Zielfunktionen Der Verlauf der Teil-Fitnesswerte in Abbildung 3.10 zeigt eine schnelle Reduktion des Werts für verzögerte Lieferung (0 nach ca. 100 Bewertungen). Die Lagerkosten erreichen nach ca. 700 Bewertungen ein Niveau von 67% und können anschließend nur noch geringfügig verbessert werden. Die Energiekosten werden nach kurzen Schwankungen zu Beginn der Optimierung kontinuierlich verringert. Die Verbesserungen (bzw. Verschlechterungen) bei der Gesamtdurchlaufzeit tragen, bedingt durch die vergleichsweise geringe Gewichtung, nur wenig zum Gesamtfitnesswert bei (ca. 10%). Dementsprechend gering ist die Auswirkung der Verbesserungen (bzw. Verschlechterungen).

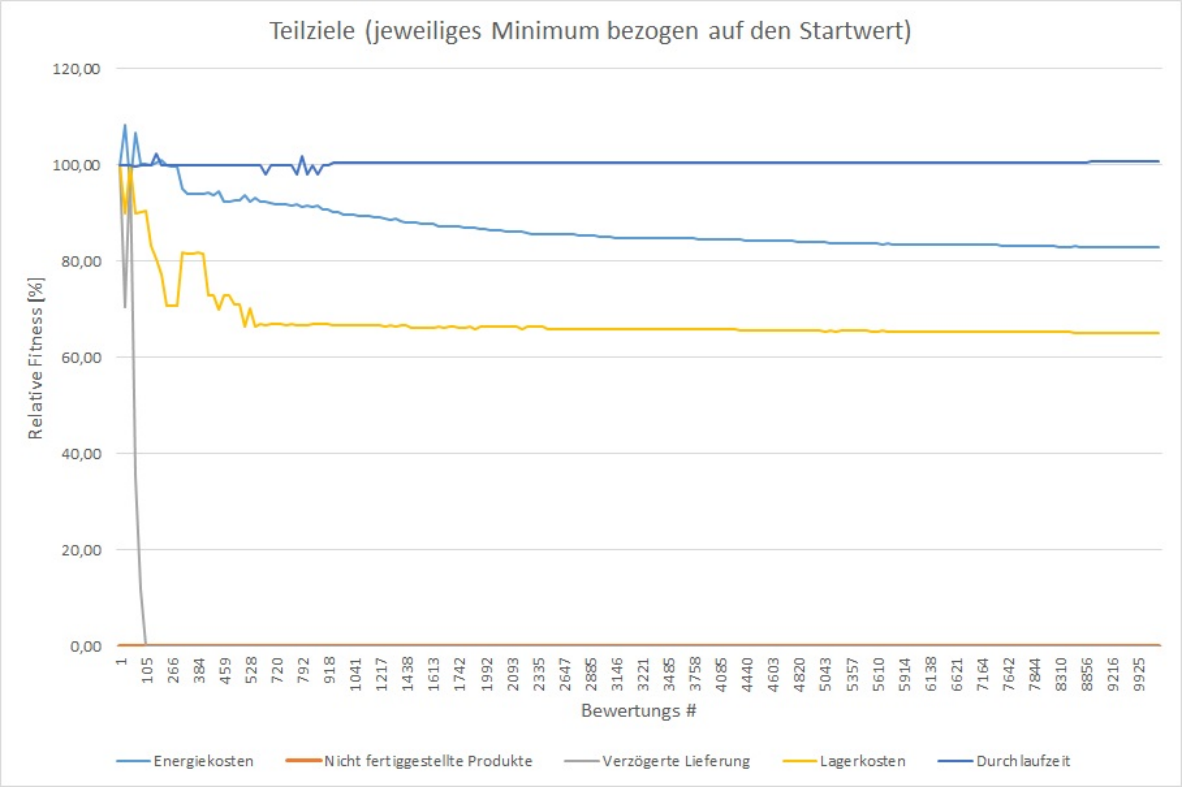

Abbildung 3.10: Teilziele GA-3112

Anteil an ungültigen (bzw. schlechten) Lösungen während der Optimierung Abbildung 3.11 zeigt den mittleren Anteil ungültiger Lösungen (bzw. den Anteil an Lösungen mit nicht fertiggestellten Produkten) pro Generation. Der Anteil steigt bis zur 100. Generation beinahe linear auf ca. 22% an und bleibt anschließend auf diesem Niveau. Ziel der weiteren Verbesserungsmaßnahmen ist zunächst die Reduktion dieses Anteils durch die Definition weiterer Nebenbedingungen und anderen Maßnahmen.

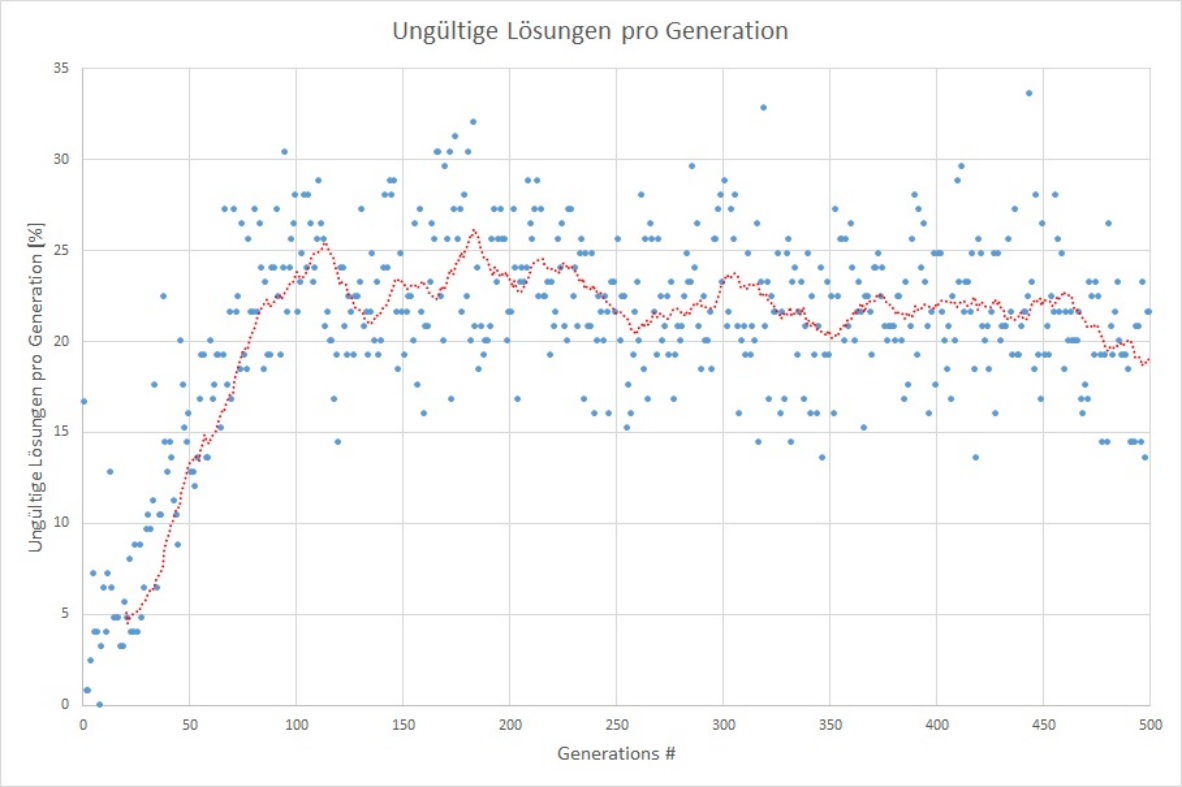

Abbildung 3.11: Anteil ungültiger Lösungen während des Optimierungsvorgangs GA-3112

3.5 Einschränkung des Lösungsraums und Vermeidung von ungültigen Lösungen

Da beinahe ein Viertel aller vom Mutationsoperator erzeugten Lösungen ungültig sind (siehe Abbilung 3.11), werden in diesem Abschnitt Maßnahmen zur Reduktion dieses Anteils getestet. Dazu werden weitere Nebenbedingungen definiert und Anpassungen an der Mutationsfunktion durchgeführt. Zur weiteren Einschränkung des Lösungsraums wird ein Speicher mit bereits bewerteten Lösungen eingeführt. Außerdem werden die Variablen des Lösungsvektors auf ganze Zahlen beschränkt.

3.5.1 Diskretisierung des Lösungsvektors

Eine Beschränkung der Lösungsvektoreinträge auf ganze Zahlen bzw. natürliche Zahlen schränkt den Lösungsraum auf eine endliche Menge an Alternativen ein. Bisher waren die Lösungsvektoreinträge in Stunden angegeben (siehe Abbildung 3.3). Eine Beschränkung auf ganze Stunden ist jedoch nicht sinnvoll, da die Lösung ansonsten zu ungenau

wird. Aus diesem Grund wurden 60s und 30s Intervalle getestet. Die Ergebnisse sind in Abbildung 3.12 dargestellt.

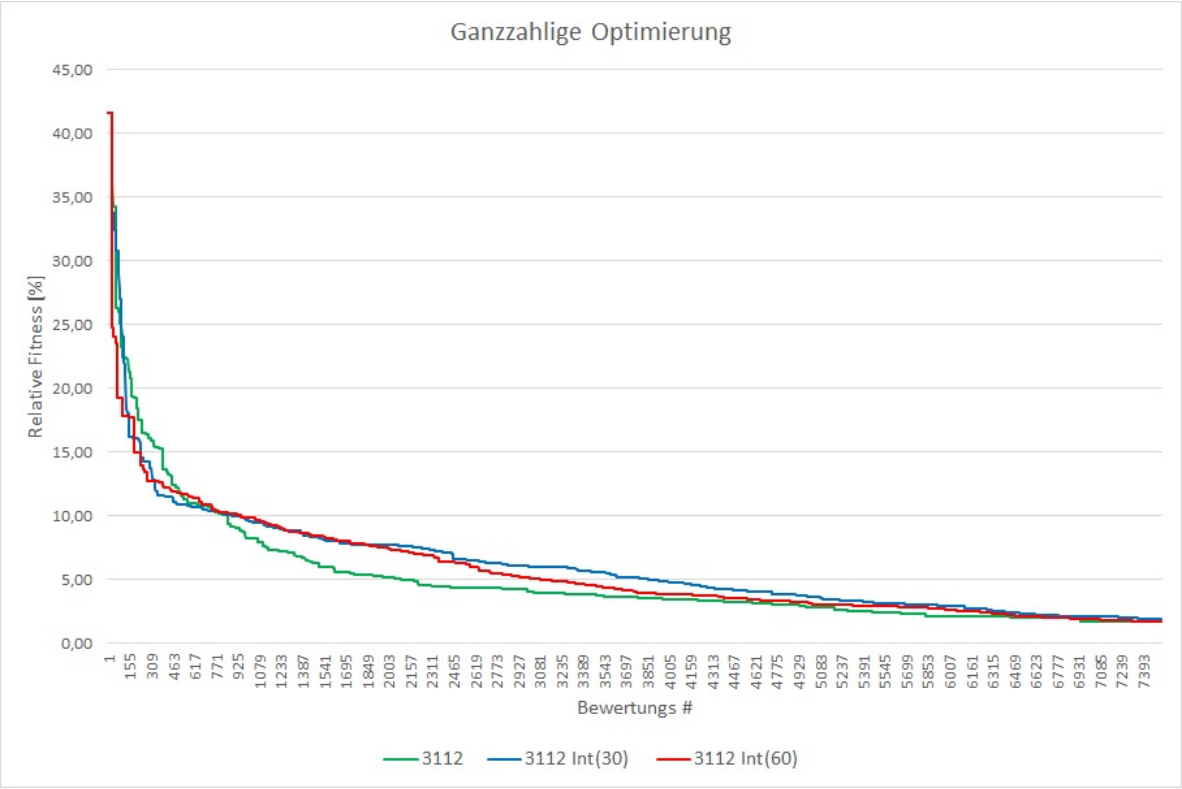

Abbildung 3.12: Relative Fitness GA-3112 Integer

Beide GA-3112 Int reduzieren den Zielfunktionswert zu Beginn der Optimierung (bis ca. 500 Bewertungen) schneller als der GA-3112. Danach ist aber die Performance des GA-3112 besser. Alle Verfahren liefern gleich gute Endergebnisse. Der Unterschied zwischen GA-3112 Int(60) und GA-3112 Int(30) fällt nur gering aus, wobei der GA-3112 Int(60) etwas bessere Ergebnisse liefert.

3.5.2 Integration eines Speichers für bereits bewertete Lösungen

Bisher wurde die Simulation nur zur Bewertung neuer Lösungen ausgeführt. Eine erneute Bewertung wurde durch die Verwendung eines entsprechenden Speichers bereits verhindert, die wiederholte Erzeugung gleicher Lösungen durch den Algorithmus jedoch nicht. Die Diskretisierung des Lösungsvektors führt dazu, dass durchschnittlich 10.5% der Lösungen mindestens zweimal erzeugt werden (beim GA-3112 Int(60)). Um dies zu verhindern, wird angelehnt an die Tabu-Liste von Tabu-Search-Algorithmen, ein Speicher mit bereits bewerteten Lösungen im Mutationsoperator eingeführt. Bevor eine

erzeugte Lösung in die nächste Population aufgenommen wird, wird erst überprüft, ob diese neu ist. Durch diese Maßnahme werden die Performance-Nachteile des GA-3112 Int(60) gegenüber dem GA-3112 ausgeglichen (siehe Abbildung 3.13).

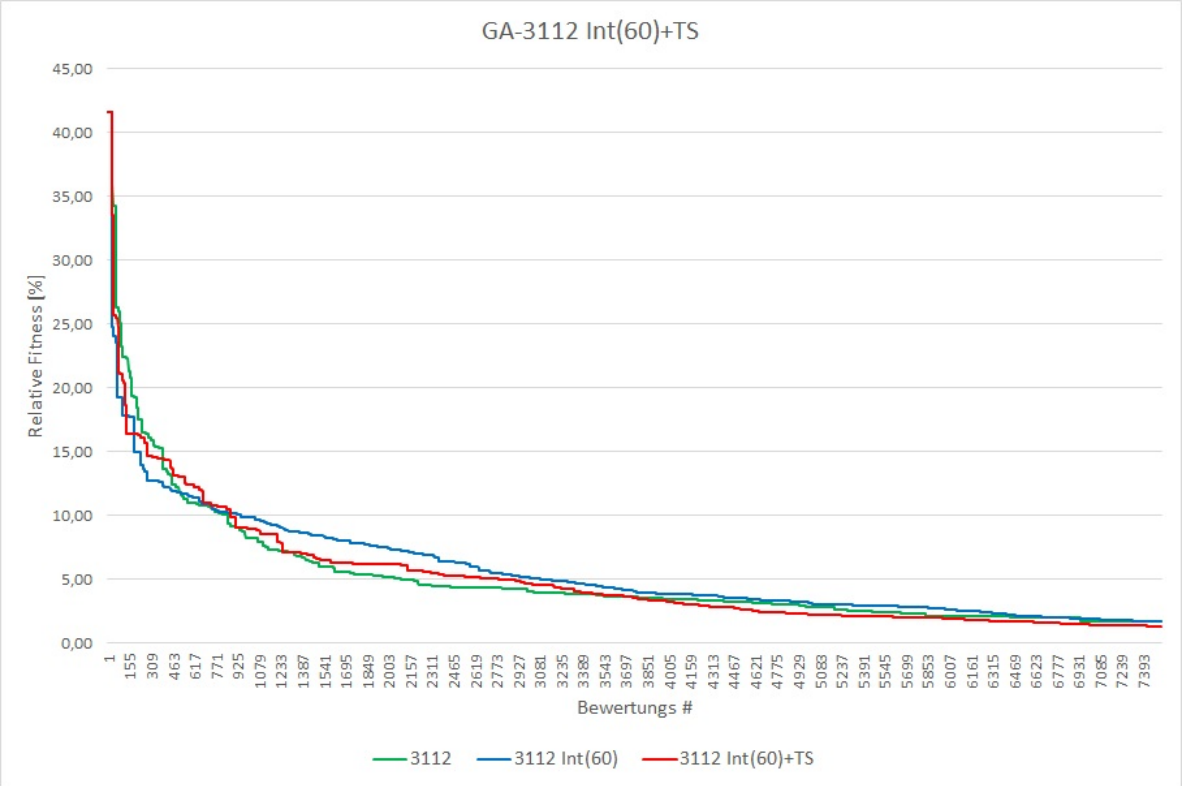

Abbildung 3.13: Relative Fitness GA-3112 Int+TS

3.5.3 Weitere Nebenbedingungen

Der Anteil an ungültigen Lösungen, die während der Optimierung erzeugt werden, hat sich im Vergleich zum GA-3112 nicht reduziert (siehe Abbildung 3.14). Um dies zu erreichen werden folgende neue Nebenbedingungen getestet:

