TECHNISCHE UNIVERSITÄT WIEN
Vienna | Austria

Dominik Putz, BSc

Kurzfristige Einsatzoptimierung eines thermischen Kraftwerksblocks

Diplomarbeit

Technische Universität Wien

Institut für Energiesysteme und Elektrische Antriebe
Vorstand: Schrödl Manfred O.Univ.Prof. Dipl.-Ing. Dr.techn.

Betreuung: Privatdoz. Dipl.-Ing. Dr.techn. Johann Auer,
Projektass. Mag.rer.nat. Daniel Schwabeneder

Wien, Januar 2019

Danksagung

Ich möchte mich an dieser Stelle bei einigen Personen bedanken, die mich bei der Entwicklung und Umsetzung dieser Arbeit unterstützt haben. Ein Dankeschön für die tolle Zusammenarbeit mit der EVN und ihren stets freundlichen Mitarbeitern. Dabei möchte ich Robert Halbweis hervorheben, der mir immer hilfreiches Feedback zu meinem aktuellen Arbeitsfortschritt gegeben hat. Nicht unerwähnt möchte ich auch meinen Betreuer Prof. Auer lassen, der stets bemüht ist meinen Ehrgeiz zu fördern. Bei meiner Familie sowie Partnerin Nadine möchte ich mich für die tolle Unterstützung bedanken.

Kurzfassung

Durch den zunehmenden Ausbau von erneuerbaren Energieträgern steigt die Volatilität der Erzeugung in der Energieversorgung. Von grundlegender Bedeutung für die Versorgungssicherheit ist das Gleichgewicht zwischen Nachfrage und Erzeugung zu jedem Zeitpunkt. Die dafür verwendeten thermischen Kraftwerksblöcke unterliegen technischen Einschränkungen, die eingehalten werden müssen, um einen reibungslosen und unfallfreien Betrieb zu gewährleisten. Die Energieversorgungsunternehmen nutzen auf Basis von technischen und ökonomischen Daten dafür entwickelte Softwarelösungen, um für die thermischen Kraftwerksblöcke optimale Fahrpläne für die elektrischen Abrufleistungen zu generieren. Der Schwerpunkt dieser Applikationen liegt in der Optimierung. Aufgrund von größer werdenden Anforderungen und dem steigenden Grad an Komplexität der zu lösenden Modelle, ist es für die Softwareentwicklung eine größer werdende Herausforderung optimale Fahrpläne zu ermitteln. Ziel der Arbeit ist, anhand einer zu programmierenden Applikation die Vorteile der noch jungen Programmiersprache Julia auszuloten. Der Schwerpunkt der Analyse soll dabei auf der Optimierungsdauer, dem Speicherbedarf und der Bedienbarkeit liegen und mit der bereits bestehenden Applikation, die die EVN aktuell verwendet, verglichen werden.

Abstract

The increasing expansion of renewable energy sources increases the volatility of energy generation. Of fundamental importance for security of supply is the balance between demand and generation at all times. The thermal power plant units which are used for this purpose are subject to technical restrictions that must be observed in order to ensure smooth and accident-free operation. The energy supply companies use software solutions developed for this purpose on the basis of technical and economic data in order to generate optimal timetables for the electric power plant blocks for the electrical retrieval services. The focus of these applications is optimization. Due to increasing demands and the increasing degree of complexity of the models to be solved, it is a growing challenge for software development to determine optimal timetables. The aim of the work is to explore the advantages of the still young programming language Julia by means of an application to be programmed. The focus of the analysis will be on the duration of the optimization, the storage requirements and the operability and will be compared with the existing application which is currently used by EVN.

Inhaltsverzeichnis

1 Einleitung

Im täglichen Leben ist Energie in Form von Elektrizität und Wärme unverzichtbar. Dabei nehmen deren Nutzung aufgrund des Verhaltens der Bevölkerung und die Stillung deren Bedürfnisse stetig zu. Die Energieversorgung spielt dabei eine essentielle Rolle. Genauer gesagt ist es Aufgabe der Energiewirtschaft die Versorgung von Haushalten mit Energie sicherzustellen. Die sogenannten Energieversorgungsunternehmen (EVU) bilden das Rückgrat der Energiewirtschaft. Sie betreiben meist mehrere Kraftwerksparks für die Erzeugung von elektrischer und thermischer Energie, die für die Versorgung von Haushalten und Betrieben genutzt werden.

Durch den zunehmenden Ausbau von erneuerbaren Energieträgern steigt die Volatilität der Erzeugung. Die regenerativen Energieträger besitzen im Gegensatz zu fossilen Quellen keine zeitliche Unbeschränktheit. Diese Aspekte erschweren die Versorgung und vor allem deren Planung, welche Aufgabe der EVUs ist. In der Energieversorgung ist es von Bedeutung, dass die Nachfrage gleich der Erzeugung entspricht. Da der Bedarf an Energie grundsätzlich festgelegt und im Allgemeinen auch prognostizierbar ist, muss die Erzeugung als Variable angesehen werden. Dabei werden regenerative Energieerzeuger bevorzugt behandelt und bei einem etwaigen Erzeugungsüberschuss die Erzeugung von thermischen Kraftwerken gezielt reduziert. Die thermischen Kraftwerke unterliegen technischen Einschränkungen, die eingehalten werden müssen, um einen reibungslosen und unfallfreien Betrieb zu gewährleisten. Aus dieser Sachlage wird ersichtlich, dass zu bestimmten Zeiten und Umwelteinflüssen die Steuerung und Regelung von thermischen Kraftwerken eine große Herausforderung für die EVUs ist. Es steht außer Zweifel, dass die EVUs ihre Kraftwerke profitmaximierend steuern müssen, um nicht in wirtschaftliche Misslage zu geraten. Die Portfolio-Optimierung hat als Aufgabe den optimalen Kraftwerkseinsatz zur jeweiligen Marktsituation zu gewährleisten. Die EVUs nutzen dafür entwickelte und angepasste Softwarelösungen mit Schwerpunkt Optimierung. Durch die steigenden Anforderungen und den Grad an Komplexität ist es für die Softwareentwicklung eine größer werdende Herausforderung optimale Lösungen zu bieten. Die EVN AG nutzt zurzeit (Stand 2018) eine eigens entwickelte Applikation in Java. Um auch in Zukunft eine geeignete Portfolio-Optimierung betreiben zu können, muss die verwendete Applikation stets erweitert und adaptiert werden. Dadurch bietet sich die Möglichkeit nach Alternativen zu untersuchen, in Form einer anderen Programmiersprache, an. Eine junge Programmiersprache Julia, die am MIT entwickelt wurde, ist spezialisiert auf numerische Lösungen. Die Verbesserung von Optimierungsmodellen ist Forschungsgegenstand in verschiedenen Wissenschaften und dadurch ergibt sich die Aktualität des Themas.

Gegenstand dieser Arbeit ist, anhand einer zu programmierenden Applikation die Vorteile der noch jungen Programmiersprache Julia auszuloten. Im Mittelpunkt dieser Arbeit

steht die Frage, ob Julia im Vergleich zu anderen Programmiersprachen, inbesondere Java, eine einsatzstarke Alternative darstellt. Der Schwerpunkt der Analyse soll dabei bei der Optimierungsdauer, dem Arbeitsspeicherbedarf, der Implementierbarkeit der Applikation liegen und mit anderen Lösungen verglichen werden. Dabei wird als weitere mögliche Softwarelösung die Applikation in MatLab mit unterschiedlichem Lösungsansatz entwickelt und in den Vergleich eingebunden.

Im ersten Teil der Arbeit wird das Kapitel State of the Art erläutern, wie die Optimierung bei einem Energieversorger aktuell abläuft. Die verwendete Programmiersprache Julia wird genauer erklärt und die Vorteile dem Leser verdeutlicht. Im Kapitel Methode wird der zu optimierende thermische Kraftwerksblock näher beschrieben, wobei das sogenannte *Unit Commitment Model* grob erklärt wird, auf deren Basis die Applikation entwickelt wurde. Unter anderem werden die Rahmenbedingen und Einschränkungen festgelegt, denen das Modell unterliegt. In weiterer Folge wird ein Überblick über die verwendeten Optimierungsverfahren gemacht. Im Zentrum des Kapitels stehen der Lösungsansatz, das Verfahren und die mathematische Beschreibung in MatLab und in Julia. Aufbauend werden die Ergebnisse dargestellt und verständnisvoll erklärt. Der Abschnitt Synthese enthält eine umfangreiche Analyse und Vergleich der Ergebnisse. Dabei werden die Ergebnisse aus verschiedenen Perspektiven betrachtet und objektiv hinsichtlich ihrer Qualität bewertet. Auftretende Fragen für weitere Untersuchungen werden im abschließenden Kapitel thematisiert.

2 State Of The Art

Die Ermittlung eines optimalen Fahrplans für die elektrische Abrufleistung eines oder mehrerer Kraftwerksblöcke basiert auf gemischt-ganzzahliger linearer Programmierung. Dieses Verfahren wird in den folgenden Abschnitten ausführlich behandelt, erläutert und wurde als Grundlage für einen Lösungsansatz gewählt. Man unterscheidet grundsätzlich zwischen mittel- und kurzfristiger Einsatzoptimierung. Ziel der Arbeit ist die genaue Betrachtung der Optimierungsmöglichkeiten bei kurzfristiger Einsatzoptimierung, sprich Day-Ahead und Intra-Day Fahrplänen. Als Standard in der Energiewirtschaft wird eine 7-Tage Prognose im 15-Minuten Raster mittels gemischt-ganzzahliger linearer Programmierung vorgenommen. Abhängig von der Datenlage kann die Berechnung Sekunden bis mehrere Minuten dauern. Dieser Gesichtspunkt sollte in den weiteren Untersuchungen berücksichtigt werden. Oftmals werden Vereinfachungen und Linearisierungen vorgenommen, um die Abarbeitung schneller zu gestalten.

Lange Berechnungszeiten entstehen bei Zeiträumen von über 28 Tagen bzw. kleiner Zeitgranularität von beispielsweise 15 Minuten. Ein Kraftwerkseinsatzpark mit vielen verschiedenen Erzeugern, die große Volatilitäten aufweisen, hat sich als ein Modell erwiesen, das häufig mit sehr langen Simulationsdauern für die Lösung auskommen muss. Deshalb führt hier ein Gedanke auf ein alternatives Lösungsverfahren. Die Methode der dynamischen Programmierung wurde gewählt, da diese ein vielversprechendes Lösungsverfahren darstellt. Dieses Verfahren ist im Allgemeinen schon seit langem bekannt. Eine detaillierte Beschreibung dieses Verfahrens findet im Kapitel Methode ausführlich statt.

Die dynamische Programmierung kann auf verschiedene Arten implementiert werden. Die EVN nutzt zurzeit eine Applikation, die in Java entwickelt wurde. Durch den stetig wachsenden Grad an Komplexität der zu lösenden Modelle stellt sich die Frage, ob alternative Programmiersprachen für die Implementierung der dynamischen Programmierung geeigneter wären. Als zu untersuchende Sprache wird Julia gewählt, da sie im Vergleich zu anderen Programmiersprachen im Hinblick auf Performance bei der Lösung von Optimierungsproblemen großes Potential aufweist [1].

Julia wurde speziell für High Performance Anwendungen konzipiert, realisiert durch eine modulare Compiler-Unterbau-Architektur, kurz LLVM[1]. Für nähere Information wird auf einschlägige Fachliteratur verwiesen. Julia ist dynamisch typisiert[2], fühlt sich wie eine Skriptsprache an und unterstützt interaktive Anwendungen. Sie verwendet Multiple Dispatch als Paradigma, wodurch es leicht ist, viele objektorientierte und

[1] Low Level Virtual Machine

[2] Zur Laufzeit eines Programms wird erst der Datentyp einer Variable festgelegt im Gegensatz zur statischen Typisierung.

funktionale Programmiermuster auszudrücken. Die Standardbibliothek bietet asynchrone E/A, Prozesssteuerung, Protokollierung, Profilerstellung und einen Paketmanager. Julia zeichnet sich durch numerisches Rechnen aus. Die Syntax ist spezialisiert für Mathematik, numerische Datentypen und Parallelität. Julias Multiple Dispatch[3] eignet sich für die Definition zahl- und arrayähnlicher Datentypen [2].

[3]Die Auswahl einer Methode ist nicht alleine abhängig vom Typ eines Objekts, sondern vom Typ mehrerer Objekte.

3 Beschreibung des Modells

Dieses Kapitel widmet sich der Beschreibung des entwickelten Modells. Darunter fallen allgemeine Eigenschaften, die das Modell beschreiben und welche Rahmenbedingungen bzw. Restriktionen es zu erfüllen hat. Die wesentlichen Merkmale werden hervorgehoben und für das Verständnis kurz erläutert. Vor allem werden die Vorgaben, welche der Kraftwerksblock zu erfüllen hat exakt beschrieben. Dabei beschränkt sich die mathematische Beschreibung nur auf grundlegende Definitionen und erst in den nachfolgenden Kapiteln wird im Detail darauf eingegangen.

3.1 Beschreibung des Kraftwerksblocks

Bei dem zu modellierenden thermischen Kraftwerksblock handelt es sich um einen fiktiven Gas- und Dampf-Kombikraftwerksblock (GuD-Kraftwerk). Ein GuD-Kraftwerk kombiniert eine Gasturbine mit einem Dampfkraftwerk zur Energieerzeugung. Die Abwärme, die aus der Gasturbine entsteht, wird über einen Abhitzekessel bzw. Dampferzeuger in einer Dampfturbine weiter genutzt. Bei näherer Betrachtung des thermodynamischen Kreisprozesses kommt es durch die Kombination beider Abläufe zu einer Wirkungsgraderhöhung. Moderne Kombikraftwerke gehören zu den effizientesten konventionellen Kraftwerken mit Wirkungsgraden bis zu 63 % [3]. Durch Fernwärmeauskopplung lässt sich dieser noch weiter steigern. Ein GuD-Kraftwerk ist aufgrund seiner technischen Beschaffenheit sehr flexibel einsetzbar und kann hervorragend dafür verwendet werden Mittel- sowie Spitzenlast abzudecken.

Der Zustand des thermischen Kraftwerksblocks wird vom aktuellen Marktpreis bestimmt. Der Kraftwerksblock ist ein sogenannter *Price-Taker*. Sein Betriebszustand beeinflusst somit nicht den Strompreis im Gegensatz zu einem *Market-Maker*. Dieser Umstand ist von grundlegender Bedeutung, da der Marktpreis unabhängig von der elektrischen Abrufleistung betrachtet werden darf. Andernfalls würde das Modell weitaus komplexer ausfallen.

Der Fahrplan des thermischen Blocks wird anhand einer Software Applikation erstellt. Die Applikation errechnet den Abrufplan in Abhängigkeit des Strompreises. Die Daten für den Strompreis werden von der österreichischen Strombörse EXAA (Energy Exchange Austria), welche ein Marktplatz für den Energiehandel ist, bezogen.

3.2 Unit Commitment Model

Durch die vorhin beschriebenen Gesichtspunkte ist klar ersichtlich, dass die Steuerung von thermischen Kraftwerken zu bestimmten Zeiten sehr flexibel abzulaufen hat und dies von entsprechenden Softwareanwendungen durchgeführt werden muss, um einen optimalen Fahrplan zu gewährleisten. Durch fehlende Speicherkapazitäten bzw. Technologie muss die Produktion zu jedem Zeitpunkt gleich dem Verbrauch sein. Ein Modell, das sich als Beschreibung bewährt hat, ist das sogenannte *Unit Commitment Model*.

Das *Unit Commitment Model* wird verwendet, um Abrufleistungen von Kraftwerke, speziell im Day-Ahead Fahrplan, zu berechnen. Es wird von EVUs genutzt und hat als Aufgabe effiziente Fahrpläne für fossile und erneuerbare Kraftwerke unter verschiedenen technischen und wirtschaftlichen Restriktionen zu erstellen. Der Fokus des *Unit Commitment Models* ist die Kosten der Produktion zu minimieren bzw. den Ertrag der Produktion zu maximieren [4] unter Einhaltung spezieller Einschränkungen[1].

Die Rahmenbedingungen denen das *Unit Commitment Model* unterliegt sind unter anderem große, komplexe Probleme durch verschiedene Typen von Kraftwerken, eine hohe Anzahl an Energieerzeugern und geografische Einflussfaktoren anhand bestehender Netzinfrastruktur. Gewiss besitzt jede Einheit andere technische und kaufmännische Eigenschaften, welche die Komplexität des Problems noch weiter steigern. Zusätzlich enthält das *Unit Commitment Model* weitere Elemente, wie Prognoseunsicherheiten speziell im Hinblick auf Windenergie und Photovoltaik, Reserven, die zur Verfügung stehen müssen und politische Einflussfaktoren, wie zum Beispiel das EU ETS (European Union Emissions Trading System), welche die Dauer für die Optimierung weiter erhöhen.

Daraus lässt sich ableiten, dass die Lösung des *Unit Commitment Models* relativ viel Zeit in Anspruch nehmen kann. Für EVUs ist von Bedeutung, dass die Erstellung der Fahrpläne so wenig Zeit wie nur notwendig beansprucht. Um die Dauer der Optimierung gering zu halten, werden in erster Linie Vereinfachungen im Modell vorgenommen, welche die Komplexität stark reduzieren sowie auch immer leistungsfähigere Software- und Hardwarelösungen genutzt werden. Die Einschränkungen die das entwickelte Modell zu erfüllen hat, werden im nächsten Abschnitt erläutert.

3.3 Rahmenbedingungen des Models

Das Modell hat vorgegebenen Einschränkungen zu genügen, damit der berechnete Abrufplan auch tatsächlich umsetzbar und in der Realität fahrbar ist. Im Generellen kann man die Rahmenbedingungen in allgemeine, technische und kaufmännische Parameter unterteilen.

[1] Die Erzeugung muss zu jedem Zeitpunkt die Nachfrage decken.

3.3.1 Allgemeine Parameter

Eine Zusammenfassung der Allgemeinen Parameter ist Tabelle 3.1 zu entnehmen.

Tabelle 3.1: Allgemeine Parameter des Kraftwerksblocks.

Parameter	Einheit	Wert
Zeitschrittintervall	h	0,25
Strompreisfaktor	%	125
Zeit Beginn	1	1
Zeit Ende	1	35 040
Zustand Beginn	1	1

Das Zeitschrittintervall entspricht einer Auflösung von 15 Minuten. Durch die gewählte feine Granularität ergibt sich eine relativ hohe Schrittanzahl, welche für eine Woche bereits 672 Schritte und einem Jahr 35 040 Schritte entspricht. Durch die große Anzahl an Zeitschritten steigert sich die Komplexität des Problems, was direkte Auswirkungen auf die Berechnungsdauer hat. Dieser Umstand wird im Zuge der Arbeit noch häufig aufgegriffen und behandelt werden.

Der Strompreisfaktor ist variabel und dient in erster Linie für Sensitivitätsanalysen bei der Optimierung. Im Standardfall wird er mit 125 % oder 100 % festgelegt. Durch die maximal festgelegte Optimierungsdauer von einem Jahr ergibt sich somit eine maximale Schrittanzahl von 35 040 . Im Standardfall werden rollierende 7-Tage bis 28-Tage Optimierungen vorgenommen. Der Beginn-Zustand ist variabel und wird im Standardfall als 1 angenommen, was bedeutet, dass der Kraftwerksblock still steht. Dieser Parameter kann geändert werden[2], um ein bereits hochgefahrenes bzw. hoch- sowie abfahrendes Kraftwerk darzustellen.

3.3.2 Technische Parameter

Die technischen Parameter für den thermischen Kraftwerksblock sind in Tabelle 3.2 ersichtlich.

Die installierte Netto-Leistung ist für jeden Zeitschritt als variabel anzusehen. Im Regelbetrieb hat dieser standardmäßig den gegebenen Wert. Für bestimmte Situationen während dem Betrieb, kann es erforderlich sein die maximal technisch erlaubte Leistung geringer als die installierte Netto-Leistung festzulegen. Dies ist im Modell allenfalls zu berücksichtigen. Dasselbe gilt auch für die technische Mindestleistung, die unter Umständen erhöht werden kann. Dabei ist zu erwähnen, dass der Abrufplan die gegebenen Leistungsgrenzen in jedem Fall einhält, um einen reibungslosen Betrieb zu gewährleisten. Der Abrufplan muss den festgelegten Leistungsgrenzen entsprechend rechtzeitig angepasst werden, unter Einhaltung des maximal erlaubten Leistungsänderungsgradienten. Die minimale Stillstandsdauer beschreibt jenes Zeitintervall,

[2] Änderung nur innerhalb der zulässigen Grenzen erlaubt.

Tabelle 3.2: Technische Parameter des Kraftwerksblocks.

Parameter	Einheit	Wert
Installierte Netto-Leistung	MW	420
Technische Mindestleistung	MW	250
Minimale Stillstandsdauer	h	4
Minimale Betriebsdauer	h	6
Maximaler Leistungsänderungsgradient	MW/15 Min	45
Inverse Produktionsfunktion a	MW th.	194,4
Inverse Produktionsfunktion b	MW th./MW	1,027
Inverse Produktionsfunktion c	MW th./MW²	0,000 782
Heißstart ab Stillstandsdauer von x Stunden	h	4
Warmstart ab Stillstandsdauer von x Stunden	h	12
Kaltstart ab Sillstandsdauer von x Stunden	h	48

das nach dem Herunterfahren des Kraftwerksblocks eingehalten werden muss, bevor dieser wieder hochgefahren werden darf. Ebenso legt die minimale Betriebsdauer jenes Zeitintervall fest, dass der Kraftwerksblock nach Erreichen der technischen Mindestleistung einzuhalten hat, bevor dieser wieder heruntergefahren werden darf. Der maximale Leistungsänderungsgradient gibt den maximal erlaubten Leistungssprung zwischen zwei aufeinanderfolgenden Zeitintervallen vor. Der maximale Leistungsänderungsgradient ist unter allen Umständen einzuhalten.[3]

3.3.3 Wirkungsgrad

Der Wirkungsgrad des Kraftwerksblocks ist nicht konstant und abhängig von der elektrischen Abrufleistung. Die quadratische inverse Produktionsfunktion nach Gl. (3.1) veranschaulicht dieses Verhalten und stellt die Beziehung zwischen elektrischer Abrufleistung und Brennstoffwärmeleistung dar.

$$P_{thermisch} = a + b \cdot P + c \cdot P^2 \tag{3.1}$$

Tabelle 3.3 enthält markante Punkte der elektrischen Abrufleistung, Brennstoffwärmeleistung und dem zugehörigen Wirkungsgrad.

Abbildung 3.1 veranschaulicht das Verhalten des Wirkungsgrads in Abhängigkeit der Abrufleistung.

In Abbildung 3.1 ist zu sehen, dass die Beziehung zwischen elektrischer Abrufleistung und thermischer Brennstoffwärmeleistung annähernd linear ist. Durch lineare Approximierung kann somit die quadratische inverse Produktionsfunktion zu einer linearen Funktion vereinfacht werden. Mittels Taylor Reihenentwicklung [5] gemäß Gl. (3.2) um den Schnittpunkt 340 MW lässt sich die quadratische Funktion überführen auf Gl. (3.3).

[3] Eine Außnahme wäre ein Hoch- bzw. Herunterfahrprozess in Form eines Rampenvorgangs.

Tabelle 3.3: Der Wirkungsgrad verbindet die elektrische Abrufleistung mit der dafür notwendigen thermischen Brennstoffwärmeleistung. Der Wirkungsgrad ist i.A. nicht konstant, sondern abhängig von der elektrischen Abrufleistung. Der Zusammenhang wird durch die quadratisch inverse Produktionsfunktion beschrieben.

Abrufleistung in MW	Brennstoffwärmeleistung in MW thermisch	Wirkungsgrad in %
250	500	50
335	626	53,5
420	764	55

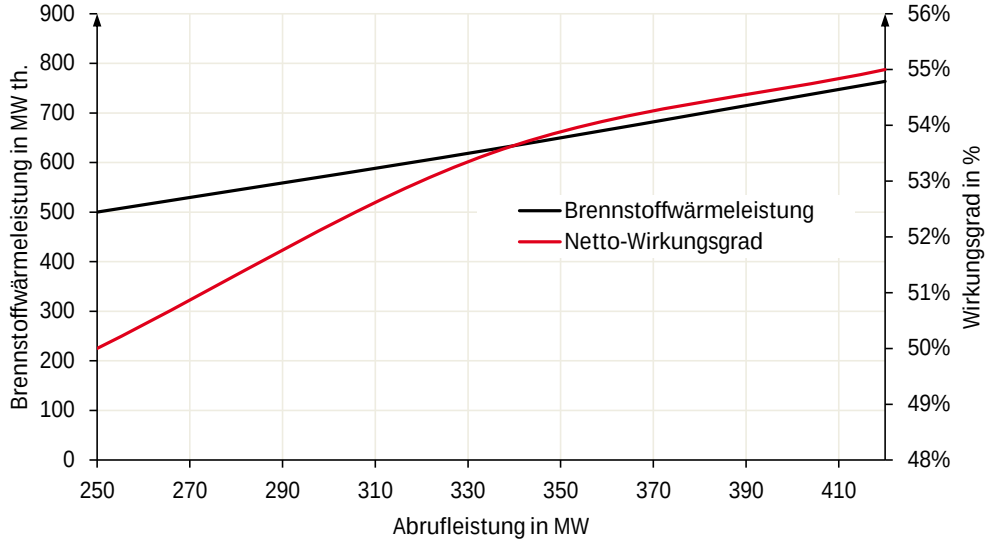

Abbildung 3.1: Thermische Brennstoffwärmeleistung und Wirkungsgrad in Abhängigkeit der elektrischen Abrufleistung.

$$f(x;a) = f(a) + f'(a) \cdot (x - a) \tag{3.2}$$

$$P_{thermisch} = a + b \cdot P + c \cdot P^2 \approx a_{lin} + b_{lin} \cdot P \tag{3.3}$$

Die Arbeit bedient sich der Methode der linearen Approximierung, um die Komplexität des Problems gering zu halten. Die verwendeten Parameter für die linearisierte Produktionsfunktion sind in Tabelle 3.4 ersichtlich.

Tabelle 3.4: Kennwerte der Taylor-Reihenentwicklung.

Parameter	Einheit	Wert
Inverse Produktionsfunktion a	MW th.	110,4
Inverse Produktionsfunktion b	MW th./MW	1,551

3.3.4 Ökonomische Parameter

Die ökonomischen Parameter sind Tabelle 3.5 zu entnehmen.

Tabelle 3.5: Ökonomische Rahmenbedingungen für den Betrieb des Kraftwerksblocks.

Parameter	Einheit	Wert
CO2 Emissionsfaktor Brennstoff	t/MWh th.	0,2
Startkosten heiß	EUR	30 000
Startkosten warm	EUR	40 000
Startkosten kalt	EUR	50 000
Stundenabhängige Betriebskosten	EUR/h	200
Strompreis	EUR/MWh	variabel
Brennstoffpreis	EUR/MWh	variabel
CO2 Preis	EUR/t	variabel
Elektrische Zusatzkosten	EUR/MWh	variabel

Als Datenquelle für den Strompreis werden für das Jahr 2017 die Werte aus dem EXAA 15 Minuten Produkt herangezogen. Diese werden für jede Stunde gemittelt, um die Volatilität innerhalb einer Stunde zu reduzieren. Für den Brennstoffpreis wird eine eigene Variable, *Brennstoffpreis all-in*, nach Gl. (3.4) eingeführt, die gleichzeitig die Emissionspreise für CO_2 miteinbezieht.

$$p_{fuelallin,t} = p_{fuel,t} + f_{CO2} \cdot p_{CO2,t} \tag{3.4}$$

Die Startkosten werden je nach Starttyp in den jeweiligen Zeitschritten als Zusatzkosten herangezogen. Somit ergeben sich die Kosten zu jedem Zeitpunkt t gemäß Gl. (3.5).

$$C_t = p_{fuelallin,t} \cdot P_{thermisch,t} + p_{O\&M,t} + p_{additional,t} \cdot P_t + C_{Start,t} \tag{3.5}$$

Es sei an dieser Stelle hervorgehoben, dass die Preise auf die entsprechend verwendete Zeitbasis von 15 Minuten gerechnet werden.

3.3.5 Rampenverlauf

Ein Kraftwerk muss aus dem Stillstand mit einer vorher festgelegten Rampe hochgefahren werden, um in den Online Status[4] zu gelangen bzw. die technische Mindestleistung erreichen zu können. Sobald der Startvorgang des Kraftwerks abgeschlossen ist, darf die Leistung innerhalb der technisch erlaubten Grenzen über den maximalen Leistungsänderungsgradienten variiert werden. Im regulären Betrieb erstreckt sich dieses Leistungsband zwischen technischer Mindest- und Maximalleistung. Je nach Art des Starts sind auch entsprechende Kosten zu kalkulieren. Für das Herunterfahren des

[4]Der Online Status entspricht jenem Zustand, bei dem die elektrische Abrufleistung größer gleich der technischen Mindestleistung ist und kein Rampenvorgang aktiv ist.

Kraftwerks wird eine Abfahrrampe definiert, die mit keinen Kosten in Verbindung gebracht wird. Für den Kraftwerksblock sind drei verschiedene Aufwärtsrampen und eine Abwärtsrampe vorgesehen. Dabei werden die Rampen in einen

- Heißstart
- Warmstart
- Kaltstart
- Abfahrt

gegliedert. Abbildung 3.2 stellt die Rampenvorgänge und ihren zeitlichen Ablauf grafisch dar.