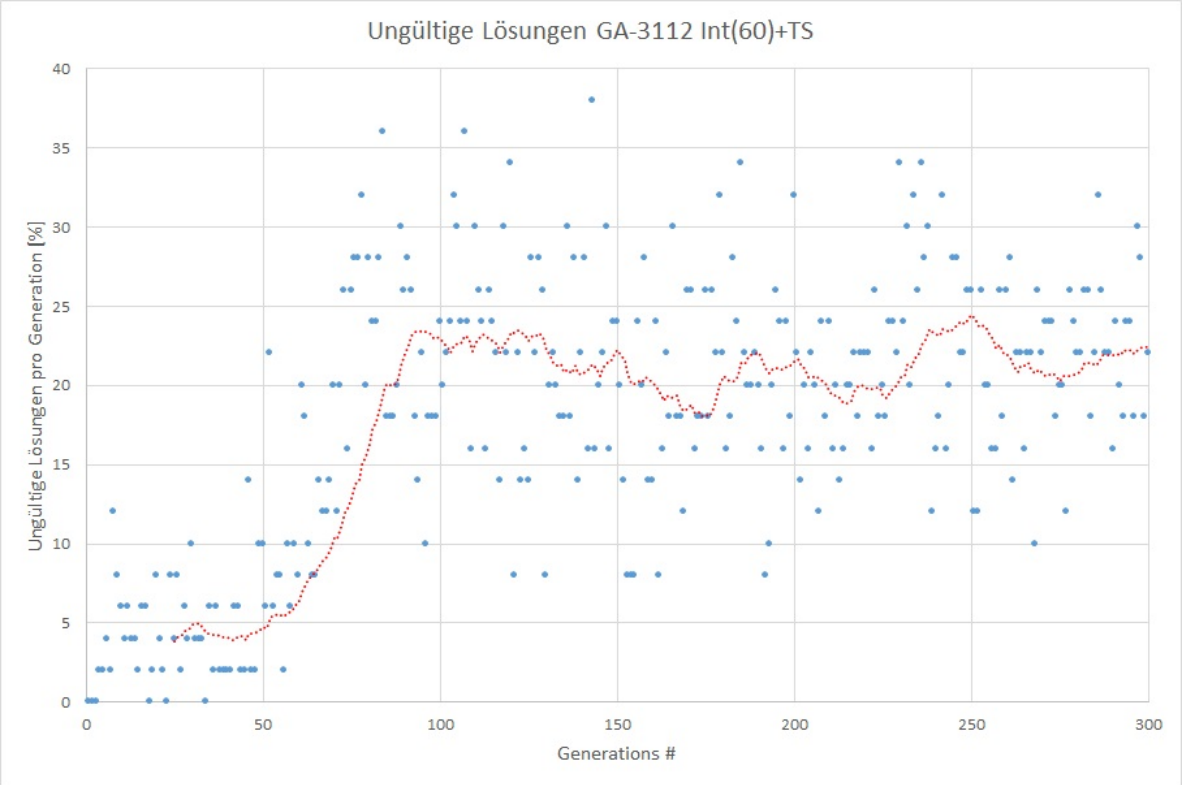

Abbildung 3.14: Ungültige Lösungen pro Generation GA-3112 Int(60)+TS

- Verhinderung von Überschneidungen der Betriebszeiten von Lager und Merger. Dazu wird nach der Erzeugung einer Lösung überprüft, ob Überschneidungen vorliegen und diese gegebenenfalls repariert. Dies wird erreicht, indem die Startzeitpunkte nachfolgender Lose nach hinten verschoben werden, bis die Überschneidungen beseitigt sind. Abbildung 3.15 zeigt, dass diese Maßnahme die Leistungsfähigkeit des Algorithmus kaum beeinflusst, aber den durchschnittlichen Anteil an ungültigen Lösungen weiter erhöht. Dies lässt sich einerseits dadurch erklären, dass die Simulation Lösungen mit Überschneidungen akzeptiert (und nicht als ungültig bewertet) und andererseits dadurch, dass die Lösungsvektoren durch eine Reparatur stärker verändert werden, als durch die ursprünglichen Operationen (Vertauschung von Losreihenfolgen und Änderung von Startzeiten im Lager). Durch die größeren Veränderungen ist es wahrscheinlicher, dass die Betriebszeiten mit den entsprechenden Betriebszeiten der restlichen Aggregate nicht mehr zusammenpassen (der Ofen startet beispielsweise zu spät). Die Nebenbedingung wird dennoch beibehalten, da sie gewährleistet, dass nur sinnvolle Produktionspläne erzeugt werden.

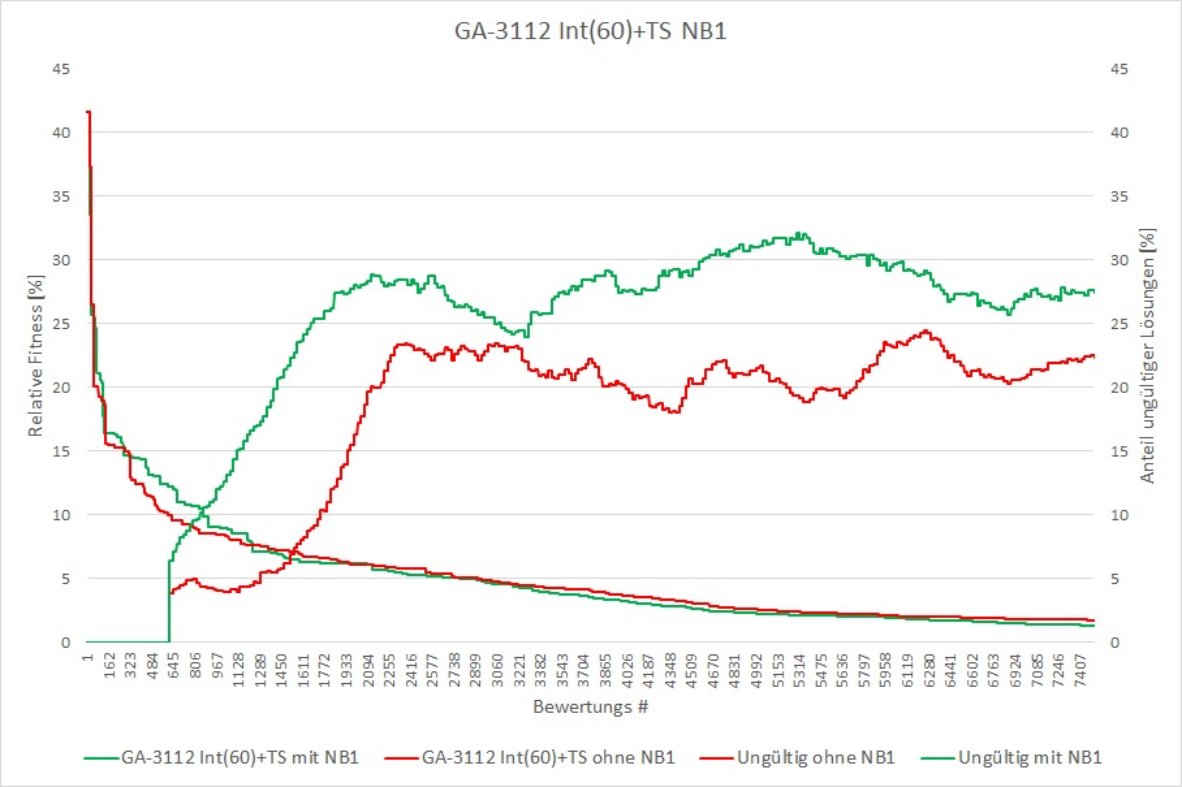

Abbildung 3.15: Ungültige Lösungen mit und ohne Nebenbedingung 1 (gleitender Durchschnitt über 25 Generationen)

- Eine nähere Untersuchung der ungültigen Lösungen zeigt, dass diese hauptsächlich durch die Vertauschung der Losreihenfolgen entstehen (siehe Abbildung 3.16). Durch die Vertauschung von Produktionslosen unterschiedlicher Größe sind die Betriebszeiten der Aggregate oft nicht mehr aufeinander abgestimmt. Bei der Verschiebung größerer Lose nach hinten kann es beispielsweise vorkommen, das die restliche Betriebszeit des Ofens (bzw. Frosters) nicht mehr ausreicht, um alle Produkte fertig zu produzieren. Dies macht sich vor allem im fortgeschrittenen Optimierungsvorgang bemerkbar, da dort die Betriebszeiten der Aggregate bereits angepasst (d.h. verkürzt) sind. Es erscheint daher sinnvoll, die Vertauschung von Losen mit einer Anpassung der Betriebszeiten von Ofen und Froster zu verbinden. Dazu werden die jeweiligen Betriebsintervalle (von Ofen und Froster) bei einer Vertauschung der Losreihenfolge ebenfalls getauscht. Werden z.B. das dritte und vierte Los vertauscht, werden das dritte und vierte Betriebsintervall des Ofens ebenfalls angepasst, indem die (Ofen-) Starts beibehalten und die Ausschaltzeitpunkte verschoben werden. Die Gesamtbetriebszeit bleibt dadurch gleich. Durch diese Anpassungen können aber Überschneidungen von Betriebszeiten auftreten, was eine Reparatur der Lösung erforderlich macht. Dies wird erreicht, indem

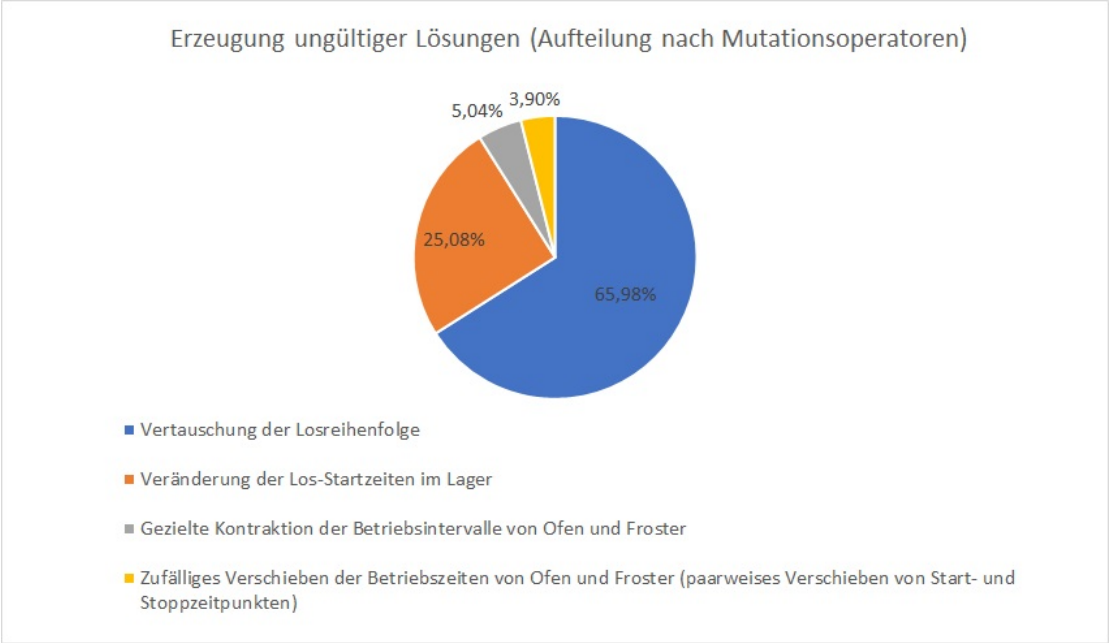

Abbildung 3.16: Erzeugung ungültiger Lösungen

betroffene Intervalle nach hinten verschoben werden, bis die Überschneidungen beseitigt sind. Anschließend muss überprüft werden, ob die obere Zeitbeschränkung des letzten Intervalls eingehalten wird. Falls nicht, werden die Zeiten, in denen das Aggregat nicht läuft, reduziert bis die Schranke eingehalten wird.

Abbildung 3.17 zeigt die Ergebnisse der Tests dieser Anpassungen. Das Ziel den Anteil ungültiger Lösungen zu reduzieren wurde nicht erreicht. Die Performance des Algorithmus ist aber zu Beginn der Optimierung (bis 1.000 Bewertungen) besser als nur mit Nebenbedingung 1, das Endergebnis ist aber schlechter.

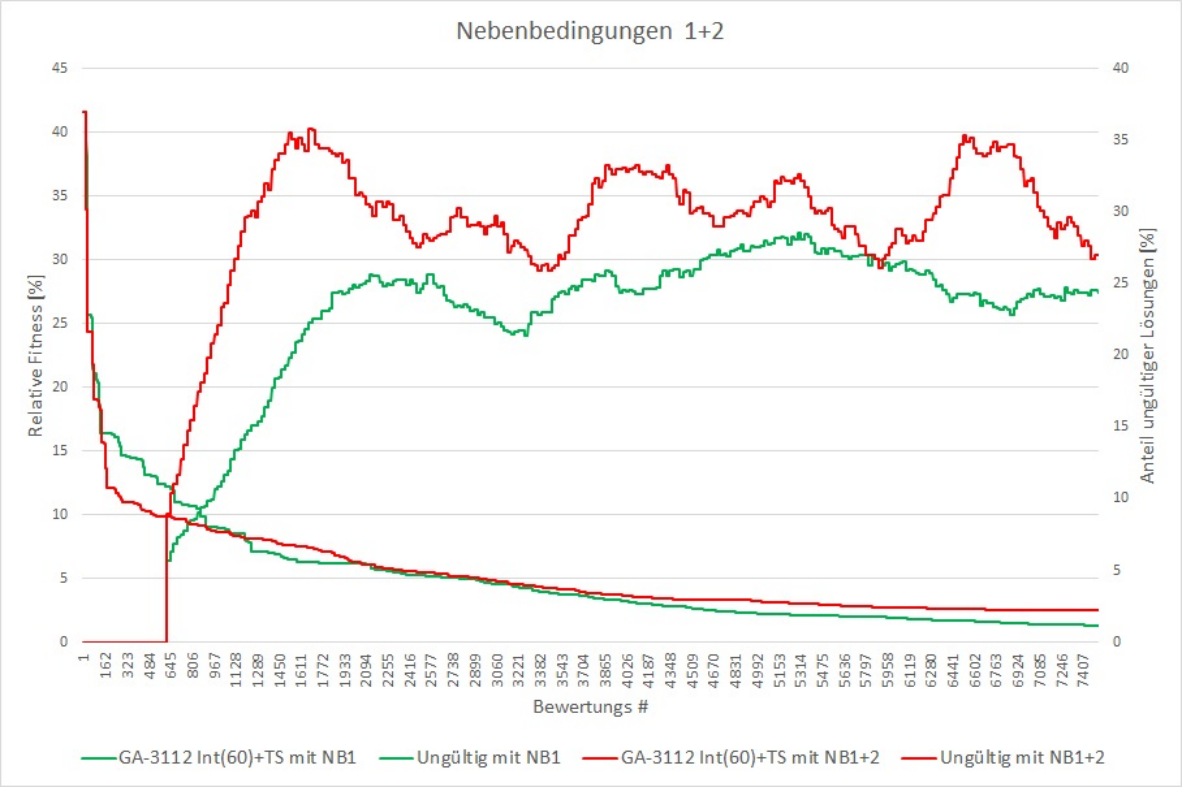

Abbildung 3.17: Ungültige Lösungen mit Nebenbedingungen (gleitender Durchschnitt über 25 Generationen)

Der hohe Anteil an ungültigen Lösungen ist darauf zurückzuführen, dass andere wichtige Informationen wie z.B. der Aufheizzustand oder der Pufferstand vor dem Ofen nicht beim Tausch berücksichtigt werden. Die Umverteilung der Betriebszeiten führt aber dazu, dass am Beginn der Optimierung schnell gute Ergebnisse erzielt werden. Im späteren Verlauf sind jedoch die Betriebszeiten der Aggregate bereits relativ genau an die Größe der Produktionslose angepasst. Die Durchführung eines Tausches ohne Berücksichtigung aller Randbedingungen wie dem Aufheizzustand etc. führt dann mit größerer Wahrscheinlichkeit zu einer ungültigen Lösung. Die Änderungen werden daher verworfen.

3.6 Anpassung verschiedener Algorithmen-Parameter

In diesem Abschnitt werden die Einflüsse einiger Parameter wie z.B. der verwendeten Populationsgröße auf die Leistungsfähigkeit des Algorithmus untersucht. Alle Untersuchungen werden mit dem GA-3112 Int(60)+TS mit Nebenbedingung 1 durchgeführt, alle anderen Einstellungen bleiben unverändert.

3.6.1 Populationsgröße

Genetische Algorithmen arbeiten mit einer Menge von Lösungen (einer Population), aus der durch verschiedene Operationen neue Lösungen erzeugt werden. In der hier verwendeten Variante des GA bleibt die Anzahl an Lösungen pro Generation konstant und wird als Populationsgröße P bezeichnet. Bisher wurde eine Populationsgröße von $P = 25$ verwendet, nun soll untersucht werden, wie sich andere Parameterwerte auf den Algorithmus auswirken. Um die Ergebnisse vergleichbar zu machen, wird die Gesamtanzahl an Bewertungen konstant gehalten (7500 Bewertungen). Die Bewertungsanzahl B ergibt sich aus $B = G * P$, wobei G Anzahl an Generationen ist, daher muss diese für verschiedene Werte von P entsprechen angepasst werden.

In Abbildung 3.18 ist der Verlauf der relativen Fitnesswerte für mehrere P-Größen dargestellt. Eine Verringerung von P erhöht die Geschwindigkeit des Algorithmus. Ab $P = 10$ nimmt aber die Varianz in den Ergebnisse stark zu und der Algorithmus konvergiert teilweise gegen schlechtere lokalen Optima (siehe Abbildung 3.19). Aus diesem Grund wird für alle weiteren Tests eine Populationsgröße von $P = 15$ verwendet. Das erhöht die Geschwindigkeit im Vergleich zu vorher und führt zu keinen schlechteren Ergebnissen.