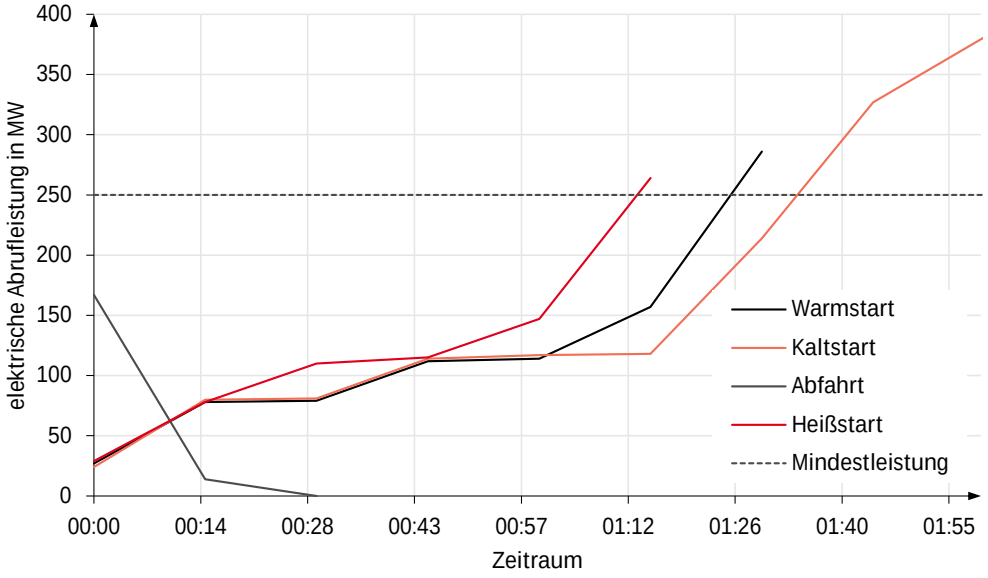

Abbildung 3.2: Verlauf der Startrampen Heißstart, Warmstart, Kaltstart vom Stillstand beginnend und der Abfahrrampe von einem beliebigen Betriebszustand ausgehend.

Die entsprechenden Werte für Abrufleistung und Brennstoffwärmeleistung für den jeweiligen Rampentyp sind in Tabelle 3.6 bis Tabelle 3.9 dargestellt.

Tabelle 3.6: Leistungswerte eines Heißstartverlaufs vom Stillstand beginnend.

Heißstart			
Beginn hh:mm	Ende hh:mm	Abrufleistung in MW	Brennstoffwärmeleistung in MW th.
00:00	00:15	29	162
00:15	00:30	78	297
00:30	00:45	110	400
00:45	01:00	115	453
01:00	01:15	147	461
01:15	01:30	264	559

Tabelle 3.7: Leistungswerte eines Warmstartverlaufs vom Stillstand beginnend.

Warmstart			
Beginn hh:mm	Ende hh:mm	Abrufleistung in MW	Brennstoffwärmeleistung in MW th.
00:00	00:15	27	148
00:15	00:30	78	284
00:30	00:45	79	332
00:45	01:00	112	410
01:00	01:15	114	427
01:15	01:30	157	451
01:30	01:45	286	593

Tabelle 3.8: Leistungswerte eines Kaltstartverlaufs vom Stillstand beginnend.

Kaltstart			
Beginn hh:mm	Ende hh:mm	Abrufleistung in MW	Brennstoffwärmeleistung in MW th.
00:00	00:15	24	134
00:15	00:30	80	303
00:30	00:45	81	339
00:45	01:00	114	415
01:00	01:15	117	424
01:15	01:30	118	426
01:30	01:45	214	503
01:45	02:00	327	657
02:00	02:15	381	710

Tabelle 3.9: Leistungswerte bei Abfahrt des Kraftwerksblocks in den Stillstand.

Abfahrt			
Beginn hh:mm	Ende hh:mm	Abrufleistung in MW	Brennstoffwärmeleistung in MW th.
00:00	00:15	167	366
00:15	00:30	14	88
00:30	00:45	0	0

An dieser Stelle sei erwähnt, dass die Brennstoffwärmeleistung bei einem Rampenverlauf nicht über die quadratisch inverse Produktionsfunktion berechnet wird, sondern definierte Leistungswerte besitzt.

3.3.6 Zielfunktion

Das Ziel des Modells ist die Maximierung des Ertrags nach Gl. (3.6). Der Ertrag entspricht der Summe des Deckungsbeitrags zu jedem Zeitschritt. Der Deckungsbeitrag setzt sich aus der Differenz zwischen dem Profit durch die Stromvermarktung, dem Zukauf von Brennstoff und sonstigen Kosten zusammen.

$$\Pi = \sum_{t=1}^{T} p_{market,t} \cdot P_t - C_t \tag{3.6}$$

Der Optimierungszeitraum T ist nach Bedarf festlegbar. Die verwendeten Zeiträume und deren zugehörige Schrittanzahl sind Tabelle 3.10 zu entnehmen.

Tabelle 3.10: Verwendete Zeiträume und die damit resultierende Anzahl an notwendigen Schritten für den jeweiligen Zeitraum.

Zeitraum	Schritte	Monate
1 Tag	96	0,036
1 Woche	672	0,25
2 Wochen	1344	0,5
3 Wochen	2016	0,75
1 Monat	2688	1
5 Wochen	3360	1,25
6 Wochen	4032	1,5
7 Wochen	4704	1,75
2 Monate	5376	2
3 Monate	8064	3
4 Monate	10 752	4
6 Monate	16 128	6
8 Monate	21 504	8
9 Monate	24 192	9
1 Jahr	35 040	12

3.3.7 Grenzkosten

Die Grenzkosten spielen in der Energiewirtschaft eine essentielle Rolle. Durch eine Erhöhung der Produktion um eine „Einheit“entstehen zusätzliche Kosten. Die „Einheit“ist im Sinne einer differentiellen Betrachtung zu verstehen. Die dadurch entstehenden zusätzlichen Kosten entsprechen den Grenzkosten, die auch als Marginalkosten bezeichnet werden. Bei der Betrachtung in der Stromerzeugung über die Grenzkosten spielen Investitions- und Kapitalkosten keine Rolle, deshalb sind sie kurzfristige Kosten. Als Beispiel sei ein Kohle Kraftwerk angeführt, dass mit etwas höherer Leistung auch für längere Zeit pro Tag betrieben werden kann. Die Produktion steigt, was einen erhöhten Brennstoffbedarf zu Folge hat. Die variablen Kosten steigen, eventuell zusätzlich durch

Emissionskosten weiter verstärkt. Die Investitions- und Kapitalkosten verändern sich nicht. Die Grenzkosten werden nicht durch den Wirkungsgrad, sondern genau durch den differentiellen Wirkungsgrad bestimmt. Wenn der Wirkungsgrad im Teillastbereich absinkt, so sinkt auch der differentielle Wirkungsgrad. Für die Deckung der Nachfrage spielen die Grenzkosten eine bedeutende Rolle, da sie direkt die Einsatzreihenfolge der Kraftwerke bestimmt. Durch die sogenannte Merit Order und dem Market Clearing Price bestimmt das letzte Gebot, das noch einen Zuschlag für den Strompreis erhält den Strompreis. Somit wird der Strompreis vom teuersten Kraftwerk bestimmt, das noch zur Deckung der Nachfrage abgerufen wird.

Die Grenzkosten in Euro/MWh entsprechen der 1. Ableitung der Kostenfunktion nach der elektrischen Leistung. Der Rechenvorgang ist in Gl. (3.7) - Gl. (3.10) gezeigt.

$$C_t = p_{fuelallin,t} \cdot P_{thermisch,t} + p_{O\&M,t} + p_{additional,t} \cdot P_t + C_{Start,t} \tag{3.7}$$

$$P_{thermisch} = a_{lin} + b_{lin} \cdot P \tag{3.8}$$

$$C_t = p_{fuelallin,t} \cdot (a_{lin} + b_{lin} \cdot P_t) + p_{O\&M,t} + p_{additional,t} \cdot P_t + C_{Start,t} \tag{3.9}$$

$$\frac{\mathrm{dC_t}}{\mathrm{d}P_t} = MC = p_{fuelallin,t} \cdot b_{lin} + p_{additional,t} \tag{3.10}$$

Die Grenzkosten eines Gas- und Dampfkombikraftwerks belaufen sich häufig in einem Bereich zwischen 30 Euro/MWh bis 70 Euro/MWh.

3.3.8 variable Durchschnittskosten

Die variablen Durchschnittskosten in Gl. (3.11) in Euro/MWh enthalten keine sprungfixen Startkosten und geben Auskunft über die Leerlaufkosten eines Kraftwerks.

$$\begin{aligned} AC_t &= p_{fuelallin,t} \cdot \frac{P_{thermisch,t}}{P_t} + p_{additional,t} \\ &= p_{fuelallin,t} \cdot \frac{a_{lin} + b_{lin} \cdot P_t}{P_t} + p_{additional,t} \end{aligned} \tag{3.11}$$

Sie betragen 0, sofern das Kraftwerk still steht und pendeln sich bei der technischen Maximalleistung ein. Speziell beim An- bzw. Abfahren steigen die variablen Durchschnittskosten stark an. Sie erhöhen sich, sobald das Kraftwerk in Betrieb ist und der Strompreis die Grenzkosten unterschreitet, wenn auch nur kurzfristig.

3.3.9 Rechenergebnis

Als Ergebnis liefert das Modell mittels Export eine *.csv* Datei mit den jeweiligen Zuständen, dem optimalen Abrufplan, die Brennstoffwärmeleistung, der Deckungsbeitrag, die Grenzkosten und die variablen Durchschnittskosten, welche im weiteren Verlauf grafisch aufbereitet werden.

4 Analytische Methode

Der folgende Abschnitt widmet sich der analytischen Methode für die Implementierung der Software Applikation. Eingangs werden die verwendeten Optimierungsverfahren kurz beschrieben und auf deren wichtigsten Merkmale und Unterschiede eingegangen. Im Anschluss werden die beiden Modelle mit dem jeweiligen Lösungsansatz und die für das Verständnis wichtigen Kernpunkte erläutert.

4.1 Überblick Optimierungsverfahren

Für die Modellierung werden zwei unterschiedliche Optimierungsverfahren verwendet. In MatLab wird das Modell mit einer gemischt-ganzzahlig linearen Optimierung (Mixed Integer Linear Programming – MILP) und in Julia mit rückwärts dynamischer Programmierung (Backward Dynamic Programming - DP) realisiert. Im Folgenden werden beide Optimierungsverfahren beschrieben.

4.1.1 Mixed Integer Linear Programming (MILP)

Die gemischt-ganzzahlig lineare Optimierung ist eine praktische Modellierung zur Lösung von Problemen, für die es keine ausdrücklichen Algorithmen gibt. Grundsätzlich besteht sie aus einer zu optimierenden Zielfunktion und aus einer festgelegten Anzahl an Gleichungen und Ungleichungen, die das mathematische Problem beschränken. Die Lösung muss alle Gleichungen und Ungleichungen erfüllen, um als gültige Lösung zu existieren, andernfalls ist die Zielfunktion nicht optimierbar bzw. existiert nur eine triviale Null-Lösung. Exakt ausgedrückt ist die zu optimierende Variable die sogenannte Entscheidungsvariable, der Lösungsraum ist die Vielzahl zulässiger Lösungen und die Zielfunktion weißt jeder Lösung einen Wert zu. Die Entscheidungen werden somit Variablen zugeordnet und der Lösungsraum[1] wird durch die Gleichungen und Ungleichungen beschrieben. Die Lösungen werden nach dem Wert des Zielfunktionals verglichen. Die verwendete Zielfunktion beschränkt sich auf eine Dimension im Gegensatz zu mehrdimensionalen Optimierungsproblemen, wie die Pareto-Optimierung oder multikriterielle Optimierung. Als Beispiele in der linearen Optimierung sind Prozess-Scheduling, optimaler Kraftwerkseinsatz (operativ), optimale Investitionsentscheidungen und kürzeste Wege Probleme anzuführen.

[1] Der Lösungsraum ist konvex, da die Zielfunktion und die erlaubten Werte konvex sind. Streng genommen erfolgt durch die lineare Zielfunktion die Optimierung über einen konvexen Polyeder.

Die Lösung von gemischt-ganzzahlig linearen Problemen wird in der Praxis meist von sogenannten Solvern, wie zum Beispiel *GUROBI* oder *CPLEX*, vorgenommen. Zweifellos ist die Ermittlung einer optimalen Lösung häufig eine schwierige Aufgabe und nimmt abhängig von Größe und Komplexität bedeutend viel Zeit in Anspruch. Für das Lösungsverfahren gibt es verschiedene Ansätze. Am weitverbreitetsten sind die Branch-and-Bound Methode, Schnittebenenverfahren oder die Verwendung von Heuristiken. Das Branch-and-Bound Verfahren nutzt einen sogenannten Entscheidungsbaum für die Lösungsfindung. Der Ansatz versucht durch Probieren zu einer Lösung zu kommen. Da dieses Prinzip sehr ineffizient und zeitintensiv ist, wird die Menge der möglichen Lösungen in weitere kleinere mögliche Lösungsteilmengen (Branch) aufgespalten mittels geeigneter Schranken (Bound), damit der Lösungsraum möglichst klein gehalten wird. Heuristiken funktionieren nach dem Trial and Error Prinzip. Es werden anhand von statistischen Auswertungen und Zufalls-Stichproben ein Ausschlussverfahren generiert, um dadurch zu einer möglichen Lösung zu gelangen. In der Praxis kommen meist Kombinationen aus verschiedenen Ansätzen zur Verwendung. Die meisten Probleme in der Realität weisen Nichtlinearitäten und/oder diskrete Werte (etwa 0 oder 1 Entscheidungen) auf, welche linearisiert werden müssen. Die Anwendung von linearer Programmierung erfordert Kompromisse und Geschick des Modellierers, damit das Modell die Realität trotz Linearisierung möglichst gut abbildet. Große Optimierungsmodelle erfordern auch numerische Überlegungen, etwa bei der Skalierung von Variablen und Gleichungen.

In der Realisierung des Modells mit MatLab wird der Solver *GUROBI* verwendet. Dieser verwendet als Lösungsansatz eine Kombination aus Branch-and-Bound, Presolving, Schnittebenenverfahren, Heuristiken und Parallelismus [6].[2]

Als Objective versteht *GUROBI* die Zielfunktion, welche es zu minimieren gilt. Die Einschränkungen, als Constraints bezeichnet, hat das Modell einzuhalten. Die Form des Mixed Integer Programming Problems lässt sich durch die Standardform anhand Gl. (4.1) - Gl. (4.6) darstellen.

$$Objective : min_{x_1,x_2,...} f(x_1, x_2, ...) \tag{4.1}$$

$$boundConstraints : g_i(x_1, x_2, ...) \leq b_i, i = 1, ..., p \tag{4.2}$$

$$linearConstraints : h_j(x_1, x_2, ...) = 0, j = 1, ..., q \tag{4.3}$$

Zielfunktion

$$f(x_1, x_2, ...) \tag{4.4}$$

Ungleichungsbedingungen

$$g_i(x_1, x_2, ...) \leq b_i \tag{4.5}$$

[2] Für weitere Informationen wird auf einschlägige Fachliteratur verwiesen.

Gleichungsbedingungen

$$h_j(x_1, x_2, ...) = 0 \tag{4.6}$$

Sofern sich das Problem auf diese Grundform bringen lässt und die Bedingung der Linearität eingehalten wird, kann versucht werden, es durch ein Optimierungsverfahren zu lösen. Dabei kann es unlösbar sein, d.h. es existieren keine zulässigen Lösungen. Sofern es lösbar ist, unterscheidet man zwischen unbeschränkter linearer Programmierung, wofür es keine optimale Lösung gibt oder beschränkter linearer Programmierung, die genau eine optimale Lösung zulässt.

4.1.2 Backward Dynamic Programming (DP)

Die dynamische Programmierung kann zur Lösung von Optimierungsproblemen herangezogen werden, wenn sich dieses Problem in Teilprobleme zerlegen lässt. Das sogenannte Optimalitätsprinzip von Bellman [7] beschreibt den Zusammenhang, dass sich die optimale Lösung des Problems aus den optimalen Lösungen der Teilprobleme zusammensetzt.[3] Die Summe aus den lokalen Optima entspricht dem globalen Optimum. Die Teilprobleme sind einfacher zu lösen bzw. zu optimieren und lassen sich somit als optimale Lösung des Gesamtproblems heranziehen. Des Weiteren werden einmal berechnete Lösungen von Teilproblemen gespeichert und bei gleichartigen Teilproblemen wird auf die zuvor berechnete Zwischenlösung zurückgegriffen, anstatt sie erneut zu berechnen. Dieses Verfahren auf dem sich das Modell stützt wird *Memoization* genannt. Dies hat direkte Auswirkung auf die Rechenzeit der Optimierung. Die dynamische Programmierung verwendet eine „Bottom-Up“Lösungsstrategie. Die Dynamische Programmierung kann in der Entwicklung relativ komplexen Charakter besitzen und in der Realisierung simpel ablaufen. Das folgende Schema dient dazu das Grundprinzip von dynamischer Programmierung zu verstehen. Im Allgemeinen kann dynamische Programmierung nur auf Probleme angewendet werden, die sich auch wirklich in kleine Teilprobleme aufspalten lassen. Das hat zur Folge, dass die Lösung des großen Problems sich aus den Lösungen der kleineren Teilprobleme zusammensetzt, wie in Abbildung 4.1 dargestellt.

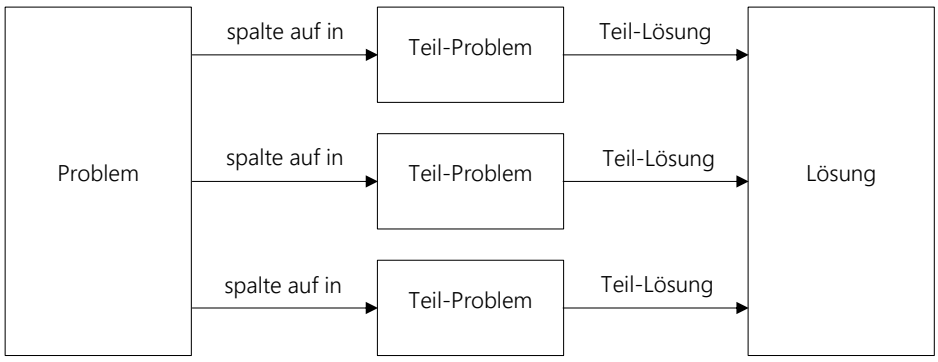

Abbildung 4.1: Konzept der dynamischen Programmierung: Zerlegung des komplexen Optimierungsproblems in einfachere Teil-Probleme.

[3] Das Optimalitätsprinzip von Bellman [8] ist nur bei konvexen Problemen anwendbar.

Durch die vielen Teil-Lösungen gelangt man schneller an die Lösung des Gesamtproblems, da diese einfacher und schneller zu lösen sind. Die CPU wird dadurch entlastet. Hingegen wird der Arbeitsspeicher mehr belastet, da im Gegenzug die Teil-Lösungen gespeichert werden müssen. Im implementierten Modell in Julia, das dynamische Programmierung nutzt, wird durch eine Erweiterung des Algorithmus der Arbeitsspeicher entlastet. Der Grund dafür liegt darin, dass nicht jedes Teil-Problem gespeichert wird. Sobald kleine Teil-Probleme in ein größeres Teil-Problem gelöst wurden, wird nur mehr das Ergebnis des größeren Teil-Problems gespeichert, wie in Abbildung 4.2 dargestellt. Die vorher berechneten Ergebnisse werden verworfen, da ihre Information redundant ist. Dadurch verringert sich der Speicheraufwand bei größeren Problemen enorm.

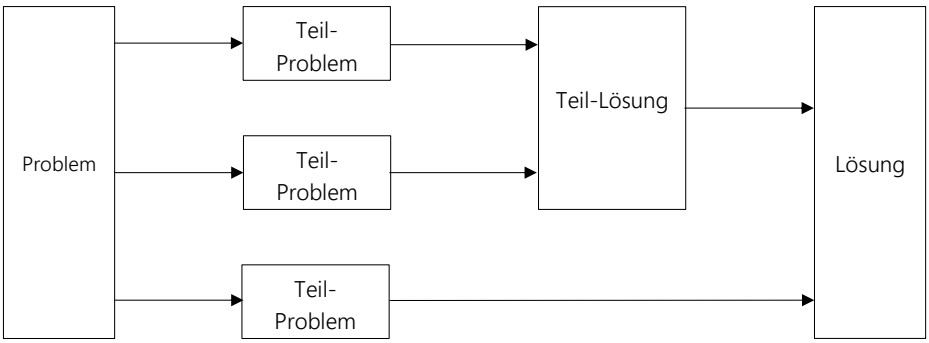

Abbildung 4.2: Erweitertes Konzept der dynamischen Programmierung durch Einführung einer weiteren Lösungsebene, die zwischengespeichert und für spätere Teil-Lösungen verwendet wird.

Die dynamische Programmierung ist grundsätzlich ein einfaches Optimierungsverfahren zur Lösung von komplizierten Algorithmen. An dieser Stelle sei erwähnt, dass die Überlegung des Lösungsansatzes, bis hin zur Umsetzung sehr aufwändig und zeitintensiv sein kann. Dieser Nachteil wird durch das elegante und schnelle Lösen von Problemen aufgewogen. Prinzipiell kann der grundlegende Ablauf in vier Schritte zerlegt werden:

1. Charakterisierung der Struktur einer optimalen Lösung
2. Rekursive Definition des Wertes einer optimalen Lösung
3. Berechnung des Wertes einer optimalen Lösung (rekursiv)
4. Konstruktion der optimalen Lösung aus berechneter Information

Das Verfahren zur Lösung des Modells basiert auf diesen Schritten. Es sei an dieser Stelle nachdrücklich bemerkt, dass sich die Umsetzung weitaus komplexer gestaltet.

4.2 Modellierung mittels Mixed Integer Linear Programming in MatLab

In diesem Abschnitt wird die Modellentwicklung in MatLab mittels gemischt-ganzzahliger linearer Programmierung detailliert beschrieben. Das Fundament für das zu optimierende

Modell liefert die mathematische Beschreibung auf dem bereits erwähnten Standardmodell der linearen Programmierung. Auf markante Modellabschnitte wird eingegangen, sofern dies für das Verständnis erforderlich ist.

Die Zeitreihen für die Simulation liegen über *.xls* Dateiformat vor und werden direkt in MatLab eingelesen. Für die Berechnung werden MatLab 2015 und *YALMIP* als Optimierungs-Toolbox verwendet. Die Berechnung wird auf einem *Intel Core i5-4690K* (4 CPUs; 3,50 GHz), *Windows 10 Enterprise 64-Bit* und 8 GB Arbeitsspeicher ausgeführt.

4.2.1 MatLab Toolbox YALMIP

Für die Lösung des Optimierungsproblems wird die Toolbox *YALMIP* verwendet. Sie vereinfacht die Darstellung von Modellen der linearen Programmierung. Grundsätzlich ist es möglich mit wenigen simplen Befehlen bereits ein Modell zu erstellen und zu lösen.

```
Options = sdpsettings('solver', 'GUROBI', ...
  'GUROBI.Threads', 4, ...
  'GUROBI.MIPFocus', 2, ...
  'GUROBI.MIPGap', 0.05, ...
  'GUROBI.MIPGapAbs', 1e2, ...
  'GUROBI.TimeLimit', 1000, ...
  'verbose', 2, ...
  'warning', 1, ...
  'showprogress', 2);
```

Codeausschnitt 4.1: Parameterkonfiguration der YALMIP Toolbox für GUROBI.

Als Konfiguration für *GUROBI* werden die Einstellungen gemäß Codeausschnitt 4.1 vorgenommen. In Tabelle 4.1 ist die Beschreibung der relevanten Parameter enthalten.

Tabelle 4.1: Parameterkonfiguration der YALMIP Toolbox für GUROBI.

Parameter	Beschreibung
Gurobi.Threads	Anzahl der genutzten Kerne für die Berechnung
Gurobi.MIPFocus	Fokus auf Performance oder Genauigkeit
Gurobi.MIPGap	Abbruch bei erreichen des relativen Werts
Gurobi.MIPGapAbs	Abbruch bei erreichen des absoluten Werts
Gurobi.TimeLimit	Abbruch nach Überschreitung des Zeitlimits

Nach der Definition der Constraints und dem Objective kann das Modell mittels des Befehls in Gl. (4.7) gelöst werden.

$$Solution = optimize(Constraints, Objective, Options) \tag{4.7}$$

In erster Linie lässt sich die Zielfunktion häufig einfach implementieren. Am Rande sei bemerkt, dass die Formulierung der Einschränkungen unter Umständen sehr komplex sein kann. An dieser Stelle sei der Vollständigkeit halber erwähnt, dass es oft möglich

ist eine Einschränkung auf mehrere Arten zu implementieren. Dabei muss der Aspekt beachtet werden, dass diese auch zu unterschiedlichen Ausführungsdauern führen.

4.2.2 Zielfunktion

Die mathematische Beschreibung des Modells basiert auf dem Standardmodell der linearen Programmierung, wodurch *GUROBI* das Modell lösen kann.

Die Maximierung des Ertrags als Zielfunktion ergibt sich nach Gl. (4.8).

$$\begin{aligned}\Pi = max_{P_n} \sum_{n=1}^{N} [\sum_{k=1}^{K} (p_k \cdot P_{n,k} \cdot \Delta T) \\ - [p_{n,k,fuelallin} \cdot (a_n \cdot u_{n,k} + b_n \cdot P_{n,k} + c_n \cdot P_{n,k}^2) \\ + p_{n,misc} \cdot P_{n,k} + (p_{n,O\&M} \cdot u_{n,k})] \cdot \Delta T \\ - (C_{n,cold}^{SU} \cdot z_{n,k,cold} + C_{n,warm}^{SU} \cdot z_{n,k,warm} + C_{n,hot}^{SU} \cdot z_{n,k,hot})]\end{aligned} \tag{4.8}$$

In Tabelle 4.2 ist die Beschreibung der verwendeten Symbole und Entscheidungsvariablen der Zielfunktion enthalten.

Tabelle 4.2: Symbollegende für die zu minimierende Zielfunktion.

Symbol	Einheit	Beschreibung
n	p.u.	Block n ... N
k	p.u.	Zeitschritt k ... K
p_k	EUR/MWh	aktueller Spot-Marktpreis
$P_{n,k}$	MW	elektrische Abrufleistung
T	h	Zeitintervall (15 Minuten)
$p_{n,k,fuellallin}$	EUR/MWh	Brennstoffpreis
a_n	MW th.	quadr. inv. Produktionsfunktion a
b_n	MW th./MW	quadr. inv. Produktionsfunktion b
c_n	MW th./MW²	quadr. inv. Produktionsfunktion c
$p_{n,misc}$	EUR/h	stundenabhängige Kosten
$p_{n,O\%M}$	EUR/MWh	Betriebskosten
$u_{n,k}$	1	On/Off Status
C^{SU}	EUR	Startkosten (Heiß, Warm, Kalt)
$z_{n,k,...}$	1	Starttyp

Die Zielfunktion besteht aus dem Gewinn, erwirtschaftet durch die Vermarktung am Strompreis abzüglich dem Verlust, verursacht durch den Brennstoffeinkauf, Startkosten, Betriebskosten und sonstige Kosten. In der Zielfunktion ist bereits eine Erweiterung auf mehrere verschiedene Kraftwerksblöcke enthalten. Das Modell behandelt jedoch nur einen thermischen Kraftwerksblock.

Da *GUROBI* nur Minimierungsprobleme lösen kann und an dieser Stelle ein Maximierungsproblem aufgrund der Ertragsmaximierung vorliegt, wird lediglich das Vorzeichen nach Gl. (4.9) verändert, um das gewünschte Resultat zu erreichen.

$$TotalProfit = -\left[Earnings - (Costs_{Fuel} + Costs_{misc} + Costs_{O\&M} + Costs_{start})\right] \quad (4.9)$$

4.2.3 Entscheidungsvariablen

Für die Optimierung werden zusätzlich zur Abrufleistung weitere Entscheidungsvariablen, sogenannte Symbolic Decision Variables (*sdpvar*) angelegt, die in Tabelle 4.3 aufgelistet und beschrieben sind.

Tabelle 4.3: Beschreibung der Entscheidungsvariablen, die für die Lösung in GUROBI definiert sind.

Symbol	Datentyp	Beschreibung
Power_Current	Int	elektrische Abrufleistung
On_Off	Bool	Block in Betrieb
Online	Bool	techn. Mindestleistung überschritten
Start_Up	Bool	Startinformation
Shut_Down	Bool	Stillstandsinformation
Start_Hot	Bool	Heißstart wird ausgeführt
Start_Warm	Bool	Warmstart wird ausgeführt
Start_Cold	Bool	Kaltstart wird ausgeführt
Ramp_Hot	Bool	aktiver Heißstartvorgang
Ramp_Warm	Bool	aktiver Warmstartvorgang
Ramp_Cold	Bool	aktiver Kaltstartvorgang
Ramp_Down	Bool	aktiver Abfahrvorgang
Ramp_Hot_Power	Int	Leistungswerte der Heißstartrampe
Ramp_Warm_Power	Int	Leistungswerte der Heißstartrampe
Ramp_Cold_Power	Int	Leistungswerte der Warmstartrampe
Ramp_Down_Power	Int	Leistungswerte der Kaltstartrampe
Ramp_Hot_DiffPower	Int	Differenzleistungswerte eines Heißstarts
Ramp_Warm_DiffPower	Int	Differenzleistungswerte eines Warmstarts
Ramp_Cold_DiffPower	Int	Differenzleistungswerte eines Kaltstarts
Ramp_Down_DiffPower	Int	Differenzleistungswerte einer Abfahrt

Die Entscheidungsvariablen werden über die Kennung *intvar* bzw. *binvar* deklariert, wobei bei Deklaration die Dimension bereits festgelegt werden muss. Nach der Optimierung werden die Entscheidungsvariablen über den Befehl *value(...)* zu verwertbaren Werten konvertiert.