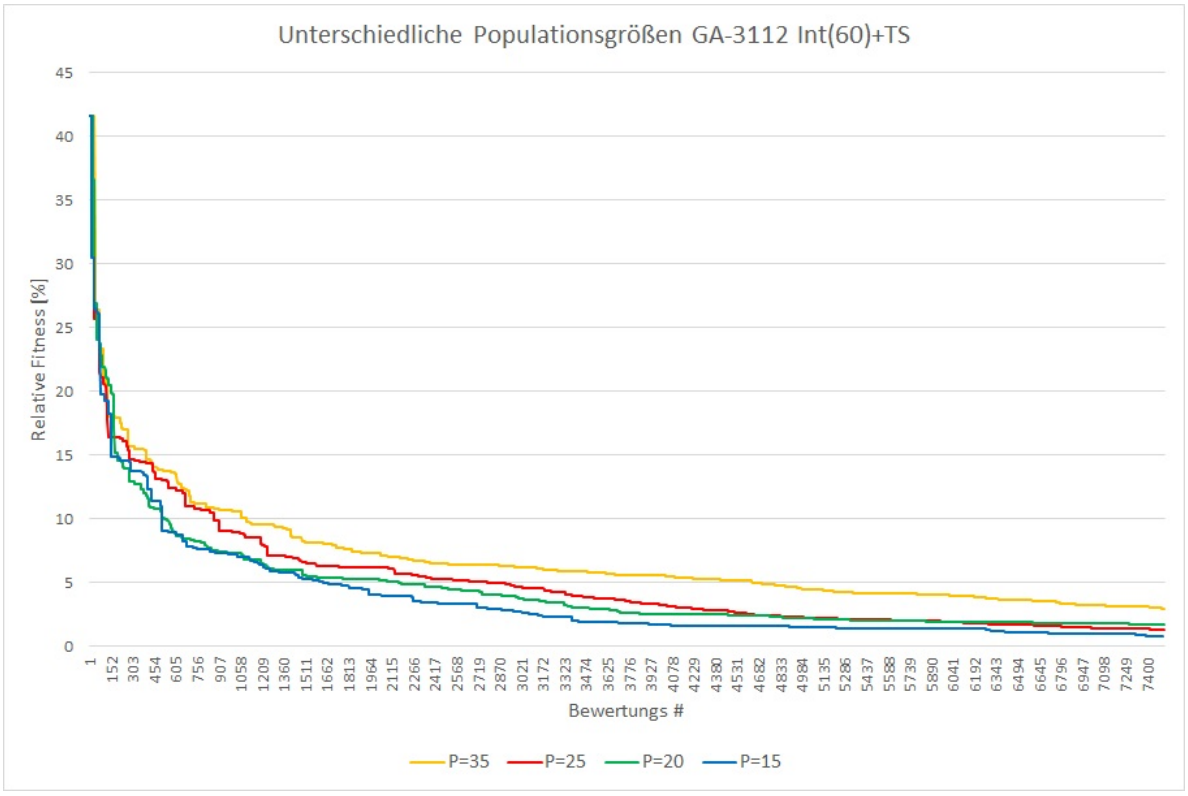

Abbildung 3.18: Relative Fitness für unterschiedliche Populationsgrößen

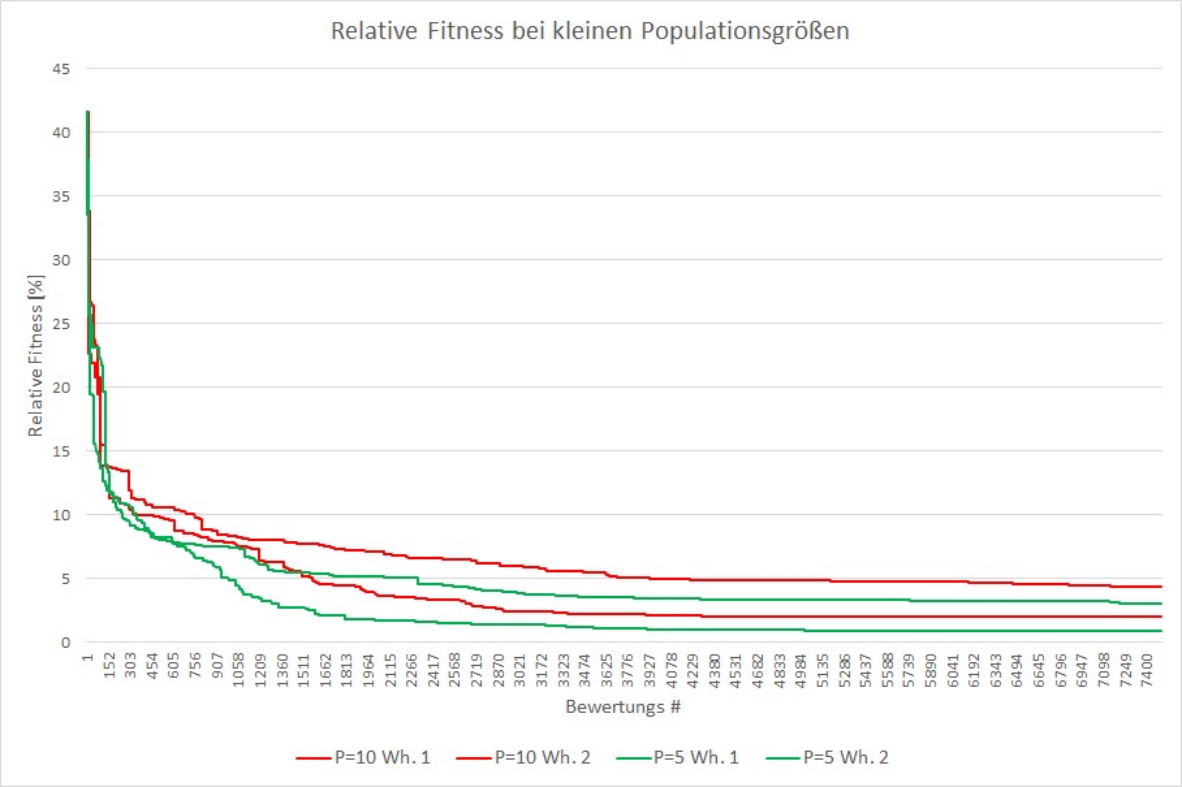

Abbildung 3.19: Relative Fitness bei kleinen Populationsgrößen

3.6.2 Anzahl der veränderten Lösungsvektoreinträge

Die Anzahl an veränderten Lösungsvektoreinträgen durch die Mutation der ersten beiden Teillösungsvektoren (Lager und Merger) ist eine gleichverteilte Zufallszahl. So ist die Wahrscheinlichkeit, dass nur ein Eintrag oder alle Einträge verändert werden, gleich groß [5]. Dies führt dazu, dass der Lösungsvektor oftmals sehr stark verändert wird, was im fortgeschrittenem Optimierungsvorgang (für die Feinabstimmung) von Nachteil sein kann. Im Folgenden wird daher getestet, wie sich kleinere Veränderungen nach 2.500 Bewertungen auswirken. Dazu wird anstatt der Displacement Mutation nach 2.500 Bewertungen eine Exchange Mutation (Beschreibung in Kapitel 3.4.1) für die Reihenfolgemutation verwendet, was dazu führt, dass immer nur zwei Vektoreinträge verändert werden. Außerdem wird bei dem Mutationsoperator für die Startzeiten der Lose im Lager nach 2.500 Bewertungen nur noch ein Vektoreintrag verändert. Die Ergebnisse der Tests dieser Maßnahmen sind in Abbildung 3.20 dargestellt.

[5] Dies betrifft die Operationen Vertauschung von Losreihenfolgen und Änderung von Los-Startzeiten im Lager. Die Kontraktion von Aggregat-Betriebszeiten sowie deren Verschiebung erfolgen immer nur einzeln.

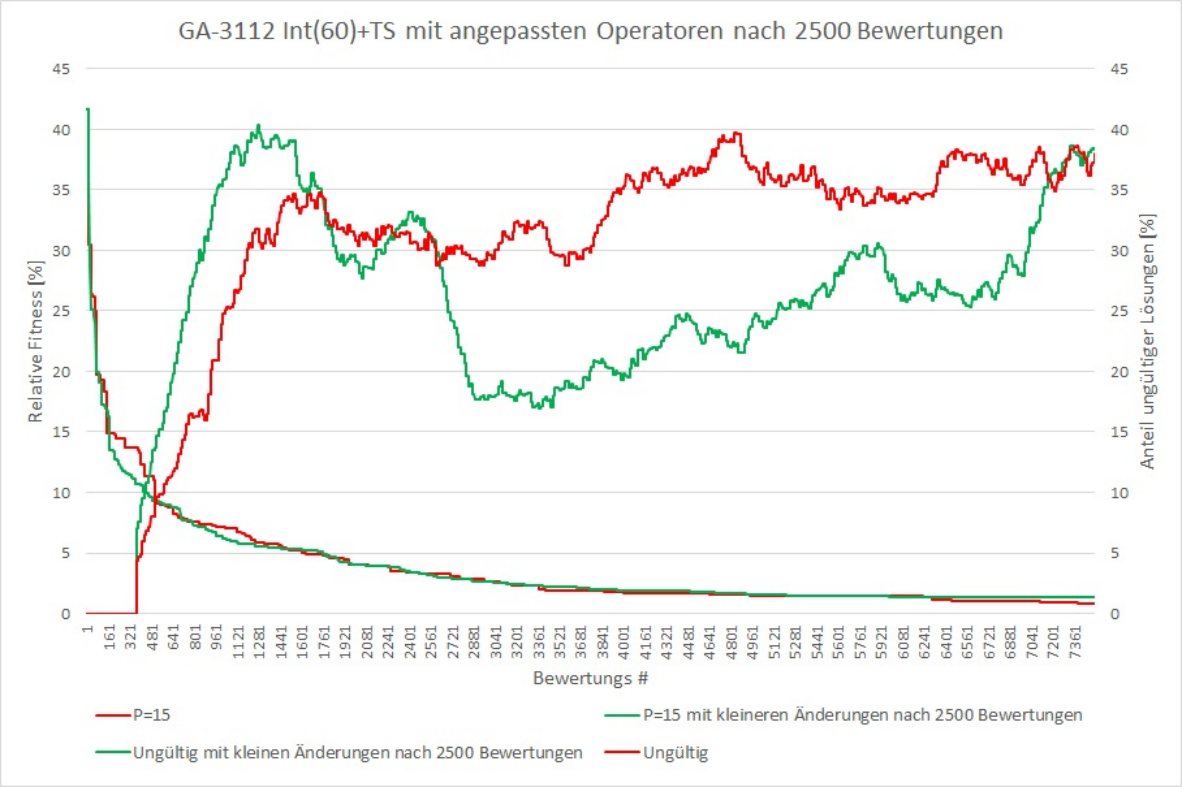

Abbildung 3.20: Angepasster Mutationsoperator nach 2.500 Bewertungen

Kleinere Änderungen im fortgeschrittenem Optimierungsvorgang haben in diesem Testszenario keinen Einfluss auf den Verlauf der relativen Fitnesswerte. Die geringen Unterschiede werden auf statische Schwankungen zurückgeführt. Ein deutlicher Unterschied ist aber beim Anteil an ungültigen Lösungen pro Generation zu erkennen. Dieser reduziert sich nach der Anpassung der Operatoren, erreicht aber nach ca. 7.200 Bewertungen wieder das vorherige Niveau.

Diese Anpassungen werden trotz des geringen Effekts übernommen, da erwartet wird, dass sich positive Auswirkungen erst bei komplexeren Testszenarien bemerkbar machen.

3.6.3 Unterschiedliche Wahrscheinlichkeitsdichtefunktionen für die Mutation

Es wurde bereits in Kapitel 3.4.2 gezeigt, dass die Verwendung einer normalverteilten Zufallszahl mit Erwartungswert 0 und einer definierten Standardabweichung bei der Mutation von Lösungsvektoreinträgen Vorteile gegenüber einer gleichverteilten Zufallszahl bietet. Dies liegt daran, dass bei einer Normalverteilung hauptsächlich kleinere Änderungen realisiert werden aber auch größere Änderungen möglich sind. Dadurch kann

der Algorithmus sowohl grobe Anpassungen zu Beginn als auch feine Abstimmungen am Ende der Optimierung durchführen. Zusätzlich zu einer normalverteilten Änderung von Vektoreinträgen mit Mittelwert $\mu = 0$ und Standardabweichung $\sigma = 25\%$ wurde bisher eine Verringerung von σ mit ansteigender Generationszahl gemäß der Formel:

$$\sigma_G = \sigma_{G-1} * \left(1 - \frac{G-1}{G_{max}}\right) \tag{3.6}$$

σ_G : Standardabweichung für die aktuelle Generation
σ_{G-1} : Standardabweichung der letzten Generation
G : Generationsnummer
G_{max} : Maximale Generationsnummer

durchgeführt. Dies sollte die Fähigkeit zur Feinabstimmung am Ende des Optimierungsvorgangs weiter verbessern. Abbildung 3.21 zeigt jedoch, dass dadurch keine Verbesserungen erzielt werden konnten. Ab ca. 4200 Bewertungen kann der Algorithmus, bedingt durch das kleine σ den Zielfunktionswert oft nicht weiter verbessern. Bei konstant gehaltenem σ besteht das Problem hingegen nicht. Aus diesem Grund wird bei allen weiteren Tests eine konstante Standardabweichung verwendet.

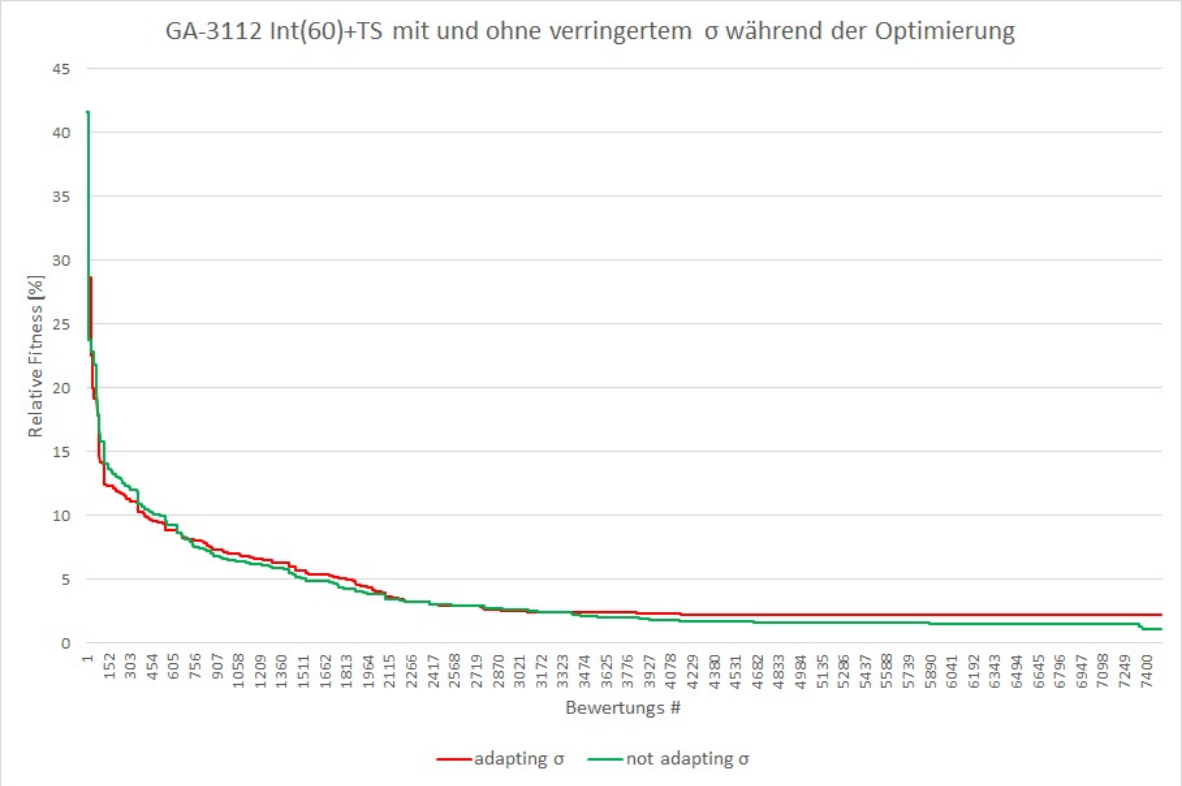

Abbildung 3.21: GA-3112 Int(60)+TS mit konstanter und variabler Standardabweichung

Bevor die Auswirkungen einer Variation von σ auf die restlichen Operatoren der Mutationsfunktion untersucht werden, wird getestet wie sich eine Veränderung bei der Kontraktion der Betriebsintervalle von Ofen und Froster auswirkt.

Eine Reduktion von $\sigma = 25\%$ auf $\sigma = 15\%$ führt dazu, dass der Algorithmus in der ersten Hälfte des Optimierungsvorgangs (bis ca. 3.600 Bewertungen) langsamer wird (siehe Abbildung 3.22). Eine Erhöhung auf $\sigma = 35$ macht den Algorithmus jedoch nur unwesentlich schneller. Das Endergebnis bleibt bei allen Varianten nahezu gleich, weshalb die Standardabweichung für die Kontraktion der Betriebsintervalle bei $\sigma = 25\%$ belassen wird.

Abbildung 3.23 zeigt die Ergebnisse für verschiedene σ bei den restlichen Operatoren der Mutationsfunktion (Änderung der Startzeitpunkte im Lager und verschieben der Betriebsintervalle). Der Einfluss auf die Geschwindigkeit ist hier nur klein, wobei mit größeren Parameterwerten bessere Ergebnisse erzielt wurden. Bei allen weiteren Tests wird daher eine Standardabweichung von $\sigma = 35$ für die Verschiebung von Betriebszeiten und die Änderung von Startzeiten im Lager verwendet.

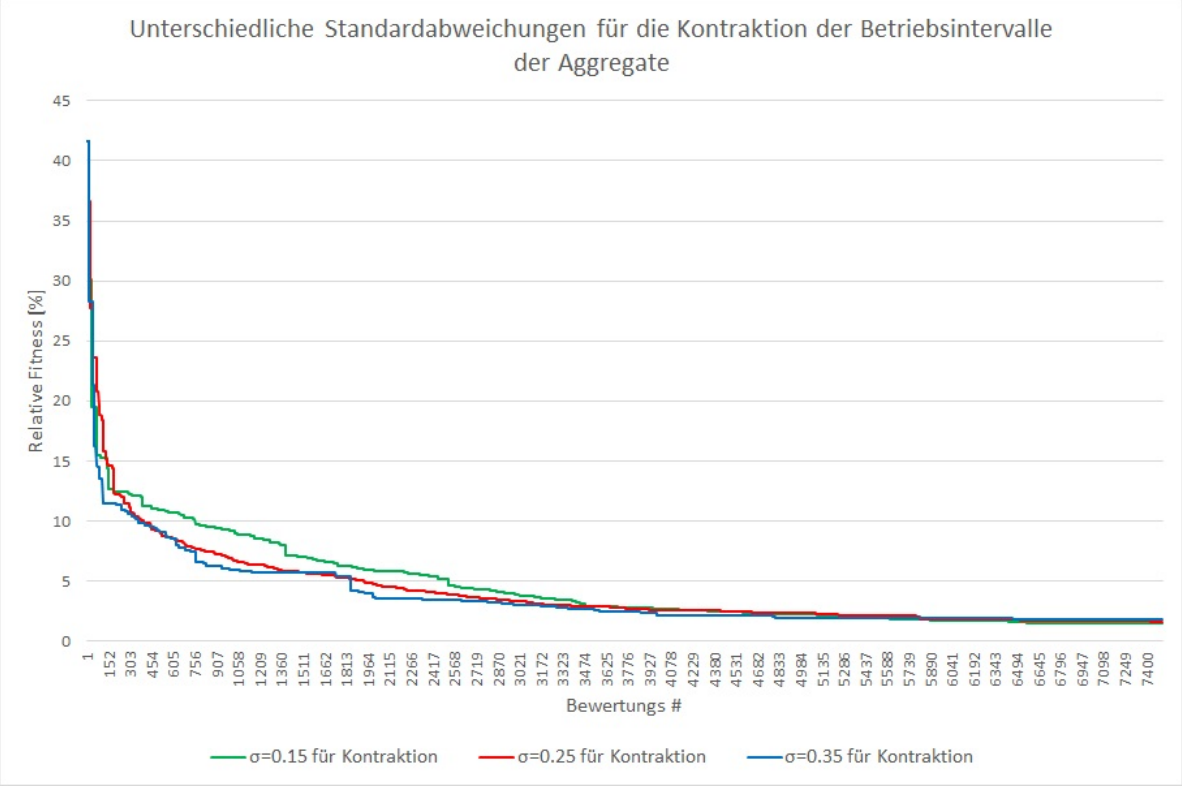

Abbildung 3.22: Unterschiedliche Standardabweichungen für die Kontraktion der Betriebsintervalle der Aggregate

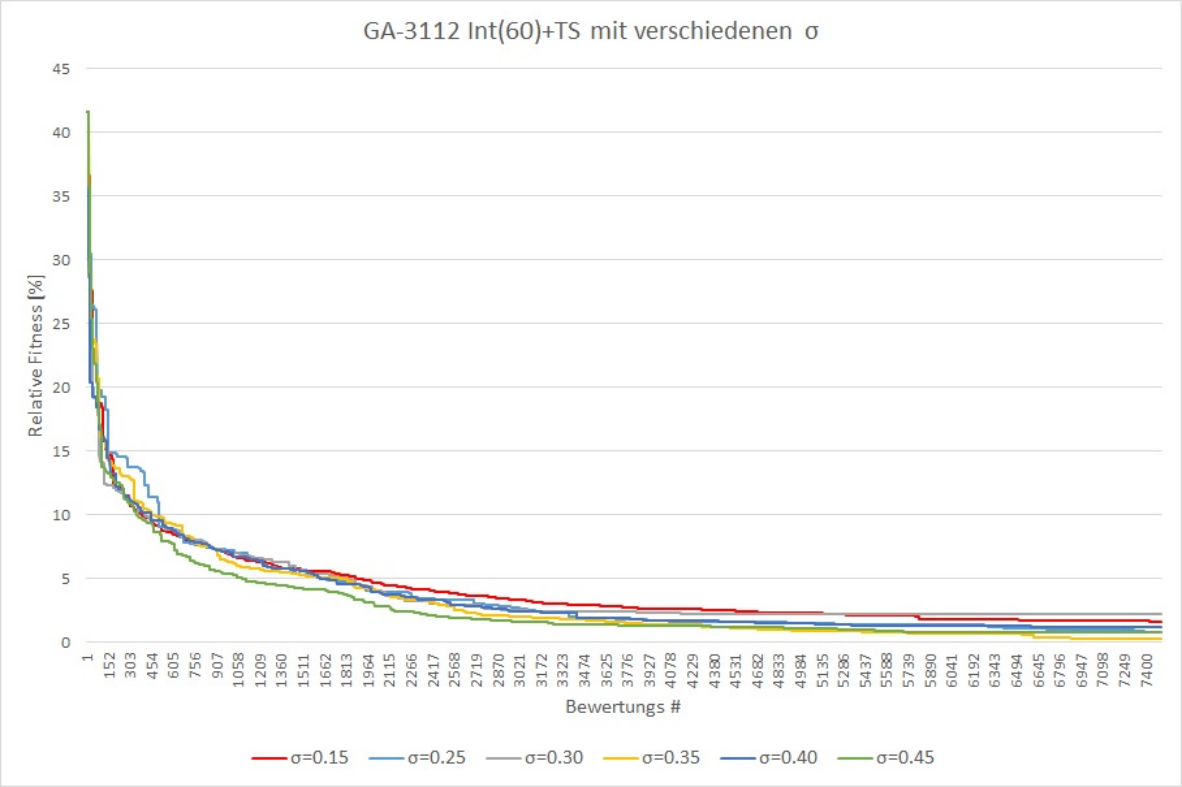

Abbildung 3.23: GA-3112 Int(60)+TS mit verschiedenen Standardabweichungen

3.6.4 Signifikanztests

Die Verbesserungen der letzten Abschnitte werden wieder auf statistische Signifikanz überprüft. Die Vorgehensweise entspricht der auf S.56. Die relativen Fitnesswerte des GA-3112 und des GA-3112 Int(60)+TS mit allen Anpassungen der letzten Abschnitte werden nach verschiedenen Anzahlen an (Lösungs-) Bewertungen verglichen. Die Null- und Alternativhypothesen lauten:

$$1.\, H_0 :\ \mu_{3112} - \mu_{3112Int(60)+TS} \leq 0$$

$$2.\, H_1 :\ \mu_{3112} - \mu_{3112Int(60)+TS} > 0$$

Die Ergebnisse sind in Tabelle 3.4 dargestellt.

Bewertungs #	Rel. Fitnesswert GA-3112	Rel. Fitnesswert GA-3112 Int(60)+TS	Differenz (p-Wert)
1000			
Mittelwert	8.51	7.26	1.25
(Varianz)	(0.23)	(2.49)	(0.12)
2000			
Mittelwert	5.36	5.08	0.28
(Varianz)	(0.55)	(0.94)	(0.14)
5000			
Mittelwert	3.12	2.16	0.96*
(Varianz)	(0.19)	(1.63)	(0.07)
7500 (Endergebnis)			
Mittelwert	2.47	1.53	0.94
(Varianz)	(0.82)	(2.16)	(0.26)

Tabelle 3.4: Ergebnisse der Signifikanztests. * entspricht einem Signifikanzniveau von 10%.

Die relativen Fitnesswerte des GA-3112 Int(60)+TS sind nur nach 5000 Bewertungen signifikant besser als die des GA-3112 (mit Signifikanzniveau $\alpha = 10\%$). An allen anderen Vergleichspunkten sind die Verbesserungen nicht signifikant.

3.7 Hybridisierung mit einer lokalen Suchheuristik

In diesem Abschnitt wird eine hybride Variante des GA-3112 Int(60)+TS vorgestellt und getestet. In Kapitel 2.3.3 wurde bereits erwähnt, dass Kombinationen von (Meta-) Heuristiken oftmals leistungsfähiger sind als deren Komponenten für sich, da sie ihre Nachteile gegenseitig ausgleichen . Ein Beispiel für eine hybride Metaheuristik ist die Kombination von evolutionären Algorithmen (z.B. GAs) mit lokalen Suchheuristiken. Evolutionäre Algorithmen sind gut geeignet für die Erkundung großer (hochdimensionaler) Lösungsräume und für die Identifikation von Bereichen mit guten Lösungen darin. Für das schnelle erreichen von lokalen Optima in diesen Bereichen sind jedoch lokale Suchheuristiken besser geeignet. Die Realisierungen solcher Kombinationen unterscheiden sich z.B. in der Häufigkeit der Anwendung von Local Search während des Optimierungsvorgangs oder der Anzahl von Individuen, die mittels Local Search verbessert werden. Eine Möglichkeit ist beispielsweise die Verbesserung aller Lösungen einer Population mittels LS, bevor die Operatoren des EA angewendet werden [29].

In dieser Arbeit wird der GA-3112 Int(60)+TS mit allen Anpassungen aus den letzten Kapiteln als übergeordneter Algorithmus verwendet. Hauptkriterien für die Auswahl einer passenden lokalen Suchmethode sind:

- Die Suchmethode muss effizient sein, d.h. sie muss möglichst schnell ein lokales Optimum finden können.
- Die Nebenbedingungen der Optimierung müssen eingehalten werden.
- Die Suchmethode muss für ganzzahlige Optimierung geeignet sein.

Diese Bedingungen werden von den zur Verfügung stehenden Algorithmen der MATLAB Global Optimization Toolbox nur vom Genetischen Algorithmus erfüllt. Andere Algorithmen wie z.B. Simulated Annealing unterstützen keine linearen Nebenbedingungen oder erfüllen die anderen Kriterien nicht. Aus diesem Grund wird als lokale Suchmethode ebenfalls ein GA-3112 Int(60)+TS, jedoch mit kleiner Populationsgröße verwendet. Wie bereits in Kapitel 3.6.1 gezeigt, führt eine kleine Populationsgröße zu einer höheren Geschwindigeit des Algorithmus. Die Ergebnisse werden dabei aber unzuverlässig, da der Algorithmus vorzeitig konvergieren kann (siehe Abbildung 3.24).

Bei der Kombination der beiden Algorithmen muss entschieden werden, wann und wie oft die lokale Suche eingesetzt wird. Es wird eine Variante getestet, bei der nur das beste Individuum (das Eliteindividuum) der jeweiligen Generation mittels LS verbessert wird und anschließend wieder in die Population des globalen GA übernommen wird. Um viele Bewertungen ohne Verbesserung des Fitnesswerts zu verhindern, wird als Abbruchkriterium des LS-GA ein maximales Limit für Generationen ohne Verbesserung von 10 verwendet. So können maximal 50 Bewertungen ohne Verbesserung (bei einer Populationsgröße von $P = 5$) durchgeführt werden, bevor wieder die globale Suche fortgesetzt wird. Der LS-GA kommt anschließend erst wieder zum Einsatz, nachdem die globale Suche eine bessere Lösung gefunden hat. Alle anderen Einstellungen werden beibehalten.

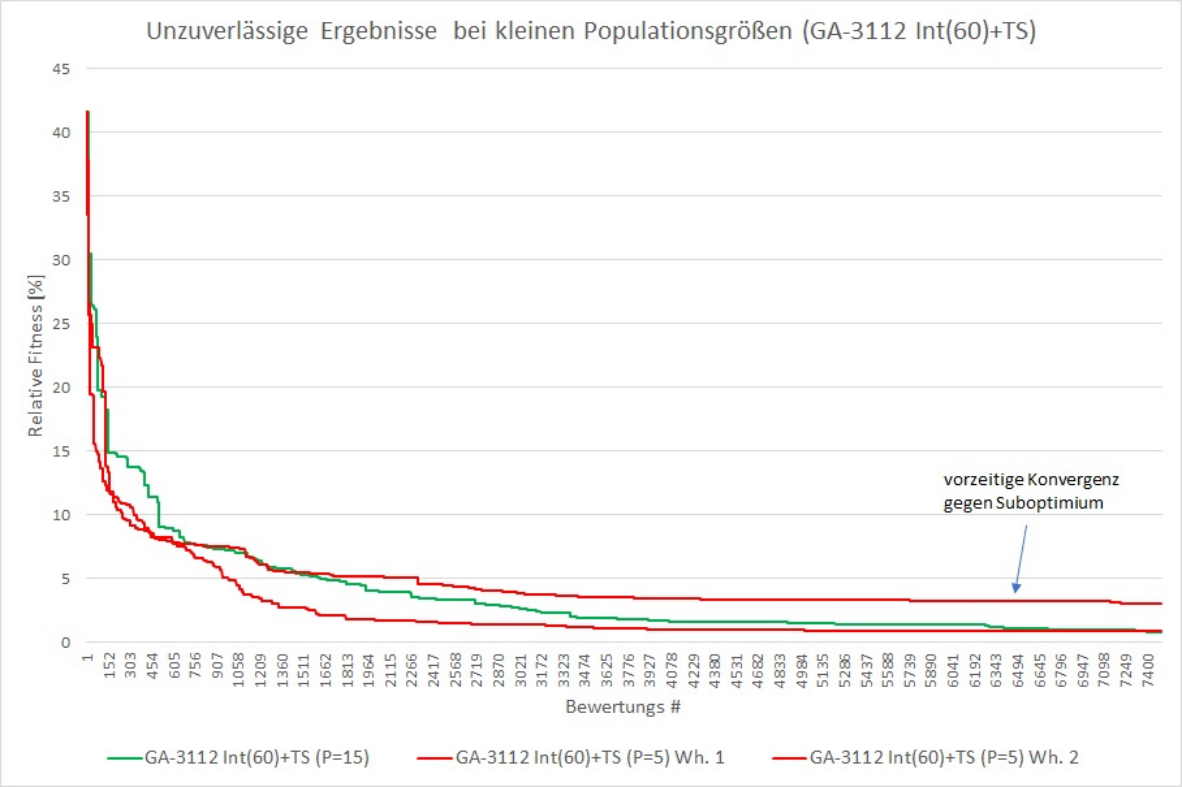

Abbildung 3.24: GA-3112 Int(60)+TS mit kleinen Populationsgrößen

Abbildung 3.25 zeigt den mittleren Verlauf der relativen Fitnesswerte für den GA-3112 Int(60)+TS hybrid. Der Algorithmus ist schneller als der GA-3112 Int(60)+TS und optimiert die Zielfunktion zuverlässig, ohne vorzeitig gegen ein lokales Optimum zu konvergieren. Tabelle 3.5 zeigt, dass der GA-3112 Int(60)+TS hybrid ca. 56% weniger Bewertungen benötigt, um den Zielfunktionswert auf 105% des bekannten Minimums zu reduzieren. Da das Ausführen der Simulation den weitaus größten Teil der Laufzeit beansprucht, wird damit auch der gesamte Zeitaufwand deutlich reduziert.