4.2.4 zulässige Grenzen

Für die Einhaltung der zulässigen Grenzen muss die Abrufleistung unterhalb der erlaubten Maximalleistung und positiv sein. Dies führt auf Gl. (4.10).

$$0 \leq Power\_Current(k) \leq On\_Off(k) \cdot Power\_Output\_Max(k) \tag{4.10}$$

Sofern der Block bereits Hochgefahren wurde und sich im Online Status befindet, also die technische Mindestleistung überschritten wurde, darf das Kraftwerk nur mehr oberhalb der technischen Mindestleistung operieren. Die Entscheidungsvariable *Online(k)* legt diese Einschränkung fest.

$$Online(k) \cdot Power\_Output\_Min(k) \leq Power\_Current(k) \tag{4.11}$$

Die Entscheidungsvariable *Online(k)* in Gl. (4.11) ist genau dann inaktiv, wenn das Kraftwerk still steht, oder aktuell eine Rampe aktiv ist. Dadurch wird die Einschränkung unwirksam und die elektrische Abrufleistung kann auch Werte unterhalb der technischen Mindestleistung annehmen. Dies ist wichtig, um eine Rampe starten lassen zu können. Um Sprünge aus dem Stillstand und der technischen Mindestleistung unterbinden zu wollen, wird eine weitere Ungleichung benötigt, welche den Leistungswert automatisch auf den entsprechenden Wert im festgelegten Rampenverlauf setzt.

$$\begin{aligned} &Ramp\_Hot\_Power(k) \\ &\qquad + Ramp\_Warm\_Power(k) \\ &\qquad + Ramp\_Cold\_Power(k) \\ &\qquad + Ramp\_Down\_Power(k) \leq Power\_Current(k) \end{aligned} \tag{4.12}$$

Die Hilfsvariablen *Ramp_..._Power(k)* der Rampen in Gl. (4.12) speichern den korrekten Leistungswert eines Rampenverlaufs. Sie enthalten genau dann einen Wert, falls eine Rampe gefahren wird, andernfalls 0. Zu einem Zeitpunkt kann nur maximal eine Hilfsvariable einer Rampe ungleich 0 sein, da gleichzeitig nur eine Rampe gefahren werden darf. Die Einschränkung lässt sich so verstehen, dass sie im regulären Betrieb lediglich die Positivität der elektrischen Abrufleistung festlegt. Diese Eigenschaft ist so gesehen redundant, da sie bereits durch eine andere Einschränkung fixiert wurde. Im Falle eines Rampenverlaufs jedoch wird sie aktiv. Dann hebt sie infolgedessen die minimale elektrische Leistung genau auf den Leistungswert der Rampe, die aktuell gefahren wird. Somit gewährleistet diese Formulierung, dass alle Rampen korrekt vollständig bis an ihre festgelegtes Ende gefahren werden und danach befindet sich das Kraftwerk entweder im Online Status oder im Stillstand. Ab diesem Zeitpunkt übernehmen andere Einschränkungen die Rolle der Einhaltung des Leistungsbandes.

Falls die Entscheidung feststeht, dass der Kraftwerksblock hochgefahren werden soll, so ist der Vorzustand mit 0 MW eindeutig definiert. Beim Abfahren jedoch kann der

Vorzustand ein beliebiger Leistungswert oberhalb der technischen Mindestleistung sein. Deshalb wird in diesem Fall die Leistung nach oben hin begrenzt und eine maximal erlaubte Leistung festgelegt. Dies ist nur der Fall, wenn die Hilfsvariable *Ramp_Down* zum Zeitpunkt *k* gesetzt ist. Ansonsten bleibt die reguläre technische Maximalleistung erhalten. *Ramp_Down_Power(k)* wird nur gesetzt, falls sich der Block in einer Abfahrtsrampe befindet und ist ansonsten 0.

$$\begin{aligned} Power\_Current(k) \leq\ & Ramp\_Down\_Power(k) \\ & + (1 - Ramp\_Down(k)) \cdot Power\_Output\_Max(k) \end{aligned} \tag{4.13}$$

Im Falle eines regulären Betriebs ist *Ramp_Down(k)* nicht gesetzt und *Ramp_Down_Power(k)* auch 0, gezeigt durch Gl. (4.13). Dadurch wird lediglich der Teil *Power_Output_Max(k)* aktiv und begrenzt die Leistung auf die technische Maximalleistung. Im Falle einer Abfahrrampe ist *Ramp_Down(k)* gesetzt und *Ramp_Down_Power(k)* ungleich 0. Dadurch wird der zweite Teil inaktiv und die Leistung automatisch mit *Ramp_Down_Power(k)* limitiert.

4.2.5 Hoch- bzw. Abfahren

Zum Zeitpunkt *k*=1 wird durch Gl. (4.14) und Gl. (4.15) festgelegt, dass weder eine Start-, noch eine Abfahrrampe zugelassen sind.

$$Start\_Up(1) == 0 \tag{4.14}$$

$$Shut\_Down(1) == 0 \tag{4.15}$$

Dies ist insofern notwendig, da der Anfangszustand für die Lösung des Optimierungsproblems feststehen muss. In der Realität ist diese Festlegung unproblematisch, da bei einer sieben Tage Optimierung der Zustand von vor sieben Tagen eindeutig ist.

Für alle weiteren Betrachtungen lässt sich durch Gl. (4.16) die Start- und Stoppvariable für alle Zeitpunkte *k* festlegen.

$$On\_Off(k) - On\_Off(k-1) == Start\_Up(k) - Shut\_Down(k) \tag{4.16}$$

Die linke Seite der Gleichung liefert genau dann 1, falls das Kraftwerk hochgefahren wird. Dadurch muss die rechte Seite auch 1 sein, damit die Gleichung erfüllt ist. Dies erzwingt *Start_Up(k)* auf 1 und speichert die Startinformation. Falls die linke Seite der Gleichung -1 ergibt, so wird das Kraftwerk vom Betrieb in den Stillstand gebracht. Dadurch wird *Shut_Down(k)* auf -1 gesetzt. Falls sich der Zustand des Kraftwerks nicht ändert, also der Block im Betrieb oder im Stillstand bleibt, so werden auch keine Start- und Stopp Informationen gespeichert.

Als zusätzliche Einschränkung gilt, dass immer nur maximal eine der beiden Entscheidungsvariablen zu einem Zeitpunkt aktiv sein darf. Dies wird durch die Ungleichung in Gl. (4.17) gewährleistet.

$$Start\_Up(k) + Shut\_Down(k) \leq 1 \tag{4.17}$$

4.2.6 Typ des Starts für Rampen

Es existiert für die Wahl der Rampe für jeden Typ eine eigene Entscheidungsvariable. Diese darf nur dann aktiv werden, falls die entsprechende Rampe gestartet wird. Um die Berechnungsdauer zu reduzieren, wird eine Prioritätenliste implizit eingeführt, die den günstigsten Starttyp, somit den Heißstart präferiert und erst als letztes Mittel, falls keine anderen Starttypen erlaubt sind, den Kaltstart zulässt. Diese Prioritätenliste wird durch eine Reihung in der Abarbeitung des Algorithmus umgesetzt.

$$Start\_Hot(k) \leq Start\_Up(k) \tag{4.18}$$

Durch die Einschränkung in Gl. (4.18) darf *Start_Hot(k)* nur dann aktiv sein, wenn auch ein Startsignal gesetzt wurde. In jedem anderen Fall wird kein Heißstartsignal gesetzt und dadurch auch kein Warm- bzw. Kaltstart freigeschaltet, da die Prioritätenliste vorher bereits abbricht.

$$\begin{aligned} Start\_Hot(k) \leq\ & On\_Off(k - (Start\_Hot\_Parameter - 1)) \\ & + On\_Off(k - (Start\_Hot\_Parameter - 2)) \end{aligned} \tag{4.19}$$

Für die Ungleichung in Gl. (4.19) ist der Wechsel der *On_Off* Variable von einem Zeitpunkt k zum darauffolgenden Zeitpunkt k-1 von Bedeutung. Sie erfasst den Zeitpunkt, wann das Kraftwerk abgeschaltet wurde. Dabei wird um die maximale zulässige Stillstandsdauer für den entsprechenden Rampentyp in die Vergangenheit geblickt. Falls zu diesem Zeitpunkt das Kraftwerk heruntergefahren wurde, ergibt die rechte Seite der Ungleichung 1. Sofern diese Zeitspanne kleiner ist, als die minimale Dauer, die für einen Warmstart benötigt wird, handelt es sich um einen Heißstart. Die Variable *Start_Hot(k)* wird somit gesetzt, da die Stillstandsdauer kleiner als die maximal zulässige Dauer für einen Heißstart ist.

Ein Warmstart ist eine Ebene unterhalb eines Heißstarts in der Prioritätenliste. Somit besteht die Möglichkeit eines Warmstarts, nachdem ein Heißstart ausgeschlossen wurde.

$$Start\_Warm(k) \leq Start\_Up(k) - Start\_Hot(k) \tag{4.20}$$

Sobald ein Heißstart erlaubt ist, ergibt die rechte Seite der Gl. (4.20) 0 und unterbindet somit ein aktiv werden einer Warmstart Kondition.

Ob ein Warmstart aufgrund der Einhaltung der Stillstandszeiten zulässig ist, entscheidet Gl. (4.21). Ihr Aufbau ist vom selben Typ der Ungleichung für den Heißstart.

$$\begin{aligned} Start\_Warm(k) \leq\ & On\_Off(k - (Start\_Warm\_Parameter - 1)) \\ & + On\_Off(k - (Start\_Warm\_Parameter - 2)) \end{aligned} \tag{4.21}$$

Falls weder ein Heiß-, noch ein Warmstart zulässig sind, kann es sich nur mehr um einen Kaltstart nach Gl. (4.22) handeln.

$$Start\_Cold(k) \leq Start\_Up(k) - Start\_Hot(k) - Start\_Warm(k) \tag{4.22}$$

Ein Kaltstart hat keine Begrenzung hinsichtlich minimaler Stillstandsdauer.

Eine weitere Restriktion nach Gl. (4.23) garantiert die Einhaltung der angeführten Gleichungen und Ungleichungen, die zuvor erläutert wurden. Die Summe aller ausgeführten Starttypen muss exakt gleich der Summe der angeforderten Starts sein.

$$sum(Start\_Up) == sum(Start\_Hot) + sum(Start\_Warm) + sum(Start\_Cold) \tag{4.23}$$

4.2.7 Minimale Betriebsdauer

Für die Einhaltung der minimalen Betriebsdauer wird mittels eines Indikators bestimmt, ob das Kraftwerk eingeschaltet wurde. Der Wert des Indikators wird dadurch auf 1 gesetzt. Andernfalls 0 oder -1. Durch die Verwendung einer Hilfsvariablen wird der zeitliche Bereich festgelegt, in der das Kraftwerk in Betrieb bleiben muss. Gl. (4.24) - Gl. (4.26) zeigen die Implementierung des Indikators.

$$indicator = On\_Off(k) - On\_Off(k - 1) \tag{4.24}$$

$$range = k : (k + Time\_On\_Min) \tag{4.25}$$

$$On\_Off(range) \geq indicator \tag{4.26}$$

Der Indikator ist nur im Falle eines Starts oder Stopps ungleich 0. Im Falle eines Starts wird er auf 1 gesetzt. Die Hilfsvariable *range* speichert eine Spannweite vom aktuellen Zeitpunkt bis zum aktuellen Zeitpunkt plus der minimalen Betriebsdauer an Zeitschritten. Durch die Ungleichung wird sichergestellt, dass die minimale Betriebsdauer eingehalten wird, da im Falle eines Betriebsstarts der Indikator aktiv ist und somit alle weiteren *On_Off* Einträge innerhalb der Spannweite auf 1 zwingt.

```
// indicator will be 1 only when switched on, otherwise 0 or -1
indicator = On_Off(k) - On_Off(k-1);
range = k:min(T, k + Time_On_Min / Time_Interval - 1);
Constraints = [Constraints, On_Off(range) >= indicator];
```

Codeausschnitt 4.2: Implementierung der Einschränkung für die Einhaltung der minimalen Betriebsdauer.

Tabelle 4.4 zeigt die interne Abbildung der Hilfsvariablen.

Tabelle 4.4: Beispielhafter Ablauf für die Einhaltung der minimalen Betriebsdauer.

Minimale Betriebsdauer						
Zeitpunkt	$k-1$	k	$k+1$	$k+2$	$k+3$	$k+4$
On_Off	0	1	1	1	1	1
Start_Up	0	1	0	0	0	0

An dieser Stelle wird darauf hingewiesen, dass die minimale Betriebsdauer nicht ab Startkondition, sondern ab Vollendung eines Rampenverlaufs einzuhalten ist. Dies führt auf den Umstand, dass unterschiedliche minimale Betriebsdauern realisiert werden müssen, da die Rampenvorgänge auch unterschiedlichen zeitlichen Charakter besitzen.

4.2.8 Minimale Stillstandsdauer

Die minimale Stillstandsdauer wird nach dem gleichen Prinzip, wie die minimale Betriebsdauer eingehalten. Eine Änderung des Indikators bewirkt eine Detektion auf eine Abfahrt des Kraftwerksblocks. Die Ungleichung für die Einschränkung wird angepasst. Gl. (4.27) - Gl. (4.29) zeigen die Implementierung des Indikators.

$$indicator = On\_Off(k-1) - On\_Off(k) \tag{4.27}$$

$$range = k : (k + Time\_Off\_Min) \tag{4.28}$$

$$On\_Off(range) \leq 1 - indicator \tag{4.29}$$

Der Indikator ist nur im Falle eines Starts oder Stopps ungleich 0. Im Falle eines Stopps wird er auf 1 gesetzt. Die Hilfsvariable *range* speichert eine Spannweite vom aktuellen Zeitpunkt bis zum aktuellen Zeitpunkt plus der minimalen Stillstandsdauer an Zeitschritten. Durch die Ungleichung wird sichergestellt, dass die minimale Stillstandsdauer eingehalten wird, da im Falle eines Stopps der Indikator aktiv ist und somit alle weiteren *On_Off* innerhalb der Spannweite auf 0 zwingt.

```
// indicator will be 1 only when switched off, otherwise 0 or -1
indicator = On_Off(k-1) - On_Off(k);
range = k:min(T, k + Time_Off_Min / Time_Interval - 1);
Constraints = [Constraints, On_Off(range) <= (1 - indicator)];
```

Codeausschnitt 4.3: Implementierung der Einschränkung für die Einhaltung der minimalen Stillstandsdauer.

Tabelle 4.5 zeigt die interne Abbildung der Hilfsvariablen.

Tabelle 4.5: Beispielhafter Ablauf für die Einhaltung der minimalen Stillstandsdauer.

Minimale Stillstandsdauer						
Zeitpunkt	$k-1$	k	$k+1$	$k+2$	$k+3$	$k+4$
On_Off	1	0	0	0	0	1
Shut_Down	0	1	0	0	0	0

4.2.9 Art der Rampe

Für jeden Rampenverlauf existiert eine eigene Entscheidungsvariable, die während dem gesamten Verlauf der Rampe aktiv ist, ansonsten zu jedem Zeitpunkt 0. Nicht zu verwechseln mit der Entscheidungsvariable für den Start des Rampenverlaufs, die nur initial beim Start der Rampe aktiv ist. Als Beispiel wird der Heißstart genauer erläutert. Die Verfahren für Warm-, Kaltstart und Abfahrt sind analog implementiert.

$$sum(Ramp\_Hot) == 6 \cdot sum(Start\_Hot) \tag{4.30}$$

Anhand Gl. (4.30) wird eine Beziehung zwischen der Entscheidungsvariable *Ramp_Hot*, die während des kompletten Rampenvorgangs gesetzt ist und *Start_Hot*, die nur im Zeitschritt, wann die Rampe begonnen wird, aktiv ist, hergestellt. Durch die Länge eines Rampenvorgangs wird auch die Gesamtanzahl der aktiven Zeitschritte festgelegt.

$$\begin{aligned} &Ramp\_Hot(k) + Ramp\_Hot(k+1) \\ &+ Ramp\_Hot(k+2) + Ramp\_Hot(k+3) \\ &+ Ramp\_Hot(k+4) + Ramp\_Hot(k+5) \geq 6 \cdot Start\_Hot(k) \end{aligned} \tag{4.31}$$

Die Ungleichung in Gl. (4.31) bewirkt, dass sobald ein Heißstart signalisiert wurde, alle sechs darauffolgenden zeitlichen Einträge in *Ramp_Hot* auch aktiv werden und somit die Information gewonnen wird, über welchen Zeitraum einen Rampenverlauf stattfindet. Bei genauerer Betrachtung tritt an dieser Stelle ein entscheidender Nachteil des Modells zum Vorschein. Die Modellierung ist nicht komplett von den Eingabeparametern entkoppelt. Bei Änderung der Eingabeparameter müssen unter Umständen Teile des Programmcodes angepasst werden.