Relativer Fitnesswert (bez. auf bek. Minimum)	**GA-3112 Int(60)+TS**	**GA-3112 Int(60)+TS hybrid**	**Differenz**
115%	153	134	-12.4%
110%	548	322	-41.2%
105%	2045	883	-56.8%
101%	-	5565	-

Tabelle 3.5: Benötigte Anzahl an Bewertungen für das Erreichen von Fitnessniveaus

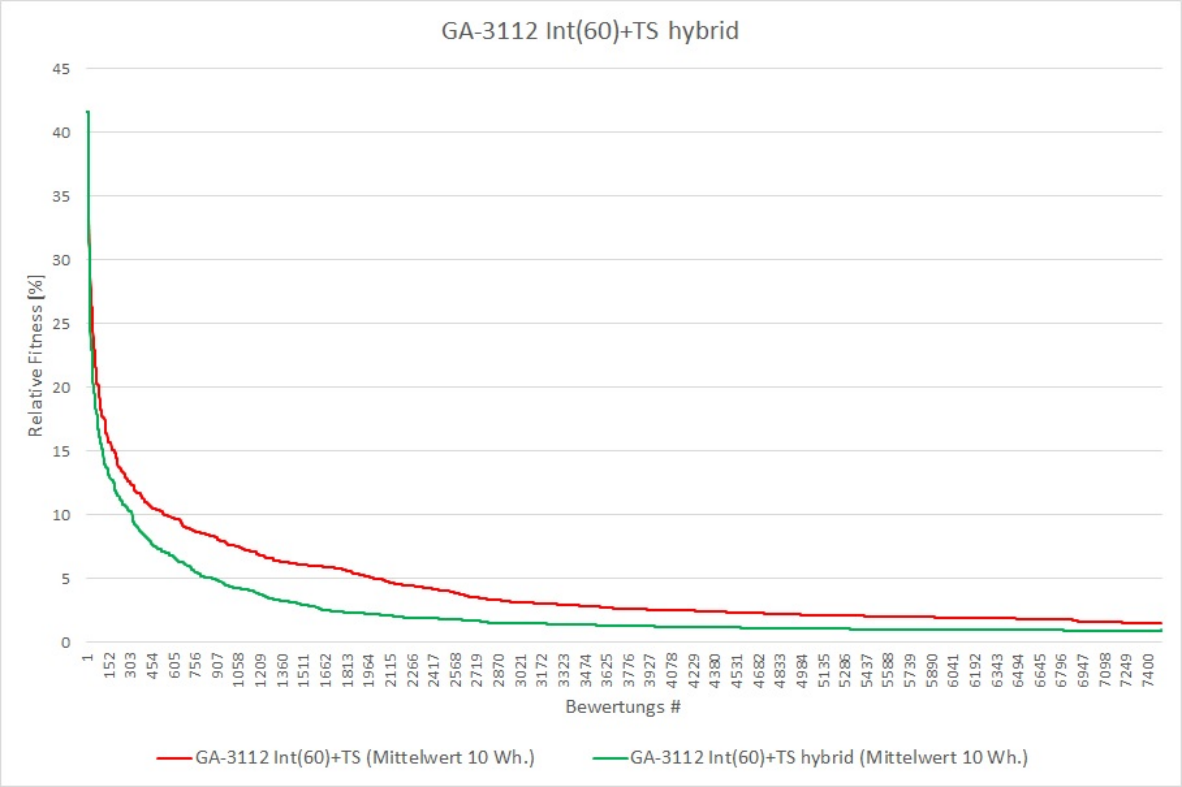

Abbildung 3.25: GA-3112 Int(60)+TS hybrid

3.7.1 Signifikanztests

Tabelle 3.6 zeigt, dass durch die Hybridisierung an allen Messpunkten, außer beim Endergebnis, signifikant bessere Ergebnisse erzielt werden. Die Vorgehensweise bei der Durchführung der Signifikanztests entspricht der auf S.56.

Bewertungs #	Rel. Fitnesswert GA-3112 Int(60)+TS	Rel. Fitnesswert GA-3112 Int(60)+TS hybrid	Differenz (p-Wert)
1000			
Mittelwert	7.26	4.39	2.87***
(Varianz)	(2.49)	(1.96)	(<0.01)
2000			
Mittelwert	5.08	2.35	2.73***
(Varianz)	(0.94)	(0.50)	(<0.01)
5000			
Mittelwert	2.16	1.16	1.00**
(Varianz)	(1.63)	(0.11)	(0.02)
7500 (Endergebnis)			
Mittelwert	1.53	0.95	0.58
(Varianz)	(2.16)	(0.16)	(0.15)

Tabelle 3.6: Ergebnisse der Signifikanztests. *,**,*** entsprechen einem Signifikanzniveau von 10%, 5% bzw. 1%.

In Abbildung 3.26 sind zusammenfassend die Ergebnisse der Verbesserungsmaßnahmen nach den einzelnen Abschnitten dargestellt. Zusätzlich ist der Fitnessverlauf für den Hybrid Tuned Int (60) GA+PS aus [50] abgebildet.

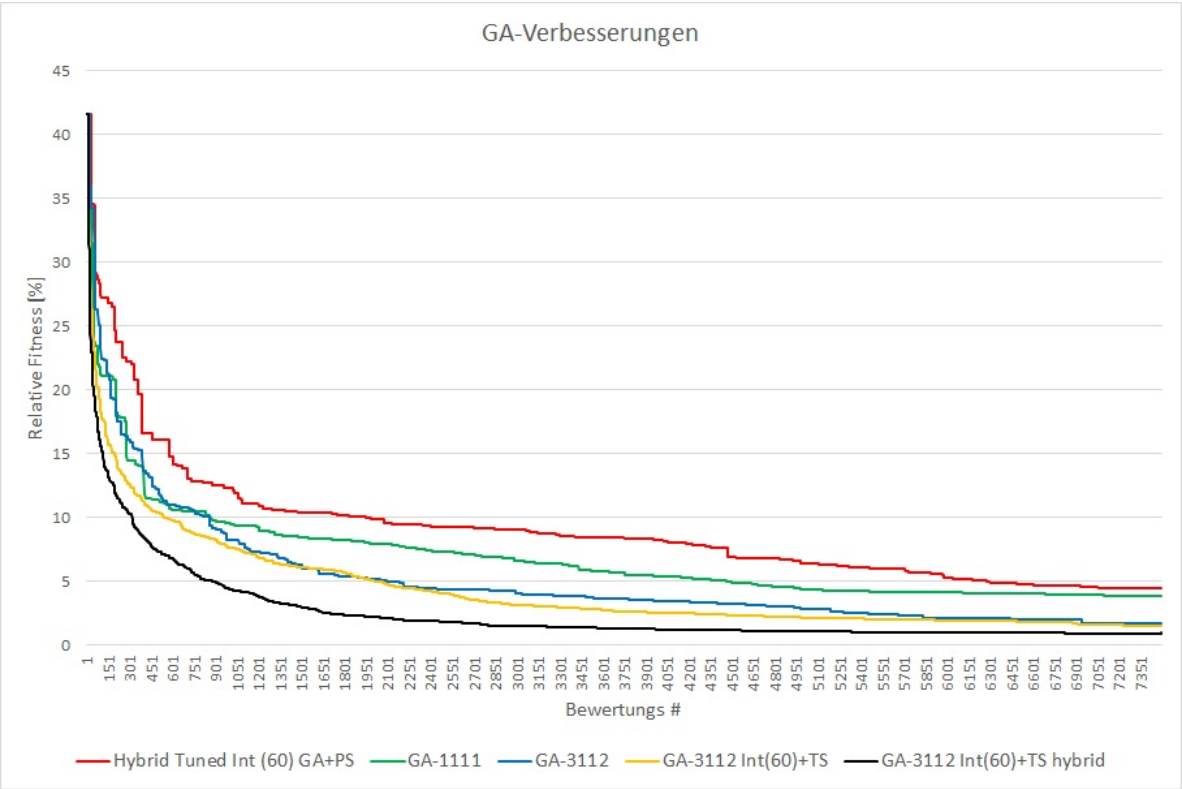

Abbildung 3.26: Performance des GA nach Verbesserungsmaßnahmen

4 Zusammenfassung und Ausblick

4.1 Zusammenfassung

Die Hauptziele dieser Arbeit waren:

- Die Auswahl geeigneter Optimierungsalgorithmen für ein Optimierungsproblem der simulationsbasierten Optimierung.
- Der Vergleich zweier Algorithmen (PSO und GA) an dem Optimierungsproblem.
- Die Verbesserung eines bestehenden Optimierungsmoduls, welches auf einem genetischen Algorithmus basiert.

Das betrachtete Simulationsmodell ist eine vereinfachte Abbildung einer realen Produktionslinie für Backwaren. Neben Materialflüssen umfasst die Simulation auch den Energieverbrauch der Anlage sowie das thermisch- physikalische Verhalten der Aggregate. Das Optimierungsproblem besteht hauptsächlich aus der Suche nach Reihenfolgen und Einsteuerungszeitpunkten für Produktionslose um eine Zielfunktion zu minimieren. Diese setzt sich aus unterschiedlich gewichteten Teilzielen zusammen und enthält neben dem Energieverbrauch auch Anteile für den Lieferverzug, die Lagerhaltungskosten, die Durchlaufzeit und einen Anteil für nicht fertig produzierte Produkte. Ziel der Optimierung ist daher das Auffinden eines Produktionsplanes, welcher die Zielfunktion für ein Testszenario minimiert.

Die Einordnung des Problems im Rahmen des Operations Research erfolgte in Kapitel 2.5.2. Die Zuordnung der Aufträge (Produkte) zu den Maschinen ist ein Problem der Maschinenbelegungsplanung (Scheduling). Aufgrund der Anordnung der Maschinen (Fließproduktion) weist das Problem Ähnlichkeiten mit Flow-Shop-Scheduling Problemen auf. Die einzelnen Produkte können sich während der Herstellung nicht überholen, was dazu führt, dass die Bearbeitungsreihenfolge auf jeder Maschine identisch ist. Dieser Spezialfall wird als Permutations-Flow-Shop-Scheduling Problem (PFSSP) bezeichnet. Das vorliegende Problem unterscheidet sich jedoch in einigen Aspekten von den in der Literatur häufig behandelten PFSS Standardproblemen. Dazu gehören die Miteinbeziehung der Energiekosten in die Optimierung, die Verwendung eines hybriden Simulationsmodells, die zusätzliche (optimale) Steuerung der Maschinen und eine spezielle Anordnung der Maschinen, die Produkttyp-gesteuerte Verzweigungen enthält. Verschiedene Lösungsmethoden für spezielle Standardprobleme aus der Literatur

konnten daher nicht direkt auf die vorliegende Problemstellung übertragen werden.

In Kapitel 2.5.2 wurde außerdem gezeigt, dass dieses Optimierungsproblem NP-schwer ist, was bedeutet, dass es mittels exakter Methoden nicht effizient lösbar ist. Diese Art von Optimierungsproblemen ist mittels Heuristiken nur näherungsweise lösbar. Zusätzlich liegt die Zielfunktion nicht in geschlossener, mathematischer Form vor, sondern muss für jeden Lösungsvorschlag durch die zeitintensive Simulation ausgewertet werden. Dem Optimierungsalgorithmus stehen also keine Informationen über die zugrundeliegende Funktion wie z.B. dem Gradienten zur Verfügung. In Kapitel 2.7 wurde gezeigt, dass sich insbesondere Metaheuristiken für diese Problemart eignen. Vor allem populationsbasierte Metaheuristiken sind in der Lage, mit angemessenem Zeitaufwand, gute Lösungen in hochdimensionalen Lösungsräumen mit multiplen lokalen Optima zu finden.

Im zweiten Abschnitt der Arbeit wurden zwei populationsbasierte Metaheuristiken (PSO und GA) an dem Problem verglichen. Dazu wurden die Algorithmen der Global Optimization Toolbox in MATLAB mit Standardeinstellungen verwendet. Beide Algorithmen konnten die Zielfunktion im Vergleich zu einer Startlösung reduzieren, das bisher bekannte Minimum für das verwendete Testszenario (siebentägiger Simulationszeitraum) wurde aber von keinem der beiden Algorithmen erreicht. Der PSO-Algorithmus lieferte bei den Tests die besseren Ergebnisse. Der Algorithmus erzeugte aber, bedingt durch die fehlende Unterstützung linearer Nebenbedingungen, einen großen Anteil ungültiger Lösungen. Da die Entwicklung eines eigenen PSO-Algorithmus den Rahmen der Arbeit gesprengt hätte, wurden nur für den GA weitere Anpassungen durchgeführt.

Ausgehend von einem bestehenden, GA-basierten Optimierungsmodul wurden verschiedene GA-Varianten getestet. Dies umfasste den Test verschiedener Skalierungs-, Selektions-, und Mutationsmethoden mithilfe einer angepassten Funktion für die Erzeugung neuer Lösungen. Alle getesten Varianten minimierten den Zielfunktionswert zuverlässig. Ausgehend von einer Startlösung (die aus realen Betriebsbedingungen abgeleitet ist) konnte der Zielfunktionswert um ca. 29% reduziert werden. Die Verbesserungen teilen sich folgendermaßen auf die Teil-Zielfunktionswerte auf:

- -19% Energiekosten
- -100% Lieferverzug (zu späte Fertigstellungszeitpunkte wurden vollständig verhindert)

- -34% Lagerhaltungskosten
- +1.5% Gesamtdurchlaufzeit

Bis auf eine kleine Verschlechterung des Anteils für die Gesamtdurchlaufzeit konnten alle Teilzielfunktionswerte verbessert werden. Der resultierende Algorithmus GA-3122 lieferte an definierten Messpunkten (nach verschiedenen Bewertungsanzahlen) signifikant bessere Ergebnisse als der ursprüngliche Algorithmus GA-1111.

Weitere Verbesserungsmaßnahmen umfassten die Einführung von ganzzahligen Lösungsvektoreinträgen, die Einführung einer Tabu-Liste für bereits bewertete Lösungen und den Test weiterer Nebenbedingungen. Für den resultierenden GA-3112 Int(60)+TS wurden in Kapitel 3.6 außerdem einige Parametereinstellungen vorgenommen. Die so erzielten Verbesserungen gegenüber dem GA-3112 waren aber nur teilweise signifikant. Es wird erwartet, dass sich die Verbesserungsmaßnahmen, vor allem die Beschränkung auf ganzzahlige Lösungsvektoren, stärker bei komplexeren Problemen auswirken.

Als letzte Verbesserungsmaßnahme wurde die Hybridisierung mit einer schnellen Variante des GA-3112 Int(60)+TS getestet. Durch die Verwendung einer sehr kleinen Populationsgröße P wird der Algorithmus schneller, die Ergebnisse werden jedoch unzuverlässig, da der Algorithmus meist vorzeitig konvergiert. Durch die Integration in einen übergeordneten GA mit größerem P konnte dieser Nachteil ausgeglichen werden. Im Vergleich zum GA-3112 Int(60)+TS konnten die Ergebnisse somit nochmals signifikant verbessert werden.

Der ursprüngliche Algorithmus GA-1111 benötigt ca. 4.500 Bewertungen (Iterationen) für das erreichen eines 105% Fitnessniveaus (bezogen auf das bekannte Minimum). Der in dieser Arbeit entwickelte Algorithmus GA-3112 Int(60)+TS hybrid benötigt dafür ca. 900 Bewertungen.

4.2 Ausblick

Der weitaus größte Zeitanteil bei der Optimierung mit der entwickelten Methode entfällt auf die Simulationslaufzeit. Eine Verkürzung der Laufzeit bietet somit großes Verbesserungspotential. Ein Ziel weiterführender Arbeiten sollte daher die Entwicklung einer schnelleren Simulation sein.

Eine weitere Verbesserungsmöglichkeit wäre die parallelisierte Auswertung mehrerer Lösungsvorschläge. Dies bietet sich durch die Verwendung eines populationsbasierten Algorithmus an. Anstatt die Lösungsvektoren jeder Population sequentiell auszuwerten, könnte die Auswertung auch gleichzeitig, durch parallele Simulationsläufe, erfolgen.

Die Integration einer Selbstadaption des Algorithmus wäre ein weiterer Verbesserungsansatz. Anstatt dem Algorithmus verschiedene Parameterwerte fix zuzuweisen, könnte ein Lernmechanismus integriert werden, der die Parameterauswahl während des Optimierungslaufs übernimmt. Dies würde zu einer universelleren Anwendbarkeit führen.

A Anhang (MATLAB Code)

A.1 Mutationsoperator für den Test verschiedener GA Varianten

```
function  xoverKids  = mutationFunction(~,options,GenomeLength,~,...
    ~,thisPopulation,~)
global t_end;
global optInterval;
global generations;
global nebenbedSchranken;
global nebenbedKoefMatrix;
global scalingMethod;
global selectionMethod;
global mutationMethod;
global mutationMethodValue;

% Zaehler fuer Generationen
generations = generations +1;
% Struktur des Loesungsvektors
vectorIndices=getGlobalIndices;
% Obergrenze fuer Loesungseintraege
maxSteps = t_end/optInterval;
% Minimale Bearbeitungszeiten auf den Maschinen
[ovenMin,freezerMin] = getMinTimeSpans;

% Definition der Parameter fuer die Wertmutation
if mutationMethodValue==2
    maxChange=25;
    maxChange1=25;
else
    maxChange=25;
    maxChange1=25;
end
% Definition der Wahrscheinlichkeitsdichtefunktion fuer die
% normalverteilte Wertmutation
if mutationMethodValue==2
    maxChange=maxChange*(1-(generations-1)/options.Generations);
    pd = makedist('Normal','mu',0,'sigma',maxChange);
end
```

```
% Steuerung der Operatoren (da ein verschieben von Betriebszeiten zu Beginn
% der Optimierung nicht moeglich ist)
if generations <=2*GenomeLength
    changeOptions=3;
else
    changeOptions=4;
end

%% Skalierung der Fitnesswerte der aktuellen Population
% Bestimmung der benoetigten Anzahl an Elternindividuen
nParents=size(thisPopulation,1)-options.EliteCount;
% Laden der Fitnesswerte
allScores=getGlobalGoals;
% Bestimmen der Fitnesswerte dieser Population
if generations~=1
    scores=allScores(1,((generations-1)*size(thisPopulation,1)+1):end);
else
    scores=allScores(1,(1:end));
end
% Skalierung
switch scalingMethod
    case 1
        expectation=fitscalingrank(scores,nParents);
    case 2
        expectation=fitscalingprop(scores,nParents);
    case 3
        expectation=fitscalingtop(scores,nParents);
    case 4
        expectation=fitscalingshiftlinear(scores,nParents);
end

%% Selektion fuer die Mutation
switch selectionMethod
    case 1
        parents=selectionroulette(expectation,nParents,options);
    case 2
        parents=selectionstochunif(expectation,nParents,options);
    case 3
        parents=selectiontournament(expectation,nParents,options);
end
```

```
%% Mutation durch Anwendung der Operatoren
for kidNumber=1:nParents
    % Auswahl des naechsten Elternindividums
    nextKid=thisPopulation(parents(kidNumber),:);
    % Fuenf Versuche, um eine gueltige Loesung zu erzeugen
    for t=1:5
        % Zufaellige Auswahl des Mutationsoperators
        changeMethod=randi(changeOptions);
        switch changeMethod
        case 1 %% Mutation Storage und Merger (Losreihenfolge)
            switch mutationMethod
                case 1 % Displacement Mutation
                %Auswahl der zu verschiebenden Sektion (Storage)
                point1=0;
                point2=0;
                while point1>=point2 || (point2-point1+1)==...
                    (vectorIndices(2)-1)
                    point1=randi(vectorIndices(2)-1);
                    point2=randi(vectorIndices(2)-1);
                end
                shiftSection=nextKid(1,point1:point2);
                nextKid=[nextKid(1,1:(point1-1)) nextKid(1,(point2+1):...
                end)];
                %Auswahl des Einfuegepunkts
                newIndices=vectorIndices(2)-1-(point2-point1+1);
                point3=randi(newIndices);
                %Einfuegen des zu verschiebenden Anteils (Storage)
                nextKid=[nextKid(1,1:point3) shiftSection nextKid(1,...
                (point3+1):end)];
                %Uebertragung auf den Merger
                point1=vectorIndices(2)+point1-1;
                point2=vectorIndices(2)+point2-1;
                shiftSection=nextKid(1,point1:point2);
                nextKid=[nextKid(1,1:(point1-1)) nextKid(1,(point2+1):...
                end)];
                point3=vectorIndices(2)+point3-1;
                nextKid=[nextKid(1,1:point3) shiftSection nextKid(1,...
                (point3+1):end)];

                case 2 %Exchange Mutation
```

```
%Auswahl der Swap-Gene
point1=0;
point2=0;
while point1==point2
    point1=randi(vectorIndices(2)-1);
    point2=randi(vectorIndices(2)-1);
end
%Austausch der Gene (Storage)
val1=nextKid(point1);
val2=nextKid(point2);
nextKid(point2)=val1;
nextKid(point1)=val2;
%Austausch der Gene (Merger)
point1=vectorIndices(2)+point1-1;
point2=vectorIndices(2)+point2-1;
val1=nextKid(point1);
val2=nextKid(point2);
nextKid(point2)=val1;
nextKid(point1)=val2;

case 3 %Scramble Mutation
%Auswahl des Scramble-Subsets
point1=0;
point2=0;
while point1>=point2
    point1=randi(vectorIndices(2)-1);
    point2=randi(vectorIndices(2)-1);
end
%Scrambling
for j=1:((point2-point1)*5)
    %Storage
    pos1=randi([point1,point2]);
    pos2=randi([point1,point2]);
    val1=nextKid(pos1);
    val2=nextKid(pos2);
    nextKid(pos2)=val1;
    nextKid(pos1)=val2;
    %Merger
    posm1=vectorIndices(2)+pos1-1;
    posm2=vectorIndices(2)+pos2-1;
    valm1=nextKid(posm1);
```

```
    valm2=nextKid(posm2);
    nextKid(posm2)=valm1;
    nextKid(posm1)=valm2;
end

case 4 %Insertion Mutation
%Auswahl der zu verschibenden Stelle und der
%Einfuegestelle
point1=0;
point2=0;
while point1>=point2
    point1=randi(vectorIndices(2)-1);
    point2=randi(vectorIndices(2)-1);
end
%Verschieben (Storage)
shiftVal=nextKid(point1);
shiftSection=nextKid(1,(point1+1):point2);
nextKid(1,point1:(point2-1))=shiftSection;
nextKid(1,point2)=shiftVal;
%Verschieben (Merger)
point1=vectorIndices(2)+point1-1;
point2=vectorIndices(2)+point2-1;
shiftVal=nextKid(point1);
shiftSection=nextKid(1,(point1+1):point2);
nextKid(1,point1:(point2-1))=shiftSection;
nextKid(1,point2)=shiftVal;

case 5 %Inversion Mutation
%Auswahl der Sektion fuer die Inversion
point1=0;
point2=0;
while point1>=point2
    point1=randi(vectorIndices(2)-1);
    point2=randi(vectorIndices(2)-1);
end
%Vertauschen der Reihenfolge (Storage)
invSection=nextKid(1,point1:point2);
invSection=fliplr(invSection);
nextKid(1,point1:point2)=invSection;
%Vertauschen der Reihenfolge (Merger)
point1=vectorIndices(2)+point1-1;
```

```
            point2=vectorIndices(2)+point2-1;
            invSection=nextKid(1,point1:point2);
            invSection=fliplr(invSection);
            nextKid(1,point1:point2)=invSection;

            case 6 %Displaced Inversion Mutation
            %Auswahl der Sektion fuer die Vertauschung und
            %Verschiebung
            point1=0;
            point2=0;
            while point1>=point2 || (point2-point1+1)==...
                (vectorIndices(2)-1)
                point1=randi(vectorIndices(2)-1);
                point2=randi(vectorIndices(2)-1);
            end
            shiftSection=nextKid(1,point1:point2);
            %Invertieren
            shiftSection=fliplr(shiftSection);
            nextKid=[nextKid(1,1:(point1-1)) nextKid(1,(point2+1):end)];
            %Auswahl des Einfuegepunkts
            newIndices=vectorIndices(2)-1-(point2-point1+1);
            point3=randi(newIndices);
            %Einfuegen invertierten Anteils (Storage)
            nextKid=[nextKid(1,1:point3) shiftSection nextKid(1,...
            (point3+1):end)];
            %Uebertragung auf den Merger
            point1=vectorIndices(2)+point1-1;
            point2=vectorIndices(2)+point2-1;
            shiftSection=nextKid(1,point1:point2);
            shiftSection=fliplr(shiftSection);
            nextKid=[nextKid(1,1:(point1-1)) nextKid(1,(point2+1):end)];
            point3=vectorIndices(2)+point3-1;
            nextKid=[nextKid(1,1:point3) shiftSection nextKid(1,...
            (point3+1):end)];
        end

    case 2 %% Zufaellige Mutation von Storage- und Mergerpositionen
        switch mutationMethodValue
            case 1 % Gleichverteilt
            % Auswahl der Anzahl an Positionen fuer die Mutation
            mutatedPositions=randi(vectorIndices(2)-1);
```

```
sign=[-1 1];
% Mutation
for l=1:mutatedPositions
    point=randi(vectorIndices(2)-1);
    mutatedPart=nextKid(1,point);
    sign1=sign(1,randi(2));
    change=sign1*randi(maxChange)/100;
    mutatedPart=mutatedPart+change*mutatedPart;
    % Beschraenkungen einhalten
    if mutatedPart <= 0
        mutatedPart=0;
    end
    if mutatedPart >= maxSteps
        mutatedPart=maxSteps;
    end
    % Uebertragung auf Merger
    difference=mutatedPart-nextKid(1,point);
    nextKid(1,point)=mutatedPart;
    nextKid(1,vectorIndices(2)+point-1)=nextKid(1,...
    vectorIndices(2)+point-1)+difference;
end

case 2 % Normalverteilt
% Auswahl der Anzahl an Positionen fuer die Mutation
mutatedPositions=randi(vectorIndices(2)-1);
% Mutation
for k=1:mutatedPositions
    point=randi(vectorIndices(2)-1);
    mutatedPart=nextKid(1,point);
    change=random(pd)/100;
    mutatedPart=mutatedPart+change*mutatedPart;
    % Beschraenkungen einhalten
    if mutatedPart <= 0
        mutatedPart=0;
    end
    if mutatedPart >= maxSteps
        mutatedPart=maxSteps;
    end
    % Uebertragung auf Merger
    difference=mutatedPart-nextKid(1,point);
    nextKid(1,point)=mutatedPart;
```

```
                nextKid(1,vectorIndices(2)+point-1)=nextKid(1,...
                vectorIndices(2)+point-1)+difference;
            end
        end

    case 3 %% Zufaellige Mutation von Ofen- oder Frosterpositionen
        % Wahl von Ofen- oder Frosterposition
        if randi(2)<2
            point = randi([vectorIndices(3) vectorIndices(4)-1],1,1);
            minOperaTime = ovenMin;
        else
            point = randi([vectorIndices(4) size(nextKid,2)-1],1,1);
            minOperaTime = freezerMin;
        end
        switch mutationMethodValue
            case 1 % Gleichverteilt
            mutatedPart = nextKid(point);
            change=randi(maxChange)/100;
            if bitand(point,1)
            % End-Punkt
                dev_amount = nextKid(point) - nextKid(point-1);
                mutatedPart = mutatedPart - change*dev_amount;
                operaTime = mutatedPart - nextKid(point-1);
            else
            % Start-Punkt
                dev_amount = nextKid(point+1) - nextKid(point);
                mutatedPart = mutatedPart + change*dev_amount;
                operaTime= nextKid(point+1) - mutatedPart;
            end
            if mutatedPart > 0 && mutatedPart < maxSteps
                %Kriterium/Restriktion minTimeSpan erfuellt?
                if operaTime >= minOperaTime
                    %mutieren
                    nextKid(1,point) = mutatedPart;
                end
            end

            case 2 % Normalverteilt
            mutatedPart = nextKid(point);
            change=abs(random(pd)/100);
            if bitand(point,1)
```

```
            % End-Punkt
                dev_amount = nextKid(point) - nextKid(point-1);
                mutatedPart = mutatedPart - change*dev_amount;
                operaTime = mutatedPart - nextKid(point-1);
            else
            % Start-Punkt
                dev_amount = nextKid(point+1) - nextKid(point);
                mutatedPart = mutatedPart + change*dev_amount;
                operaTime= nextKid(point+1) - mutatedPart;
            end
            if mutatedPart > 0 && mutatedPart < maxSteps
                %Kriterium/Restriktion minTimeSpan erfuellt?
                if operaTime >= minOperaTime
                    %mutieren
                    nextKid(1,point) = mutatedPart;
                end
            end
        end

    case 4 %% Verschieben von Ofen- und Frosterbetriebszeiten
        if randi(2) < 2
            % oven tupel
            oven=true;
            x = randi([vectorIndices(3) vectorIndices(4)-1],1,1);
        else
            % freezer tupel
            oven=false;
            x = randi([vectorIndices(4) size(nextKid,2)-1],1,1);
        end
        mutatedPart = nextKid(x);
        % Start oder Ende?
        if bitand(x,1)
            start = false;
        else
            start = true;
        end
        randomNumber = randi(maxChange1)/100;
        % delta zum vorherigen/naechsten Los
        delta = 0;
        if(start==true)
            %suche End-Zeitpunkt vorheriges Los, falls moeglich
```

```
        lastEnd = nextKid(x-1);
        if(oven==true)
        %vorheriges Los auch ein Ofen-Tupel?
            if (x-1) <= vectorIndices(4)-1 && (x-1) >= ...
                vectorIndices(3)
                delta = mutatedPart - lastEnd;
            end
        else
            %vorheriges Los auch ein Freezer-Tupel?
            if (x-1) >= vectorIndices(4) && (x-1) <= ...
                size(nextKid,2)-1
                delta = mutatedPart - lastEnd;
            end
        end
        change = randomNumber*delta;
        % change nextKid(x) & nextKid(x+1)IF delta > 0
        if delta > 0
            nextKid(x)= nextKid(x)- change;
            nextKid(x+1)= nextKid(x+1) - change;
        end
    else % x == end
        %suche Start-Zeitpunkt naechstes Los, falls moeglich
        nextStart = nextKid(x+1);
        if(oven==true)
            %naechstes Los auch ein Ofen-Tupel?
                if (x+1) <= vectorIndices(4)-1 && (x+1) >=...
                    vectorIndices(3)
                    delta = nextStart - mutatedPart;
                end
        else % freezer==true
            %naechstes Los auch ein Freezer-Tupel?
            if (x+1) >= vectorIndices(4) && (x+1) <=...
                size(nextKid,2)-1
                delta = nextStart - mutatedPart;
            end
        end
        change = randomNumber*delta;
        % change nextKid(x-1) & nextKid(x) OR delta = 0
        if delta > 0
            nextKid(x-1)= nextKid(x-1)+ change;
            nextKid(x)= nextKid(x)+ change;
```

```
                end
            end
        end

        %% Ueberpruefen, ob Loesung Nebenbedingungen entspricht
        a=nebenbedKoefMatrix*transpose(nextKid);
        b=transpose(nebenbedSchranken);
        comp=a<=b;
        nbSum=sum(comp);
        % Wenn erfuellt wird die Schleife (fuer 5 Versuche) verlassen
        if nbSum==size(nebenbedSchranken,2)
            break
        end
    end
    % Speichern der mutierten Loesung
    xoverKids(kidNumber,:)=nextKid;
end
end
```