4.2.10 Rampenverlauf

Die Information, zu welchem Zeitpunkt welche Rampe wie lange ausgeführt wird, ist bereits durch die entsprechenden Entscheidungsvariablen implementiert. Für die jeweiligen Leistungen, die für einen Rampenverlauf definiert sind, sind weitere Einschränkungen

eingeführt, die nicht von den Eingabeparametern entkoppelt sind. Als Beispiel wird der Heißstart erläutert. Die Fälle Warm-, Kaltstart und Abfahrt sind analog implementiert.

$$\begin{aligned} &Ramp\_Hot\_Power(k) = \\ &Ramp\_Hot\_Parameter(2) \cdot Start\_Hot(k) \\ &+Ramp\_Hot\_Parameter(3) \cdot Start\_Hot(k-1) \\ &+Ramp\_Hot\_Parameter(4) \cdot Start\_Hot(k-2) \\ &+Ramp\_Hot\_Parameter(5) \cdot Start\_Hot(k-3) \\ &+Ramp\_Hot\_Parameter(6) \cdot Start\_Hot(k-4) \\ &+Ramp\_Hot\_Parameter(7) \cdot Start\_Hot(k-5) \end{aligned} \tag{4.32}$$

In der Entscheidungsvariablen *Ramp_Hot_Power* in Gl. (4.32) sind die entsprechenden Leistungswerte für den Heißstart gesetzt, andernfalls sind sie 0. Sie stellen die absoluten Leistungswerte dar und korrelieren mit der Variable *Ramp_Hot*.

$$\begin{aligned} &Ramp\_Hot\_DiffPower(k) = \\ &(Ramp\_Hot\_Parameter(2) - Ramp\_Hot\_Parameter(1)) \cdot Start\_Hot(k) \\ &+ (Ramp\_Hot\_Parameter(3) - Ramp\_Hot\_Parameter(2)) \cdot Start\_Hot(k-1) \\ &+ (Ramp\_Hot\_Parameter(4) - Ramp\_Hot\_Parameter(3)) \cdot Start\_Hot(k-2) \\ &+ (Ramp\_Hot\_Parameter(5) - Ramp\_Hot\_Parameter(4)) \cdot Start\_Hot(k-3) \\ &+ (Ramp\_Hot\_Parameter(6) - Ramp\_Hot\_Parameter(5)) \cdot Start\_Hot(k-4) \\ &+ (Ramp\_Hot\_Parameter(7) - Ramp\_Hot\_Parameter(6)) \cdot Start\_Hot(k-5) \end{aligned} \tag{4.33}$$

In *Ramp_Hot_DiffPower* nach Gl. (4.33) ist die Differenz der aktuellen Leistung zur zeitlich vorhergehenden Leistung eines Rampenverlaufs enthalten, andernfalls ist der Eintrag 0. Diese Entscheidungsvariable ist erforderlich, um im Falle einer Rampe den maximalen Leistungsänderungsgradienten durch die Differenzleistung beim Anfahren zu ersetzen, ohne das Modell anderweitig zu beschränken. Tabelle 4.6 verdeutlicht den Zusammenhang der eingesetzten Entscheidungsvariablen.

4.2.11 maximaler Leistungsänderungsgradient

Der maximale Leistungsänderungsgradient muss allenfalls eingehalten werden, sofern sich der Kraftwerksblock im Online Status befindet.[4] Falls sich das Kraftwerk in einem Rampenverlauf befindet, kommt es vor, dass Leistungssprünge größer als der maximal erlaubte Leistungsänderungsgradient durchgeführt werden müssen. Diese Leistungssprünge dürfen nicht beschränkt werden. Deshalb wird eine zusätzliche Einschränkung nach Gl. (4.34) eingeführt, die den Leistungsänderungsgradienten nur oberhalb der

[4]Die technische Mindestleistung muss dafür überschritten sein.

Tabelle 4.6: Veranschaulichung eines Heißstartvorgangs und die damit verbundenen gesetzten Entscheidungsvariablen.

Beispiel: Heißstart								
Zeitpunkt	$k-1$	k	$k+1$	$k+2$	$k+3$	$k+4$	$k+5$	$k+6$
On_Off	0	1	1	1	1	1	1	1
Start_Up	0	1	0	0	0	0	0	0
Start_Hot	0	1	0	0	0	0	0	0
Ramp_Hot	0	1	1	1	1	1	1	0
Ramp_Hot_Power	0	29	78	110	115	147	264	0
Ramp_Hot_DiffPower	0	29	49	32	5	32	117	0

technischen Mindestleistung wirksam macht. Die Realisierung erfolgt mittels zwei Ungleichungen, welche jeweils für eine positive und eine negative Leistungsänderung zuständig sind.

$$\begin{aligned} &Power\_Current(k) - Power\_Current(k-1) \leq \\ &(On\_Off(k) - Ramp\_Hot(k) - Ramp\_Warm(k) - Ramp\_Cold(k)) \cdot Power\_Delta\_Max \\ &+ Ramp\_Hot\_DiffPower(k) + Ramp\_Warm\_DiffPower(k) + Ramp\_Cold\_DiffPower(k) \end{aligned} \tag{4.34}$$

Die linke Seite der Ungleichung gibt die Leistungsänderung vom vorhergehenden zum aktuellen Zeitpunkt an. Die rechte Seite der Ungleichung setzt sich aus zwei Teilen zusammen, die nur abwechselnd wirksam werden. Im ersten Fall, wenn keine Aufwärtsrampe gefahren wird, ist die Summe der Klammer gleich 1 und der maximale Leistungsänderungsgradient *Power_Delta_Max* wird wirksam, da auch alle *Ramp_..._DiffPower(k)* gleich 1 sind. Der zweite mögliche Fall behandelt einen Rampenverlauf. Es ist immer nur eine Rampenzustandsvariable gesetzt und die Summer der Klammer ergibt 0. Dadurch wird die Leistungsänderung nicht mehr durch den maximalen Leistungsänderungsgradienten bestimmt, sondern durch die entsprechende *Ramp_..._DiffPower(k)*. Es ist immer nur eine *Ramp_..._DiffPower(k)* ungleich 0 und diese bestimmt in diesem Fall die Leistungsänderung, welche ident mit der zugehörigen Rampe ist.

Für den Fall einer negativen Leistungsänderung wird lediglich der maximal erlaubte Leistungsänderungsgradient als Einschränkung anhand von Gl. (4.35) genutzt.

$$Power\_Current(k-1) - Power\_Current(k) \leq Online(k) \cdot Power\_Delta\_Max \tag{4.35}$$

Die Einschränkung die durch eine Abfahrrampe passiert, ist bereits in der Einschränkung für zulässige Grenzen enthalten, in dem sie die maximal erlaubte fahrbare Leistung für diese Zeitpunkte gezielt reduziert.

4.2.12 Aufbereitung der optimierten Entscheidungsvariablen

Nach der Lösung des Optimierungsproblems werden die Entscheidungsvariablen zur Berechnung weiterer Indikatoren herangezogen.

Die thermische Brennstoffwärmeleistung wird über die linearisierte inverse Produktionsfunktion berechnet. Die Grenzkosten zu jedem Zeitpunkt k erhält man durch die 1. Ableitung der Kostenfunktion nach der elektrischen Leistung gemäß Gl. (4.36).

$$MC = p_{fuelallin} \cdot b_{lin} + p_{additional} \tag{4.36}$$

Die variablen Durchschnittskosten in Gl. (4.37) zu jedem Zeitpunkt k werden ohne sprungfixe Startkosten berechnet und sind ungleich 0, sofern die Abrufleistung größer als 0 ist.

$$AC = p_{fuelallin} \cdot \frac{(a_{lin} + b_{lin} \cdot P)}{P} + p_{additional} \tag{4.37}$$

Die Ergebnisse werden mittels *.xls* Export in eine Excel-Datei geschrieben und dargestellt. Im Kapitel Ergebnisse werden diese visualisiert und genauer erläutert.

4.3 Modellierung mittels rückwärts dynamischer Programmierung in Julia

In diesem Abschnitt wird die Modellierung in der Julia-Umgebung detailliert beschrieben. Die Basis für die Lösung des Modells liefert der Algorithmus der rückwärts dynamischen Programmierung. Die mathematische Beschreibung ist ident zu der gemischtganzzahligen linearen Formulierung. Da das angewandte Konzept der rückwärts dynamischen Programmierung verhältnismäßig komplex ist, wird es mittels mehrerer Flussdiagramme anschaulich dargestellt. Auf wichtige Codeausschnitte wird eingegangen, sofern dies für das Verständnis erforderlich ist.

Die Zeitreihen für die Simulation liegen über eine *.xls* Datei vor und werden zu Beginn in eine *.csv* Datei konvertiert, welche direkt in Julia eingelesen werden kann. Für die Programmierung wird die IDE *JuliaPro* verwendet mit Version 0.6.4.1. Als Editor wird *Juno* genutzt. Die Simulation wird auf einem System mit *Intel Core i5-6500* (4 CPUs; 3,20 GHz), *Windows 10 Enterprise 64-Bit* und 16 GB Arbeitsspeicher ausgeführt.

4.3.1 Konzept der erweiterten dynamischen Programmierung

Die rückwärts dynamische Programmierung beschreibt die Richtung, in der das Optimierungsproblem zuerst durchlaufen wird. Dabei wird zum Zeitpunkt T begonnen und bis zum Zeitpunkt T_0 „rückwärts“iteriert. Nach dem Erreichen des Startpunkts T_0 werden die Teilprobleme nach „vorwärts“aufgelöst, bis der Zeitpunkt T erreicht wird. Mathematisch lässt sich diese Vorgehensweise über Gl. (4.38) beschreiben.

$$\Pi(T) = max\left[g(t) + \Pi(T-t)\right], t = 1...T \tag{4.38}$$

Prinzipiell kann man sich das Lösungsverfahren als einen gerichteten Graphen aus der Netzwerktheorie vorstellen. Jeder Knotenpunkt besitzt eine bestimmte Menge an Nachbarn. Es existiert für jeden Knoten eine sogenannte Adjazenzliste[5]. Die Adjazenzliste enthält die Information für die Transition von einem zum zeitlich folgenden Knotenpunkt. Dabei kann ein Übergang erlaubt oder nicht erlaubt sein. Es wird somit ein Knoten-Array aufgespannt, in dem nur bestimmte Übergänge erlaubt sind. Abbildung 4.3 veranschaulicht den Aufbau und die Funktionsweise des gerichteten Graphen. Es ist klar ersichtlich, dass von einem Knotenpunkt aus mehrere Pfade beginnen, die zum Teil in einer Sackgasse enden und abgebrochen werden müssen. Der Umstand, dass mehrere Pfade zu ein und demselben Knoten führt, ist durchaus möglich. Das hat zur Folge, dass die Speicherung der Teil-Lösungen von großer Bedeutung ist, um Rechenzeit einzusparen.

Ziel der rückwärts dynamischen Programmierung ist das Aufsuchen eines oder mehrerer Lösungspfade mit der optimalen Lösung. Es ist klar ersichtlich, dass sich die Anzahl der möglichen Pfade bei steigenden Zeitschritten rasant erhöht. Durch eine hohe Anzahl an erlaubten Nachbarn von einem Knotenpunkt, ergeben sich sehr viele mögliche Pfade. Daraus kann man folgern, dass die Anzahl der möglichen Übergänge von einem Knotenpunkt zum nächsten so gering wie nur möglich zu halten sind, um die Rechendauer gering zu halten.

Durch eine Erweiterung lässt sich die Anzahl der Pfade reduzieren. Die rückwärts dynamische Programmierung wird um eine modifizierte Prioritätenliste[6] erweitert, die aus der ganzzahlig-gemischten linearen Programmierung bekannt ist. Im Allgemeinen lässt sich diese Erweiterung so verstehen, dass bei der Ermittlung der Adjazenzliste zur Laufzeit direkt auch die Information ermittelt wird, welche Übergänge „most-likely“sind. Dadurch entstehen Prioritäten, welche Übergänge am wahrscheinlichsten zum optimalen Lösungspfad führen. Der große Vorteil besteht darin, dass viele Übergänge die zwar möglich sind, aber nicht optimal sind, erst dann berechnet werden, sobald alle vorhergehenden möglichen Übergänge keine eindeutige Lösung liefern. Die eingeführte modifizierte Prioritätenliste verkürzt die Rechendauer der Optimierung signifikant, da viele Pfade die suboptimal sind, nicht berechnet werden.

[5] Nachbarschaftsliste

[6] Streng genommen basiert die Erweiterung auf dem Branch-and-Bound Algorithmus durch Verwendung eines Entscheidungsbaumes.

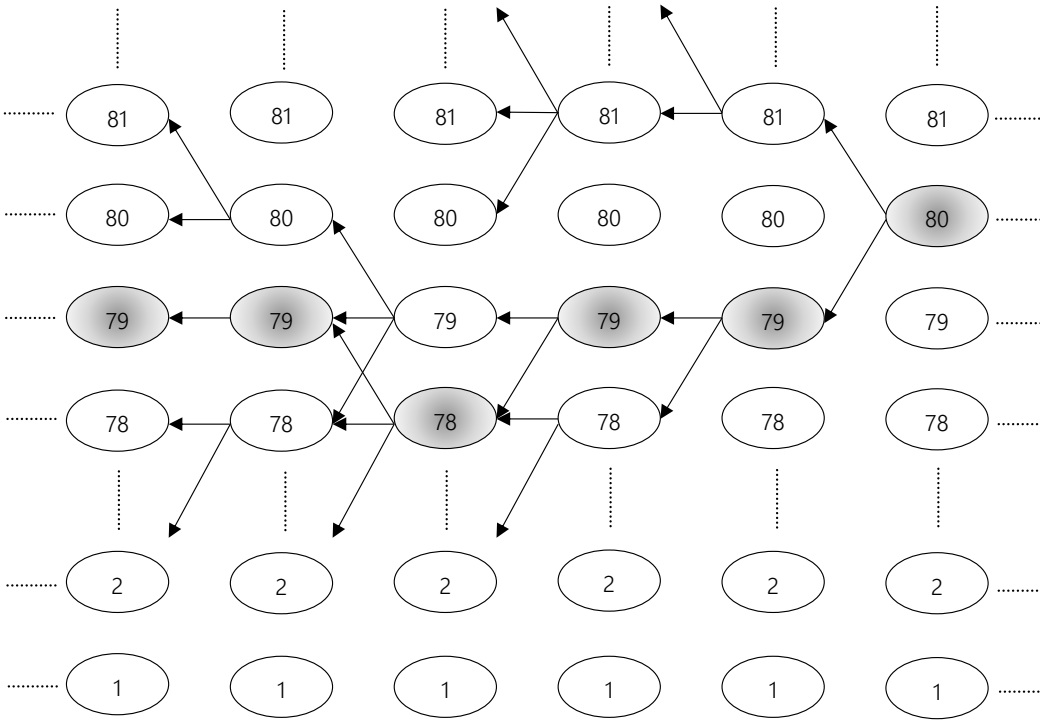

Abbildung 4.3: Ein Knoten-Array wird definiert, das alle Knoten in Form von Zuständen enthält. Auf Basis dieses gerichteten Graphen werden erlaubte und verbotene Übergänge berechnet, welche die Grundlage für die Lösung des Optimierungsmodells darstellen. In vertikale Richtung sind die Zustände aufgetragen. Die Zeitachse entspricht der horizontalen Ebene.