A.2 Mutationsoperator für den GA-3112 Int(60)+TS

```
function  xoverKids  = mutationFunction3112IntegerTS(~,options,GenomeLength,~,...
~,thisPopulation,~)

global t_end;
global optInterval;
global generations;
global nebenbedSchranken;
global nebenbedKoefMatrix;
global batchData;
global changes;

% Zaehler fuer Generationen
generations = generations +1;
% Struktur des Loesungsvektors
vectorIndices=getGlobalIndices;
% Obergrenze fuer Loesungseintraege (Zeitlimit)
maxSteps = t_end/optInterval;
% Minimale Bearbeitungszeiten auf den Maschinen
[ovenMin,freezerMin] = getMinTimeSpans;
% Laden aller bisher bewerteten Loesungsvektoren (Tabu-Liste)
[~,solVecs]=getGlobalSolutionVectors;
% Berechnung der Bearbeitungzeiten pro Los im Lager (Zur Vermeidung von Ueberschneidunge
storageOperatingTimes=zeros(1,vectorIndices(2)-1);
for i=1:(vectorIndices(2)-1)
    storageOperatingTimes(1,i)=batchData(2,i)*30;
end

% Definition der Wahrscheinlichkeitsdichtefunktionen fuer die Wertmutation
maxChange=35;
maxChange1=35;
pd = makedist('Normal','mu',0,'sigma',maxChange);
pd1= makedist('Normal','mu',0,'sigma',25);

% Steuerung der einzelnen Operationen mittels Wahrscheinlichkeiten
% (Die Verschiebung von Betriebszeiten ist zu Beginn der Optimierung nicht moeglich)
if generations <=2*GenomeLength
    changeOptions=[1/3,1;2/3,2;1,3];
else
```

```
    changeOptions=[0.25,1;0.5,2;0.75,3;1,4];
end

%% Skalierung der Fitnesswerte der aktuellen Population
nParents=size(thisPopulation,1)-options.EliteCount;
allScores=getGlobalGoals;
if generations~=1
    scores=allScores(1,end-size(thisPopulation,1)+1:end);
else
    scores=allScores(1,(1:end));
end
expectation=fitscalingtop(scores,nParents);

%% Selektion fuer die Mutation
parents=selectionroulette(expectation,nParents,options);

%% Uebernehmen des besten Individuums in die naechste Generation (Eliteindividuum)
bestInd=find(scores==min(scores));
xoverKids(1,:)=thisPopulation(bestInd(1,1),:);
methodInd=0;
if size(changes,1)==0
    changes=methodInd;
else
    changes=[changes methodInd];
end

%% Steuerung fuer grosse oder kleine Veraenderungen der Loesungsvektoren
if generations>166
    smallChange=1;
else
    smallChange=0;
end

%% Mutation
for kidNumber=2:nParents
    % Uebernehmen des naechsten Elterindividuums
    nextKid=thisPopulation(parents(kidNumber),:);
    for t=1:5
        % Auswahl der Operation per Zufall
        ch=rand;
        for i=1:size(changeOptions,1)
```

```
    if i==1
        if ch<=changeOptions(i,1)
            chMeth=changeOptions(i,2);
            break;
        end
    else
        if ch<=changeOptions(i,1) && ch>changeOptions(i-1,1)
            chMeth=changeOptions(i,2);
            break;
        end
    end
end

switch chMeth
case 1 %% Mutation Storage und Merger (Losreihenfolge)
    if smallChange ==0
        %Displacement Mutation
        %Auswahl der zu verschiebenden Sektion (Storage)
        point1=0;
        point2=0;
        order=1:(vectorIndices(2)-1);
        for i=1:(vectorIndices(2)-1)
            k=vectorIndices(3);
            order(2,i)=nextKid(1,k+1)-nextKid(1,k);
            k=k+2;
        end
        while point1>=point2 || (point2-point1+1)==...
            (vectorIndices(2)-1)
            point1=randi(vectorIndices(2)-1);
            point2=randi(vectorIndices(2)-1);
        end
        shiftSection=nextKid(1,point1:point2);
        nextKid=[nextKid(1,1:(point1-1)) nextKid(1,(point2+1):...
        end)];
        shiftSectionOrder=order(:,point1:point2);
        order=[order(:,1:(point1-1)) order(:,(point2+1):...
        end)];
        % Auswahl des Einfuegepunkts
        newIndices=vectorIndices(2)-1-(point2-point1+1);
        point3=randi(newIndices);
        % Einfuegen des zu verschiebenden Anteils (Storage)
```

```
        nextKid=[nextKid(1,1:point3) shiftSection nextKid(1,...
        (point3+1):end)];
        order=[order(:,1:point3) shiftSectionOrder order(:,...
        (point3+1):end)];
        % Uebertragung auf den Merger
        point1m=vectorIndices(2)+point1-1;
        point2m=vectorIndices(2)+point2-1;
        shiftSection=nextKid(1,point1m:point2m);
        nextKid=[nextKid(1,1:(point1m-1)) nextKid(1,(point2m+1):...
        end)];
        point3m=vectorIndices(2)+point3-1;
        nextKid=[nextKid(1,1:point3m) shiftSection nextKid(1,...
        (point3m+1):end)];
    else
        %Auswahl der Swap-Gene
        point1=0;
        point2=0;
        while point1==point2
            point1=randi(vectorIndices(2)-1);
            point2=randi(vectorIndices(2)-1);
        end
        %Austausch der Gene (Storage)
        val1=nextKid(point1);
        val2=nextKid(point2);
        nextKid(point2)=val1;
        nextKid(point1)=val2;
        %Austausch der Gene (Merger)
        point1=vectorIndices(2)+point1-1;
        point2=vectorIndices(2)+point2-1;
        val1=nextKid(point1);
        val2=nextKid(point2);
        nextKid(point2)=val1;
        nextKid(point1)=val2;
    end
    % Ueberpruefen, ob sich Bearbeitungszeiten ueberlappen und
    % reparieren der Loesungen, falls notwendig
    storageTimes=nextKid(1,1:vectorIndices(2)-1);
    storageTimes(2,:)=storageTimes(1,:)+...
    storageOperatingTimes;
    for i=1:size(storageTimes,2)
        storageTimes(3,i)=i;
```

```
        end
        storageTimes=sortrows(transpose(storageTimes));
        for i=1:(size(storageTimes,1)-1)
            if storageTimes(i+1,1)<storageTimes(i,2)||storageTimes(i+1,1)==...
            storageTimes(i,1)
                opStart=storageTimes(i+1,1);
                startIndex=storageTimes(i+1,3);
                opTime=storageTimes(i+1,2)-storageTimes(i+1,1);
                % Verschieben
                storageTimes(i+1,1)=storageTimes(i,2);
                storageTimes(i+1,2)=storageTimes(i+1,1)+...
                opTime;
                % Uebertragen auf Loesung
                storageTemp=nextKid(1,1:vectorIndices(2)-1);
                storageTemp(1,startIndex)=...
                storageTimes(i+1,1);
                nextKid=[storageTemp nextKid(1,...
                vectorIndices(2):end)];
                % Uebertragung auf Merger
                nextKid(1,vectorIndices(2)+startIndex-1)=...
                nextKid(1,startIndex)+30;
            end
        end
        methodInd =1;

    case 2 %% Zufaellige Mutation von Storage- und Mergerpositionen
        %normalverteilt
        if smallChange==0
            mutatedPositions=randi(vectorIndices(2)-1);
        else
            mutatedPositions=1;
        end
        for k=1:mutatedPositions
            point=randi(vectorIndices(2)-1);
            mutatedPart=nextKid(1,point);
            change=random(pd)/100;
            mutatedPart=round(mutatedPart+change*mutatedPart);
            if mutatedPart <= 0
                mutatedPart=0;
            end
            if mutatedPart >= maxSteps
```

```
                mutatedPart=maxSteps;
            end
            difference=mutatedPart-nextKid(1,point);
            nextKid(1,point)=mutatedPart;
            nextKid(1,vectorIndices(2)+point-1)=nextKid(1,...
            vectorIndices(2)+point-1)+difference;
        end
        % Ueberpruefen, ob sich Bearbeitungszeiten ueberlappen und
        % reparieren der Loesungen, falls notwendig
        storageTimes=nextKid(1,1:vectorIndices(2)-1);
        storageTimes(2,:)=storageTimes(1,:)+...
        storageOperatingTimes;
        for i=1:size(storageTimes,2)
            storageTimes(3,i)=i;
        end
        storageTimes=sortrows(transpose(storageTimes));
        for i=1:(size(storageTimes,1)-1)
            if storageTimes(i+1,1)<storageTimes(i,2)||storageTimes(i+1,1)==...
            storageTimes(i,1)
                opStart=storageTimes(i+1,1);
                startIndex=storageTimes(i+1,3);
                opTime=storageTimes(i+1,2)-storageTimes(i+1,1);
                % Verschieben
                storageTimes(i+1,1)=storageTimes(i,2);
                storageTimes(i+1,2)=storageTimes(i+1,1)+...
                opTime;
                % Uebertragen auf Loesung
                storageTemp=nextKid(1,1:vectorIndices(2)-1);
                storageTemp(1,startIndex)=...
                storageTimes(i+1,1);
                nextKid=[storageTemp nextKid(1,...
                vectorIndices(2):end)];
                % Uebertragung auf Merger
                nextKid(1,vectorIndices(2)+startIndex-1)=...
                nextKid(1,startIndex)+30;
            end
        end
        methodInd=2;

    case 3 %% Zufaellige Mutation von Ofen- oder Frosterpositionen
        if randi(2)<2
```

```
            point = randi([vectorIndices(3) vectorIndices(4)-1],1,1);
            minOperaTime = ovenMin;
        else
            point = randi([vectorIndices(4) size(nextKid,2)-1],1,1);
            minOperaTime = freezerMin;
        end
        %normalverteilt
        mutatedPart = nextKid(point);
        change=abs(random(pd1)/100);
        if bitand(point,1)
            % End-Punkt
            dev_amount = nextKid(point) - nextKid(point-1);
            mutatedPart = round(mutatedPart - change*dev_amount);
            operaTime = mutatedPart - nextKid(point-1);
        else
            % Start-Punkt
            dev_amount = nextKid(point+1) - nextKid(point);
            mutatedPart = round(mutatedPart + change*dev_amount);
            operaTime= nextKid(point+1) - mutatedPart;
        end
        if mutatedPart > 0 && mutatedPart < maxSteps
            % Kriterium/Restriktion minTimeSpan erfuellt?
            if operaTime >= minOperaTime
                %mutieren
                nextKid(1,point) = mutatedPart;
            end
        end
        methodInd=3;

    case 4 %% Verschieben von Ofen- und Frosterbetriebszeiten
        if randi(2) < 2
            % oven tupel
            oven=true;
            x = randi([vectorIndices(3) vectorIndices(4)-1],1,1);
        else
            % freezer tupel
            oven=false;
            x = randi([vectorIndices(4) size(nextKid,2)-1],1,1);
        end
        mutatedPart = nextKid(x);
        % Start oder Ende?
```

```
if bitand(x,1)
    start = false; % ID fuer Shiften
else
    start = true; % ID fuer Shiften
end
randomNumber = randi(maxChange1)/100;
delta = 0; % delta zum vorherigen/naechsten Los
if(start==true)
    %suche End-Zeitpunkt vorheriges Los, falls moeglich
    lastEnd = nextKid(x-1);
    if(oven==true)
        %vorheriges Los auch ein Ofen-Tupel?
        if (x-1) <= vectorIndices(4)-1 && (x-1) >= vectorIndices(3)
            delta = mutatedPart - lastEnd;
        end
    else
        %vorheriges Los auch ein Freezer-Tupel?
        if (x-1) >= vectorIndices(4) && (x-1) <= size(nextKid,2)-1
            delta = mutatedPart - lastEnd;
        end
    end
    change = round(randomNumber*delta);
    % change nextKid(x) & nextKid(x+1)IF delta > 0
    if delta > 0
        nextKid(x)= nextKid(x)- change;
        nextKid(x+1)= nextKid(x+1) - change;
    end
    else % x == end
        % suche Start-Zeitpunkt naechstes Los, falls moeglich
        nextStart = nextKid(x+1);
        if(oven==true)
        % naechstes Los auch ein Ofen-Tupel?
            if (x+1) <= vectorIndices(4)-1 && (x+1) >= ...
            vectorIndices(3)
                delta = nextStart - mutatedPart;
            end
        else % freezer==true
        % naechstes Los auch ein Freezer-Tupel?
        if (x+1) >= vectorIndices(4) && (x+1) <= size(nextKid,2)-1
            delta = nextStart - mutatedPart;
        end
```

```
                end
                change = round(randomNumber*delta);
                % change nextKid(x-1) & nextKid(x) OR delta = 0
                if delta > 0
                    nextKid(x-1)= nextKid(x-1)+ change;
                    nextKid(x)= nextKid(x)+ change;
                end
            end
            methodInd=4;
        end

        %% Ueberpruefen, ob Loesung neu ist und Nebenbedingungen entspricht
        newSol=1;
        for i=1:size(solVecs,1)
            solution = solVecs(i,:);
            if isequal(nextKid,solution)
                newSol=0;
                break;
            end
        end
        if newSol==1
            a=nebenbedKoefMatrix*transpose(nextKid);
            b=transpose(nebenbedSchranken);
            comp=a<=b;
            nbSum=sum(comp);
            if nbSum==size(nebenbedSchranken,2)
                break;
            end
        end
        end
        % Speichern der erzeugten Loesung
        xoverKids(kidNumber,:)=nextKid;
        if size(changes,1)==0
            changes=methodInd;
        else
            changes=[changes methodInd];
        end
    end
end
```

A.3 Mutationsoperator für den GA-3112 Int(60)+TS hybrid

```
function  xoverKids  = mutationFunction3112Integerneu(~,options,GenomeLength,~,...
~,thisPopulation,~)

global t_end;
global optInterval;
global generations;
global nebenbedSchranken;
global nebenbedKoefMatrix;
global batchData;
global lb;
global ub;
global Pplan_storage;
global Pplan_merging;
global Pplan_oven;
global Pplan_freezer;
global tendofopt;
global initduration;
global batchDetails;
global AplanGes;
global changes;

generations = generations +1;
vectorIndices=getGlobalIndices;
maxSteps = t_end/optInterval;
[ovenMin,freezerMin] = getMinTimeSpans;
[~,solVecs]=getGlobalSolutionVectors;
storageOperatingTimes=zeros(1,vectorIndices(2)-1);
% Berechnung der Bearbeitungzeiten pro Los im Lager
for i=1:(vectorIndices(2)-1)
    storageOperatingTimes(1,i)=batchData(2,i)*30;
end

maxChange=35;
maxChange1=35;
% Definition der Verteilungsfunktion fuer die Mutation
%maxChange=maxChange*(1-(generations-1)/(options.Generations));
pd = makedist('Normal','mu',0,'sigma',maxChange);
pd1= makedist('Normal','mu',0,'sigma',25);
```

```
if generations <=2*GenomeLength
    changeOptions=[0.3333,1;0.6666,2;1,3];
else
    changeOptions=[0.25,1;0.5,2;0.75,3;1,4];
end

%% Skalierung der Fitnesswerte der aktuellen Population
nParents=size(thisPopulation,1)-options.EliteCount;
allScores=getGlobalGoals;
if generations~=1
    scores=allScores(1,end-size(thisPopulation,1)+1:end);
else
    scores=allScores(1,(1:end));
end
expectation=fitscalingtop(scores,nParents);

%% Selektion fuer die Mutation
parents=selectionroulette(expectation,nParents,options);

%% Uebernehmen des besten Individuums in die naechste Generation (Eliteindividuum)
bestInd=find(scores==min(scores));
xoverKids(1,:)=thisPopulation(bestInd(1,1),:);
methodInd=0;
if size(changes,1)==0
    changes=methodInd;
else
    changes=[changes methodInd];
end

%% Verbesserung des besten Individuums durch Local Search
if generations~=1
    if min(scores)<min(allScores(1,1:end-size(thisPopulation,1)))
        successGlobal=1;
    else
        successGlobal=0;
    end
else
    successGlobal=1;
end
if successGlobal==1
```

```
    populationSizeLocal = 5;
    for i=1:populationSizeLocal
        initialPopulationLocal(i,:)=xoverKids(1,:);
    end
    nvarsLocal=size(initialPopulationLocal,2);
    % Definition der Optionen fuer LS
    optionsLocal = gaoptimset('PlotFcns',{@gaplotbestindiv,@gaplotbestf,...
    @gaplotstopping,@gaplotexpectation,@gaplotscorediversity,@gaplotscores,...
    @gaplotdistance,@gaplotrange},'InitialPopulation',initialPopulationLocal,...
    'PopulationSize',populationSizeLocal,'CrossoverFraction',1.0,...
    'CrossoverFcn',@mutationFunction3112Integerneu,'TolFun',1e-7,...
    'StallGenLimit',10,'EliteCount',0);
    minBeforeLocal=min(allScores);
    % Ausfuehren von LS
    [bestSolution,fvalLocal,exitflagLocal,outputLocal,populationLocal]=...
    ga(@(t)moba_fun(t,optInterval,tendofopt,initduration,batchDetails,...
    Pplan_storage,Pplan_merging, Pplan_oven,Pplan_freezer,AplanGes),...
    nvarsLocal,nebenbedKoefMatrix,nebenbedSchranken,[],[],lb,ub,[],optionsLocal);
    % Uebernehmen der Verbesserten Loesung in die Population des globalen GA
    xoverKids(1,:)=bestSolution;
end

%% Steuerung fuer grosse oder kleine Aenderungen der Loesungsvektoren
if generations>166
    smallChange=1;
else
    smallChange=0;
end

%% Mutation
for kidNumber=2:nParents
    nextKid=thisPopulation(parents(kidNumber),:);
    for t=1:5
        ch=rand;
        for i=1:size(changeOptions,1)
            if i==1
                if ch<=changeOptions(i,1)
                    chMeth=changeOptions(i,2);
                    break;
                end
            else
```

```
            if ch<=changeOptions(i,1) && ch>changeOptions(i-1,1)
                chMeth=changeOptions(i,2);
                break;
            end
        end
    end

    switch chMeth
    case 1 %% Mutation Storage und Merger (Losreihenfolge)
        if smallChange ==0
            %Displacement Mutation
            %Auswahl der zu verschiebenden Sektion (Storage)
            point1=0;
            point2=0;
            order=1:(vectorIndices(2)-1);
            for i=1:(vectorIndices(2)-1)
                k=vectorIndices(3);
                order(2,i)=nextKid(1,k+1)-nextKid(1,k);
                k=k+2;
            end
            while point1>=point2 || (point2-point1+1)==...
                (vectorIndices(2)-1)
                point1=randi(vectorIndices(2)-1);
                point2=randi(vectorIndices(2)-1);
            end
            shiftSection=nextKid(1,point1:point2);
            nextKid=[nextKid(1,1:(point1-1)) nextKid(1,(point2+1):...
            end)];
            shiftSectionOrder=order(:,point1:point2);
            order=[order(:,1:(point1-1)) order(:,(point2+1):...
            end)];
            % Auswahl des Einfuegepunkts
            newIndices=vectorIndices(2)-1-(point2-point1+1);
            point3=randi(newIndices);
            % Einfuegen des zu verschiebenden Anteils (Storage)
            nextKid=[nextKid(1,1:point3) shiftSection nextKid(1,...
            (point3+1):end)];
            order=[order(:,1:point3) shiftSectionOrder order(:,...
            (point3+1):end)];
            % Uebertragung auf den Merger
            point1m=vectorIndices(2)+point1-1;
```

```
        point2m=vectorIndices(2)+point2-1;
        shiftSection=nextKid(1,point1m:point2m);
        nextKid=[nextKid(1,1:(point1m-1)) nextKid(1,(point2m+1):...
        end)];
        point3m=vectorIndices(2)+point3-1;
        nextKid=[nextKid(1,1:point3m) shiftSection nextKid(1,...
        (point3m+1):end)];
    else
        %Auswahl der Swap-Gene
        point1=0;
        point2=0;
        while point1==point2
            point1=randi(vectorIndices(2)-1);
            point2=randi(vectorIndices(2)-1);
        end
        %Austausch der Gene (Storage)
        val1=nextKid(point1);
        val2=nextKid(point2);
        nextKid(point2)=val1;
        nextKid(point1)=val2;
        %Austausch der Gene (Merger)
        point1=vectorIndices(2)+point1-1;
        point2=vectorIndices(2)+point2-1;
        val1=nextKid(point1);
        val2=nextKid(point2);
        nextKid(point2)=val1;
        nextKid(point1)=val2;
    end
    % Ueberpruefen, ob sich Bearbeitungszeiten ueberlappen und
    % reparieren der Loesungen, falls notwendig
    storageTimes=nextKid(1,1:vectorIndices(2)-1);
    storageTimes(2,:)=storageTimes(1,:)+...
    storageOperatingTimes;
    for i=1:size(storageTimes,2)
        storageTimes(3,i)=i;
    end
    storageTimes=sortrows(transpose(storageTimes));
    for i=1:(size(storageTimes,1)-1)
        if storageTimes(i+1,1)<storageTimes(i,2)||storageTimes(i+1,1)==...
        storageTimes(i,1)
            opStart=storageTimes(i+1,1);
```

```
                startIndex=storageTimes(i+1,3);
                opTime=storageTimes(i+1,2)-storageTimes(i+1,1);
                % Verschieben
                storageTimes(i+1,1)=storageTimes(i,2);
                storageTimes(i+1,2)=storageTimes(i+1,1)+...
                opTime;
                % Uebertragen auf Loesung
                storageTemp=nextKid(1,1:vectorIndices(2)-1);
                storageTemp(1,startIndex)=...
                storageTimes(i+1,1);
                nextKid=[storageTemp nextKid(1,...
                vectorIndices(2):end)];
                % Uebertragung auf Merger
                nextKid(1,vectorIndices(2)+startIndex-1)=...
                nextKid(1,startIndex)+30;
            end
        end
        methodInd =1;

    case 2 %% Zufaellige Mutation von Storage- und Mergerpositionen
        %normalverteilt
        if smallChange==0
            mutatedPositions=randi(vectorIndices(2)-1);
        else
            mutatedPositions=1;
        end
        for k=1:mutatedPositions
            point=randi(vectorIndices(2)-1);
            mutatedPart=nextKid(1,point);
            change=random(pd)/100;
            mutatedPart=round(mutatedPart+change*mutatedPart);
            if mutatedPart <= 0
                mutatedPart=0;
            end
            if mutatedPart >= maxSteps
                mutatedPart=maxSteps;
            end
            difference=mutatedPart-nextKid(1,point);
            nextKid(1,point)=mutatedPart;
            nextKid(1,vectorIndices(2)+point-1)=nextKid(1,...
            vectorIndices(2)+point-1)+difference;
```

```
        end
        % Ueberpruefen, ob sich Bearbeitungszeiten ueberlappen und
        % reparieren der Loesungen, falls notwendig
        storageTimes=nextKid(1,1:vectorIndices(2)-1);
        storageTimes(2,:)=storageTimes(1,:)+...
        storageOperatingTimes;
        for i=1:size(storageTimes,2)
            storageTimes(3,i)=i;
        end
        storageTimes=sortrows(transpose(storageTimes));
        for i=1:(size(storageTimes,1)-1)
            if storageTimes(i+1,1)<storageTimes(i,2)||storageTimes(i+1,1)==...
            storageTimes(i,1)
                opStart=storageTimes(i+1,1);
                startIndex=storageTimes(i+1,3);
                opTime=storageTimes(i+1,2)-storageTimes(i+1,1);
                % Verschieben
                storageTimes(i+1,1)=storageTimes(i,2);
                storageTimes(i+1,2)=storageTimes(i+1,1)+...
                opTime;
                % Uebertragen auf Loesung
                storageTemp=nextKid(1,1:vectorIndices(2)-1);
                storageTemp(1,startIndex)=...
                storageTimes(i+1,1);
                nextKid=[storageTemp nextKid(1,...
                vectorIndices(2):end)];
                % Uebertragung auf Merger
                nextKid(1,vectorIndices(2)+startIndex-1)=...
                nextKid(1,startIndex)+30;
            end
        end
        methodInd=2;

    case 3 %% Zufaellige Mutation von Ofen- oder Frosterpositionen
        if randi(2)<2
            point = randi([vectorIndices(3) vectorIndices(4)-1],1,1);
            minOperaTime = ovenMin;
        else
            point = randi([vectorIndices(4) size(nextKid,2)-1],1,1);
            minOperaTime = freezerMin;
        end
```

```
        %normalverteilt
        mutatedPart = nextKid(point);
        change=abs(random(pd1)/100);
        if bitand(point,1)
            % End-Punkt
            dev_amount = nextKid(point) - nextKid(point-1);
            mutatedPart = round(mutatedPart - change*dev_amount);
            operaTime = mutatedPart - nextKid(point-1);
        else
            % Start-Punkt
            dev_amount = nextKid(point+1) - nextKid(point);
            mutatedPart = round(mutatedPart + change*dev_amount);
            operaTime= nextKid(point+1) - mutatedPart;
        end
        if mutatedPart > 0 && mutatedPart < maxSteps
            % Kriterium/Restriktion minTimeSpan erfuellt?
            if operaTime >= minOperaTime
                %mutieren
                nextKid(1,point) = mutatedPart;
            end
        end
        methodInd=3;

    case 4 %% Verschieben von Ofen- und Frosterbetriebszeiten
        if randi(2) < 2
            % oven tupel
            oven=true;
            x = randi([vectorIndices(3) vectorIndices(4)-1],1,1);
            else
            % freezer tupel
            oven=false;
            x = randi([vectorIndices(4) size(nextKid,2)-1],1,1);
        end
        mutatedPart = nextKid(x);
        % Start oder Ende?
        if bitand(x,1)
            start = false; % ID fuer Shiften
        else
            start = true; % ID fuer Shiften
        end
        randomNumber = randi(maxChange1)/100;
```

```
delta = 0; % delta zum vorherigen/naechsten Los
if(start==true)
    % suche End-Zeitpunkt vorheriges Los, falls moeglich
    lastEnd = nextKid(x-1);
    if(oven==true)
        %vorheriges Los auch ein Ofen-Tupel?
        if (x-1) <= vectorIndices(4)-1 && (x-1) >= vectorIndices(3)
            delta = mutatedPart - lastEnd;
        end
    else
        %vorheriges Los auch ein Freezer-Tupel?
        if (x-1) >= vectorIndices(4) && (x-1) <= size(nextKid,2)-1
            delta = mutatedPart - lastEnd;
        end
    end
    change = round(randomNumber*delta);
    % change nextKid(x) & nextKid(x+1)IF delta > 0
    if delta > 0
        nextKid(x)= nextKid(x)- change;
        nextKid(x+1)= nextKid(x+1) - change;
    end
else % x == end
    % suche Start-Zeitpunkt naechstes Los, falls moeglich
    nextStart = nextKid(x+1);
    if(oven==true)
        % naechstes Los auch ein Ofen-Tupel?
        if (x+1) <= vectorIndices(4)-1 && (x+1) >= vectorIndices(3)
            delta = nextStart - mutatedPart;
        end
    else % freezer==true
        % naechstes Los auch ein Freezer-Tupel?
        if (x+1) >= vectorIndices(4) && (x+1) <= size(nextKid,2)-1
            delta = nextStart - mutatedPart;
        end
    end
    change = round(randomNumber*delta);
    % change nextKid(x-1) & nextKid(x) OR delta = 0
    if delta > 0
        nextKid(x-1)= nextKid(x-1)+ change;
        nextKid(x)= nextKid(x)+ change;
    end
```

```
            end
            methodInd=4;
            end

            %% Ueberpruefen, ob Loesung neu ist und Nebenbedingungen entspricht
            newSol=1;
            for i=1:size(solVecs,1)
                solution = solVecs(i,:);
                if isequal(nextKid,solution)
                    newSol=0;
                    break;
                end
            end
            if newSol==1
                a=nebenbedKoefMatrix*transpose(nextKid);
                b=transpose(nebenbedSchranken);
                comp=a<=b;
                nbSum=sum(comp);
                if nbSum==size(nebenbedSchranken,2)
                    break;
                end
            end
    end
    % Speichern der erzeugten Loesung
    xoverKids(kidNumber,:)=nextKid;
    if size(changes,1)==0
        changes=methodInd;
    else
        changes=[changes methodInd];
    end
end
end
```