Ein weiterer wichtiger Aspekt der rückwärts dynamischen Programmierung ist die Speicherung von bereits ermittelten Lösungen von Teilproblemen. Dieses Verfahren wird „Memoization“ genannt. Jede Lösung eines Teilproblems wird zwischengespeichert und dem entsprechenden Knotenpunkt zugeordnet. Sobald dieser Knotenpunkt durch einen anderen Pfadverlauf erneut erreicht wird, erfolgt keine redundante Berechnung der optimalen Teillösung, sondern kann diese sofort über die Zwischenspeicherung abgerufen werden. Dieses Verfahren bringt große Einsparung in der Rechenzeit.

4.3.2 schematischer Ablauf

Dieser Abschnitt veranschaulicht den schematischen Ablauf des Programms. In den darauffolgenden Punkten wird im Detail auf die einzelnen Funktionsabläufe eingegangen. Der Algorithmus arbeitet die folgenden Schritte ab:

1. Zu Beginn des Programms werden alle relevanten Daten eingelesen. Die Daten werten für die Weiterverarbeitung geeignet aufbereitet.
2. Die rekursive Funktion *GetOptimalSolution(. . .)* ermittelt den optimalen Lösungspfad auf Basis von rückwärts dynamischer Programmierung.

3. *GetPathStates(...)* überprüft, ob der gefundene Lösungspfad eine zulässige und erlaubte Lösung liefert und ob der Fahrplan in der Realität auch tatsächlich umsetzbar ist.
4. *GetPowerDispatch(...)* liefert den relevanten Fahrplan für den Kraftwerksblock auf Basis des Lösungspfades.
5. *GetPathProfit(...)* liefert den optimalen Profit.

Abbildung 4.4 veranschaulicht den schematischen Ablauf des Programms.

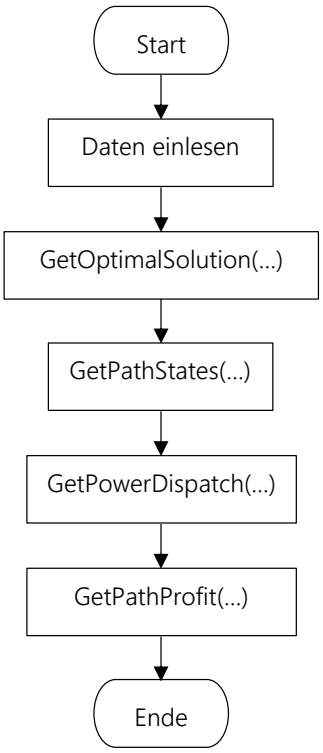

Abbildung 4.4: Schematicher Ablauf des Programmcodes aus Basis von rückwärts dynamischer Programmierung.

4.3.3 verwendete Packages

Für die Softwarelösung werden in Julia zwei Packages über den Befehl *using* eingebunden:

1. Da Julia in der Entwicklung noch relativ jung ist, treten häufiger Probleme beim Lesen und Schreiben von *.xls* Dateien auf. Deshalb wird die *.xls* Datei manuell in eine *.csv* Datei konvertiert. Über das Package „CSV“, welche eine Schnittstelle für Daten Ein- und Ausgabe darstellt und die Befehle *CSV.read()* und *CSV.write()* kann mit *.csv* Dateien gearbeitet werden.
2. Das „DataFrames“Package ist eine Bibliothek für das Arbeiten mit Tabellen programmintern. Es erleichtert die Darstellung der Daten im Programm, da die eingelesenen Daten das „DataFrames“Format haben und von da aus weiterverarbeitet

werden können. Für das Schreiben und elegante Darstellen von Daten lässt sich diese Bibliothek hervorragend nutzen.

4.3.4 Datenimport und Aufbereitung

Über die Funktion *CSV.read()* werden die relevanten Informationen eingelesen und im „DataFrames“Format gespeichert. Um die Berechnungszeit kurz zu halten werden alle Variablen in Integer Typen mittels Codeausschnitt 4.4 konvertiert, außer jenen, bei denen Gleitkommaberechnungen erforderlich sind.

```
variable = convert(Int, INPUT);
```

Codeausschnitt 4.4: Konvertierung der Variablen in ein Integer-Format.

Die Knotenpunkte werden so gewählt, dass ihre Anzahl minimal bleibt, um die Menge an möglichen Zuständen gering zu halten. Jedem Knotenpunkt wird ein ganzzahliger Leistungswert zugeordnet. Die Liste beinhaltet nur jene Leistungswerte, die auch tatsächlich fahrbar sind. Das hat zur Folge, dass unterhalb der technischen Mindestleistung nur jene Leistungswerte berücksichtigt werden, die für die Rampenverläufe erforderlich sind. Des Weiteren darf ein Leistungswert nicht doppelt oder mehrfach in der Liste vorkommen. Die erwähnten Gesichtspunkte bedeuten, dass das Knoten-Array von 0 bis zur technischen Mindestleistung nur jene Leistungswerte enthalten, die auch in einem Rampenvorgang vorkommen. Alle anderen Werte sind nicht enthalten, da sie nicht zulässig sind. Ab der technischen Mindestleistung hat das Knoten-Array eine Auflösung von jeweils 1 MW.

Für gewöhnlich wird die thermische Brennstoffwärmeleistung über die quadratisch inverse Produktionsfunktion für jeden Leistungswert berechnet. Für die Brennstoffwärmeleistung wird eine Liste definiert, die über ihren Index mit der Liste der elektrischen Abrufleistung korreliert. Dies spart Rechenzeit, da die Brennstoffwärmeleistung nicht bei jedem Iterationsschritt über die quadratisch inverse Produktionsfunktion berechnet werden muss, sondern über den entsprechenden Index direkt ausgelesen wird und nur zur Beginn einmalig berechnet werden muss.

Die technische Mindest- und Maximalleistung kann während des Betriebs verändert werden. Dadurch ist sicherzustellen, dass der maximale Leistungsänderungsgradient nicht überschritten wird. Eine Abfrage und anschließende Adaption des Leistungswerts benötigt relativ viel Zeit in der Berechnung. Eine einfachere Lösung dieses Problems, ist eine Manipulierung der Grenzen in den entsprechenden Zeitreihen. Dieser Umstand wird bereits vor Beginn der Optimierung umgesetzt. Der Codeausschnitt 4.5 garantiert, dass bei Veränderung der Leistungsgrenzen sich diese allenfalls innerhalb des maximal erlaubten Leistungsänderungsgradienten befindet. Somit befindet sich der Fahrplan stets innerhalb der erlaubten technischen Leistungsgrenzen.

```
for k in 2:T
    delta_min = power_online_min[k] - power_online_min[k-1]
      delta_max = power_online_max[k] - power_online_max[k-1]
      if (delta_max > power_gradient)
            power_online_max[k] = power_online_max[k-1] + power_gradient
      end
      if (delta_min < -power_gradient)
```

```
            power_online_min[k] = power_online_min[k-1] - power_gradient
        end
    end
```

Codeausschnitt 4.5: Anpassung der technischen Mindest- und Maximalleistung an den maximalen Leistungsänderungsgradienten.

Es werden die beiden Zeitreihen für die technische Mindest- und Maximalleistung korrigiert. Die Anpassung erfolgt auf Basis des Leistungsänderungsgradienten. Die Schleife wird solange abgearbeitet, bis alle Leistungssprünge kleiner oder gleich des maximalen Leistungsänderungsgradienten sind. Als Beispiel soll Tabelle 4.7 die Manipulation veranschaulichen.

Tabelle 4.7: Manipulation der Zeitreihen für die Einhaltung der technischen Mindest- und Maximalleistung.

Anpassung der technischen Mindest- und Maximalleistung									
Zeitpunkt	$k-1$	k	$k+1$	$k+2$	$k+3$	$k+4$	$k+5$	$k+6$	$k+7$
P_min	250	250	250	300	300	300	250	250	250
P_min neu	250	250	255	300	300	300	255	250	250
P_max	420	420	420	300	300	300	420	420	420
P_max neu	420	390	345	300	300	300	345	390	420

4.3.5 elementare Funktionen

In diesem Abschnitt werden die grundlegenden Funktionen erläutert, die implementiert sind. Dabei spielt der Faktor Durchlaufzeit und Arbeitsspeicherbedarf eine essentielle Rolle, da diese Funktionen sehr häufig aufgerufen werden. Deshalb ist es von Bedeutung, dass die folgenden Funktionen nicht zwangsläufig leicht verständlichen, aber dafür schnell ausführbaren Charakter besitzen.

4.3.6 GetStateFromPower(...)

In der Funktion *GetStateFromPower(...)* wird ein Leistungswert übergeben und der entsprechende Zustand als Knotenpunkt zurückgeliefert. Im Prinzip wäre für die Ermittlung des entsprechenden Knotenpunktes zu einem Leistungswert eine Suchfunktion über eine Schleife sinnvoll und einfach verständlich. Allerdings wird diese Funktion während der Optimierung sehr häufig aufgerufen und eine häufig aufgerufene Schleife würde negative Auswirkungen auf die Berechnungsdauer haben. Durch eine Umformulierung, wie in Codeausschnitt 4.6 zu sehen ist, kann der jeweilige Zustand über eine mathematische Operation direkt berechnet werden.

```
function    GetStateFromPower(power)
    if (power >= power_online)
        return  power - (power_max - state_max)
    elseif (power == power_min)
        return  state_min
```

```
        else
          k = 2
            while (power_levels[k] < power_online)
                if (power == power_levels[k])
                    return  k
                end
                k += 1
            end
        end
        return  -1
    end
```

Codeausschnitt 4.6: Die Funktion GetStateFromPower(...) implementiert eine mathematische Beziehung zwischen Leistungswert und dem korrelierenden Knotenzustand.

Die Voraussetzung für den korrekten Ablauf der Funktion ist die Konvertierung der Leistungswerte in ein Integer Format, welches keine Gleitkommazahlen zulässt. Die Konvertierung wurde bereits bei Programmstart realisiert. Die erste Abfrage ermittelt, ob der Leistungswert oberhalb oder gleich der technischen Mindestleistung liegt. In dem Fall lässt sich der Knotenpunkt anhand einer Grundrechnung einfach ermitteln. Ein Beispiel verdeutlicht den Sachverhalt:

- Angenommener Leistungswert: 312 MW
- technische Maximalleistung: 420 MW
- maximale Anzahl an Zuständen: 190
- Zustand: 312–(420–190)=82 entspricht 312 MW

Im Falle von Leistung 0 ist die Lösung auch eindeutig. Es verbleiben somit lediglich die Fälle, in denen sich die Leistung unterhalb der technischen Mindestleistung und oberhalb von 0 MW befindet, also der Block sich in einem Rampenverlauf befindet. Die Ermittlung dieser Zustände passiert über eine Schleife. Durch diesen Sachverhalt wird die Rechenzeit der Optimierung signifikant reduziert. Die Funktion *GetStateFromPower(...)* ist auf die Art implementiert, dass bei Änderung der Eingabeparameter der Code nicht adaptiert werden muss.

4.3.7 GetOptimalPower(...)

In der Funktion *GetOptimalPower(...)* wird die optimale Leistung zum jeweiligen Zeitpunkt berechnet und zurückgeliefert. Sie ist die Grundlage für die Entscheidung, welchen Leistungswert die Optimierung als am wahrscheinlichsten selektiert. Als Basis wird die Kostengleichung nach Gl. (4.39) herangezogen.

$$g_t = p_{market,t} \cdot P_t - \left(p_{fuelallin,t} \cdot P_{thermisch,t} + p_{O\&M,t} + p_{additional,t} \cdot P_t \right) \tag{4.39}$$

Die thermische Brennstoffwärmeleistung wird durch die quadratisch inverse Produktionsfunktion ersetzt und die Gleichung nach der Leistung P_t abgeleitet, um zu den Grenzkosten nach Gl. (4.41) zu gelangen.

$$\frac{\mathrm{d}g_t}{\mathrm{d}P_t} = p_{market,t} - \left(p_{fuelallin,t} \cdot (b + 2c \cdot p_t) + p_{additional,t}\right) \tag{4.40}$$

Um den optimalen Leistungswert P_t als lokales Extremum zu ermitteln, wird die Gleichung 0 gesetzt und nach P_t aufgelöst, was auf Gl. (4.41) führt.

$$P_t = \frac{\left(\frac{p_{market,t} - p_{additional,t}}{p_{fuelallin,t}}\right) - b}{2c} \tag{4.41}$$

Durch die Gleichung kann nun zu jedem Zeitpunkt t der zugehörige optimale Leistungswert P_t berechnet werden. Der Optimierungsalgorithmus wird versuchen den optimalen Leistungswert P_t stes einzuhalten, sofern die technischen Restriktionen nicht verletzt werden.

4.3.8 CalculateNodeProfit(...)

Die Funktion *CalculateNodeProfit(...)* berechnet den Profit bzw. die Kosten die in einem Knotenpunkt auf Basis von Gl. (4.42) entstehen.

$$C_t = \left[(p_{market,t} - p_{additional,t}) \cdot P_t - p_{fuelallin,t} \cdot P_{thermisch,t}\right] \cdot \Delta t - (p_{O\&M,t \cdot \Delta t}) \tag{4.42}$$

Die Berechnung wird durchgeführt, sobald die elektrische Leistung größer 0 MW ist, andernfalls werden die Kosten in dem Punkt auf 0 gesetzt. Die Startkosten werden in der Funktion *GetPathStates(...)* berücksichtigt und ergänzt.

4.3.9 RampingAtBeginning(...)

Die Funktion *RampingAtBeginning(...)* behandelt jene Fälle, bei denen der Kraftwerksblock nicht vom Stillstand heraus startet. Sie wird dann aufgerufen, wenn sich die Optimierung während der Laufzeit nahe dem Startschritt T_0 und der Startzustand ungleich dem Stillstand ist. Im Prinzip wird der aktuelle Leistungswert des Knotenpunkts mit dem des Startpunktes verglichen und abgefragt, ob es sich um einen Rampenverlauf handelt. Dadurch wird der Startzustand mit den zeitlich darauffolgenden Zuständen kombiniert, um einen korrekten und stetigen Fahrplanverlauf sicherzustellen. Es werden folgende Fälle behandelt:

- Eine Kaltstartrampe ist aktuell in Vorgang
- Eine Warmstartrampe ist aktuell in Vorgang
- Eine Heißstartrampe ist aktuell in Vorgang
- Eine Abfahrrampe ist aktuell in Vorgang
- Der Kraftwerksblock ist im Online Status und sollte nach Möglichkeit heruntergefahren werden.

- Der Kraftwerksblock ist im Stillstand und sollte nach Möglichkeit hochgefahren werden.
- Der Kraftwerksblock wurde soeben hochgefahren und muss die minimale Betriebsdauer einhalten, bevor er heruntergefahren werden darf.

4.3.10 GetOptimalSolution(...)

Die Funktion *GetOptimalSolution(...)* ist für die Lösung des Optimierungsproblems zuständig. Sie basiert auf dem Verfahren der rückwärts dynamischen Programmierung mit rekursivem Charakter. Als Rückgabewert liefert die Funktion den optimalen maximalen Profit für die eingegebenen Parameter. In Abbildung 4.5 ist der prinzipielle Ablauf der Funktion anhand eines Flussdiagramms dargestellt. Durch den rekursiven Charakter benötigt die Funktion eine Abbruchbedingung, um nicht in einer Endlosschleife zu enden. Die Rekursion wird beendet, sobald der Startzeitpunkt T_0 erreicht wurde.