Literatur

[1] David Achermann. »Modelling, Simulation and Optimization of Maintance Strategies under Consideration of Logistic Processes«. Diss. ETH Zürich, 2008.

[2] Satyajith Amaran u. a. »Simulation optimization: a review of algorithms and applications«. In: *4OR* 12.4 (2014), S. 301–333.

[3] Lionel Amodeo, Christian Prins und David Ricardo Sanchez. »Comparison of metaheuristic approaches for multi-objective simulation-based optimization in supply chain inventory management«. In: *Workshops on Applications of Evolutionary Computation*. Springer. 2009, S. 798–807.

[4] S Andradottir. »A review of simulation optimization techniques«. In: *Proceedings of the 30th conference on Winter simulation*. IEEE Computer Society Press. 1998, S. 151–158.

[5] Jay April u. a. »Practical introduction to simulation optimization«. In: *Simulation Conference, 2003. Proceedings of the 2003 Winter*. Bd. 1. IEEE. 2003, S. 71–78.

[6] Jerry Banks u. a. *Discrete-Event System Simulation*. Prentice-Hall, 2001.

[7] Christian Blum und Andrea Roli. »Metaheuristics in combinatorial optimization: Overview and conceptual comparison«. In: *ACM Computing Surveys (CSUR)* 35.3 (2003), S. 268–308.

[8] Christian Blum u. a. »A brief survey on hybrid metaheuristics«. In: *Proceedings of BIOMA* (2010), S. 3–18.

[9] Christian Blum u. a. *Hybrid Metaheuristics - An Emerging Approach to Optimization*. Springer, 2008.

[10] Christian Blum u. a. »Hybrid metaheuristics in combinatorial optimization: A survey«. In: *Applied Soft Computing* 11.6 (2011), S. 4135–4151.

[11] Ilhem Boussaid, Julien Lepagnot und Patrick Siarry. »A survey on optimization metaheuristics«. In: *Information Sciences* 237 (2013), S. 82–117.

[12] Russ C Eberhart, James Kennedy u. a. »A new optimizer using particle swarm theory«. In: *Proceedings of the sixth international symposium on micro machine and human science*. Bd. 1. New York, NY. 1995, S. 39–43.

[13] Russell C Eberhart und Yuhui Shi. »Comparison between genetic algorithms and particle swarm optimization«. In: *International Conference on Evolutionary Programming*. Springer. 1998, S. 611–616.

[14] Andreas Fink und Stefan Voß. »Anwendung von Metaheuristiken zur Lösung betrieblicher Planungsprobleme«. In: *Wirtschaftsinformatik* 45.4 (2003), S. 395–407.

[15] Michael C Fu. »Optimization for simulation: Theory vs. practice«. In: *INFORMS Journal on Computing* 14.3 (2002), S. 192–215.

[16] Michael C Fu, Fred W Glover und Jay April. »Simulation optimization: a review, new developments, and applications«. In: *Proceedings of the 37th conference on Winter simulation*. Winter Simulation Conference. 2005, S. 83–95.

[17] Torsten Gellert, Wiebke Höhn und Rolf H Mohring. »Sequencing and scheduling for filling lines in dairy production«. In: *Optimization Letters* 5.3 (2011), S. 491–504.

[18] Michel Gendreau und Jean-Yves Potvin. *Handbook of metaheuristics*. Bd. 2. Springer, 2010.

[19] Michel Gendreau und Jean-Yves Potvin. »Metaheuristics in combinatorial optimization«. In: *Annals of Operations Research* 140.1 (2005), S. 189–213.

[20] Michel Gendreau und Jean-Yves Potvin. »Tabu Search«. In: *Handbook of Metaheuristics* 2nd Edition (2010), S. 41–60.

[21] Laszlo Gerencser, Stacy D Hill und Zsuzsanna Vago. »Optimization over discrete sets via SPSA«. In: *Proceedings of the 31st conference on Winter simulation: Simulation—a bridge to the future-Volume 1*. ACM. 1999, S. 466–470.

[22] Fred Glover, Manuel Laguna und Rafael Marti. »Fundamentals of scatter search and path relinking«. In: *Control and cybernetics* 29.3 (2000), S. 653–684.

[23] Ronald L Graham u. a. »Optimization and approximation in deterministic sequencing and scheduling: a survey«. In: *Annals of discrete mathematics* 5 (1979), S. 287–326.

[24] Pierre Hansen, Nenad Mladenovic und Jose A Moreno Perez. »Variable neighbourhood search: methods and applications«. In: *Annals of Operations Research* 175.1 (2010), S. 367–407.

[25] John H. Holland. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence*. MIT Press, 1975.

[26] Robert Hooke und Terry A Jeeves. »“Direct Search”Solution of Numerical and Statistical Problems«. In: *Journal of the ACM (JACM)* 8.2 (1961), S. 212–229.

[27] James Kennedy. »Particle swarm optimization«. In: *Encyclopedia of machine learning*. Springer, 2011, S. 760–766.

[28] Konrad Königsberger. *Analysis 1. Sechste Auflage*. 2004.

[29] Natalio Krasnogor und James Smith. »A tutorial for competent memetic algorithms: model, taxonomy, and design issues«. In: *IEEE Transactions on Evolutionary Computation* 9.5 (2005), S. 474–488.

[30] Lothar März-Wilfried Krug, Oliver Rose und Gerald Weigert. *Simulation und Optimierung in Produktion und Logistik*. 2011.

[31] Axel Kuhn und Markus Rabe. *Simulation in Produktion und Logistik: Fallbeispielsammlung*. Springer-Verlag, 2013.

[32] Pedro Larranaga u. a. »Genetic algorithms for the travelling salesman problem: A review of representations and operators«. In: *Artificial Intelligence Review* 13.2 (1999), S. 129–170.

[33] Averill M Law und W David Kelton. *Simulation modeling and analysis*. 2000.

[34] JP Lawrence und Kenneth Steiglitz. »Randomized pattern search«. In: *IEEE Transactions on Computers* 4.C-21 (1972), S. 382–385.

[35] Jan Karel Lenstra, AHG Rinnooy Kan und Peter Brucker. »Complexity of machine scheduling problems«. In: *Annals of discrete mathematics* 1 (1977), S. 343–362.

[36] Robert Michael Lewis und Virginia Torczon. »Pattern search algorithms for bound constrained minimization«. In: *SIAM Journal on Optimization* 9.4 (1999), S. 1082–1099.

[37] Chun Liu und Andreas Kroll. »Performance impact of mutation operators of a subpopulation-based genetic algorithm for multi-robot task allocation problems«. In: *SpringerPlus* 5.1 (2016).

[38] Jacques Loukil Teghem und Daniel Tuyttens. »Solving multi-objective production scheduling problems using metaheuristics«. In: *European journal of operational research* 161.1 (2005), S. 42–61.

[39] Taicir Loukil, Jacques Teghem und Philippe Fortemps. »A multi-objective production scheduling case study solved by simulated annealing«. In: *European journal of operational research* 179.3 (2007), S. 709–722.

[40] Rafael Marti, Manuel Laguna und Fred Glover. »Principles of scatter search«. In: *European Journal of Operational Research* 169.2 (2006), S. 359–372.

[41] Nicholas Metropolis u. a. »Equation of state calculations by fast computing machines«. In: *The journal of chemical physics* 21.6 (1953), S. 1087–1092.

[42] Alexander G Nikolaev und Sheldon H Jacobson. »Simulated Annealing«. In: *Handbook of Metaheuristics* 2nd Edition (2010), S. 1–40.

[43] Ibrahim H Osman und CN Potts. »Simulated annealing for permutation flow-shop scheduling«. In: *Omega* 17.6 (1989), S. 551–557.

[44] Riccardo Poli, James Kennedy und Tim Blackwell. »Particle swarm optimization«. In: *Swarm intelligence* 1.1 (2007), S. 33–57.

[45] Markus Rager. *Energieorientierte Produktionsplanung: Analyse, Konzeption und Umsetzung*. Gabler Edition Wissenschaft. Wiesbaden: Betriebswirtschaftlicher Verlag Dr. Th. Gabler / GWV Fachverlage GmbH Weisbaden, 2008.

[46] Günther R. Raidl, Jakob Puchinger und Christian Blum. »Metaheuristic Hybrids«. In: *Handbook of Metaheuristics* 2nd Edition (2010), S. 469–496.

[47] Colin R. Reeves. »Genetic Algorithms«. In: *Handbook of Metaheuristics* 2nd Edition (2010), S. 109–140.

[48] Mauricio G.C. Resende u. a. »Scatter Search and Path-Relinking: Fundamentals, Advances, and Applications«. In: *Handbook of Metaheuristics* 2nd Edition (2010), S. 87–108.

[49] Ruben Ruiz und Jose Antonio Vazquez-Rodriguez. »The hybrid flow shop scheduling problem«. In: *European Journal of Operational Research* 205.1 (2010), S. 1–18.

[50] Thomas Sobottka, Felix Kamhuber und Wilfried Sihn. »Increasing energy efficiency in production environments through an optimized, hybrid simulation-based planning of production and its periphery«. In: *The 24th CIRP Conference on Life Cycle Engineering* (2017).

[51] James C Spall. »Multivariate stochastic approximation using a simultaneous perturbation gradient approximation«. In: *IEEE transactions on automatic control* 37.3 (1992), S. 332–341.

[52] Thomas Stützle. »Local search algorithms for combinatorial problems«. In: *Darmstadt University of Technology PhD Thesis* 20 (1998).

[53] EYLEM TEKIN und IHSAN SABUNCUOGLU. »Simulation optimization: A comprehensive review on theory and applications«. In: *IIE Transactions* 36.11 (2004), S. 1067–1081.

[54] Stefan Voss u. a. *Meta-heuristics: Advances and trends in local search paradigms for optimization*. Springer Science und Business Media, 2012.

[55] Ingo Wegener. *Komplexitätstheorie: Grenzen der Effizienz von Algorithmen*. Springer-Verlag, 2013.

[56] Karsten Weicker. *Evolutionäre Algorithmen*. Springer-Verlag, 2015.

[57] Sigrid Wenzel u. a. *Qualitätskriterien für die Simulation in Produktion und Logistik: Planung und Durchführung von Simulationsstudien*. Springer-Verlag, 2007.

[58] Darrell Whitley. »A genetic algorithm tutorial«. In: *Statistics and Computing* 4.2 (1994).

[59] David H Wolpert und William G Macready. »No free lunch theorems for optimization«. In: *IEEE transactions on evolutionary computation* 1.1 (1997), S. 67–82.

[60] Jie Xu, Barry L Nelson und JEFF Hong. »Industrial strength COMPASS: A comprehensive algorithm and software for optimization via simulation«. In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 20.1 (2010), S. 3.

Abbildungsverzeichnis

Tabellenverzeichnis