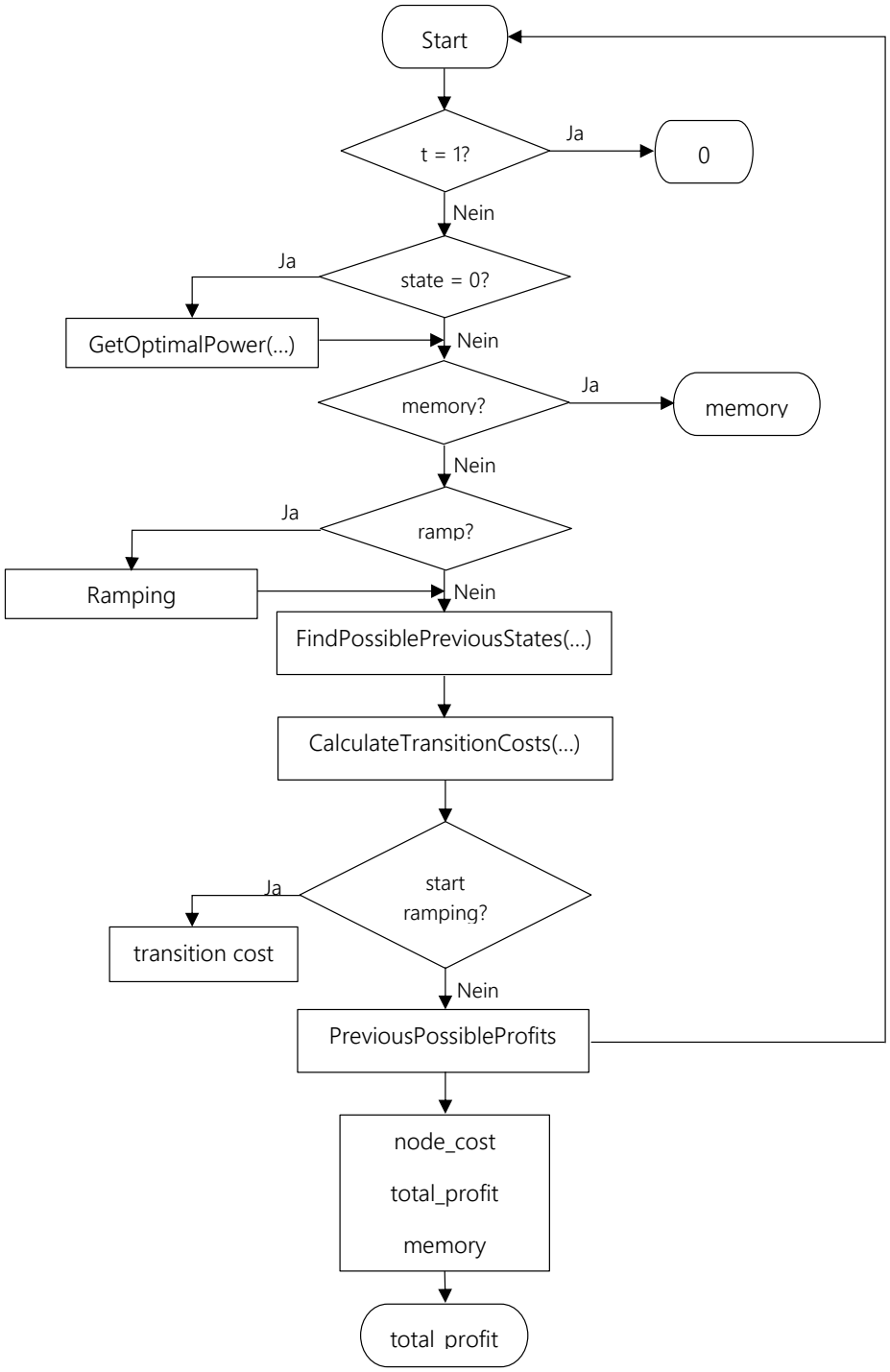

Abbildung 4.5: Schematischer Ablauf der Funktion GetOptimalSolution(...), die für die Lösung des Optimierungsproblems zuständig ist.

Der Beginn der Rekursion muss mit einem Knotenpunkt fixiert werden. Sobald die Funktion zum ersten Mal aufgerufen wird, existiert noch kein gültiger Zustand, von wo aus begonnen werden kann. Die Optimierung kann mit jedem beliebigen Knotenpunkt begin-

nen. Dieses Verhalten ist unerwünscht, da es viele verschiedene Lösungspfade zur Folge hat. Deshalb wird der optimale Initialzustand über die Funktion *GetOptimalPower(...)* anhand von Codeausschnitt 4.7 berechnet.

```
if (state_t == 0)
      state_t = GetClosestStateFromPower(GetOptimalPower(time_step_t))
end
```

Codeausschnitt 4.7: Berechnung des Initialzustandes über GetOptimalPower(...).

Für alle weiteren Betrachtungen wird davon ausgegangen, dass sich die Optimierung beliebig zwischen Start- T_0 und Endzeitpunkt T befindet und ein gültiger Vorgänger bzw. Nachfolger existiert. Im nächsten Programmschritt wird überprüft, ob für den selektierten Knotenpunkt bereits eine optimale Lösung berechnet wurde. Dies beschleunigt den Programmablauf immens, da es zu keiner mehrfachen Abarbeitung des selben Problems führt, sondern die gespeicherte Lösung[7] weiterverwendet wird.

```
if (path_memory[state_t, time_step_t] != 0)
      return  path_memory[state_t, time_step_t]
end
```

Codeausschnitt 4.8: Memoization: Falls bereits eine Lösung für diesen Knotenpunkt existiert, wird diese nicht erneut berechnet, sondern auf die bereits bestehende Lösung zurück gegriffen.

Eine Matrix speichert alle bereits berechneten Teillösungen. Falls die Matrix bereits einen Eintrag ungleich 0 verzeichnet, wurde bereits eine Lösung für den zu untersuchenden Knotenpunkt in berechnet und kann somit darauf zurückgegriffen werden. Codeausschnitt 4.8 bricht die Funktion sofort ab, indem sie den bereits berechneten Profit als Rückgabewert liefert.

Danach wird erfasst, ob aktuell ein Rampenverlauf stattfindet. Dies ist für die Berechnung von Vorzuständen von Bedeutung, da während einer Rampe der Vorgänger bzw. Nachfolger eindeutig definiert ist. Im nächsten Schritt werden durch die Funktion *FindPossiblePreviousStates(...)* alle möglichen Vorgänger erfasst.

```
previous_possible_states = FindPossiblePreviousStates(time_step_t, state_To,
  state_t, ramp_t)
```

Codeausschnitt 4.9: Ermittlung aller möglichen Vorzustände zum aktuellen Knotenpunkt durch die Funktion FindPossiblePreviousStates(...).

Codeausschnitt 4.9 liefert eine Liste mit allen möglichen Vorgängern für den aktuellen Zustand. Im nächsten Abschnitt wird der Ablauf von *FindPossiblePreviousStates(...)* detailliert erläutert. Nachdem alle potentiellen erlaubten Vorzustände ermittelt wurden, werden alle entsprechenden Kosten pro Übergang über *CalculateTransitionCost(...)* berechnet. Die einzige Art an Kosten, die durch einen Übergang entstehen können, sind Startkosten[8].

In Codeausschnitt 4.10 wird mit den vorhin ermittelten Vorgängern die Rekursion ausgeführt. Für jeden erlaubten Vorgänger wird die Funktion *GetOptimalPower(...)* mit den entsprechenden Parametern aufgerufen.

[7] Memoization

[8] Die Startkosten sind unterschiedlich und hängen vom jeweiligen Starttyp ab.

```
@inbounds previous_possible_profits[k] = GetOptimalSolution(time_begin,
  time_step_t - 1, state_To, previous_possible_states[k], ramp_t) -
  transition_cost
```

Codeausschnitt 4.10: Rekursion über alle möglichen Vorzustände, um den optimalen Pfad zu ermitteln.

Das Makro *@inbounds* unterdrückt die Überprüfung von Array-Grenzen durch den Compiler um eine schnellere Abarbeitung zu gewährleisten. Für jeden Vorgänger vom aktuellen Knotenpunkt existiert ein eigenständiger Pfad mit unterschiedlichem Profit. Dieser Pfad muss für jeden Vorgänger eigens berechnet werden, indem *GetOptimalSolution(...)* erneut aufgerufen wird, allerdings für den Vorgänger als aktuellen Knotenpunkt. Dadurch lässt sich bereits anschaulich verstehen, dass sich der Algorithmus durch den gerichteten Graphen, bestehend aus Array-Knoten, durcharbeitet. In den letzten Schritten wird der generierte Profit im aktuellen Punkt über *CalculateNodeProfit(...)* berechnet.

Der Algorithmus arbeitet rückwärts dynamisch vom aktuellen Zeitpunkt T zum Startzeitpunkt T_0. Bis der Startzeitpunkt[9] T_0, erreicht wurde, stehen noch keine Profite fest, da die durchlaufenen Pfade noch keine Informationen über den kumulierten Deckungsbeitrag enthalten. Es existiert ein großes Konstrukt an möglichen Lösungspfaden. Erst nach dem Auflösen der Rekursion[10] werden alle Werte berechnet, rückgegeben und entsprechend ausgewertet.

```
total_profit = maximum(previous_possible_profits + node_profit)
```

Codeausschnitt 4.11: Selektion des optimalen Vorzustandes aus der Liste der möglichen Vorzustände.

Durch Codeausschnitt 4.11 wird stets der profitabelste Lösungspfad selektiert. Wie eingangs bereits erläutert wurde, kann jeder Knotenpunkt mehrere Vorgänger haben mit unterschiedliche Übergangskosten. Anhand von Maximalbildung der Einträge, wird der Vorgänger mit dem maximalen Ertrag gespeichert und zum optimalen Lösungspfad hinzugefügt.

```
@inbounds path_memory[state_t, time_step_t] = total_profit
```

Codeausschnitt 4.12: Speicherung des berechneten optimalen Profits für den jeweiligen Knotenpunkt.

Letztlich muss der beste Pfad bis zum aktuellen Knotenpunkt über Codeausschnitt 4.12 gespeichert werden. Die Funktion liefert den maximalen Profit bis zum aktuellen Knotenpunkt zurück.

4.3.11 FindPossiblePreviousStates(...)

Die Funktion *FindPossiblePreviousStates()* hat die Aufgabe alle erlaubten Vorzustände zum aktuellen Zustand zu ermitteln und rückzuliefern. Ihr schematischer Ablauf ist in Abbildung 4.6 dargestellt.

[9] Der Startzeitpunkt T_0 ist gleichzusetzen mit der Abbruchbedingung der Rekursion.
[10] Auflösen der Rekursion entspricht dem vorwärts Durchlauf der dynamischen Programmierung.

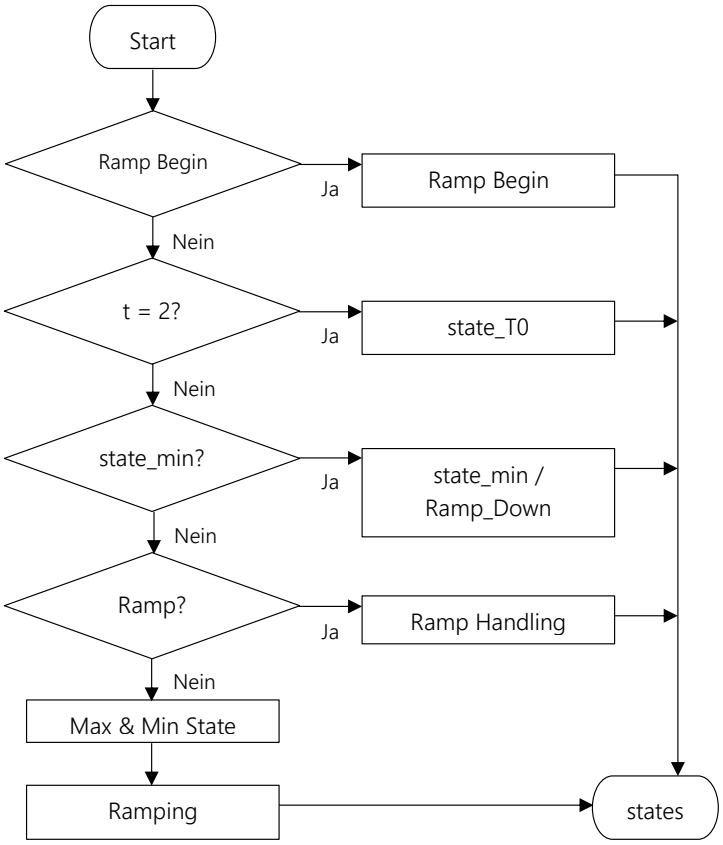

Abbildung 4.6: Schematischer Ablauf der Funktion FindPossiblePreviousStates(...) zur Auffindung erlaubter Vorzustände.

Bei jedem Aufruf wird durch Codeausschnitt 4.13 eine Liste mit möglichen Vorzuständen erstellt, die zu Beginn leer ist und abhängig von den Randdaten mit Zuständen befüllt wird.

```
previous_possible_states = Array{Int64}(0)
```

Codeausschnitt 4.13: Definition des Arrays für mögliche Vorzustände.

Falls sich der Algorithmus nahe dem Startzeitpunkt T_0 befindet, muss überprüft werden, ob in der Vergangenheit eine Rampe begonnen wurde. Dies zeichnet sich darin aus, dass der Startzustand ungleich vom Stillstand ist und einen Leistungswert größer 0 MW besitzt[11]. Die Adaption passiert in der Funktion *RampingAtBeginning(...)* in Codeausschnitt 4.14, welche direkt den korrekten Zustand zurückliefert.

```
if (time_step_t < (@LENGTH_RAMP_COLD() + 1))
    ramp_beginning = RampingAtBeginning(time_step_t, state_To, state_t)
      if (ramp_beginning != -1)
          push!(previous_possible_states, ramp_beginning)
          return  previous_possible_states
      end
```

[11]Dafür muss der Leistungswert einem Eintrag in einer der definierten Rampenvorgänge gleichen.

```
end
```

Codeausschnitt 4.14: Ermittlung, ob sich der Kraftwerksblock zu Beginn in einem Rampenverlauf befindet. Gegebenenfalls erfolgt eine Adaption der erlaubten Vorzustände.

Codeausschnitt 4.16 garantiert den korrekten Übergang zwischen festgelegten Startzustand zum Zeitpunkt T_0 und dem berechneten optimalen Nachfolger.

```
if (time_step_t == 2)
push!(previous_possible_states, state_To)
    return previous_possible_states
end
```

Codeausschnitt 4.15: Vorzustand ist bereits festgelegt durch den Startzustand, falls die Rekursion einen Zeitschritt vor dem Startzeitpunkt angelangt ist.

In den beiden darauffolgenden Abfragen wird erfasst, ob sich der Kraftwerksblock zurzeit im Stillstand oder in einem Rampenverlauf befindet. Da es in diesen Fällen nur einen oder zwei erlaubte Vorzustände gibt, muss deren Abarbeitung anhand von Abfragen und Schleifen abgehandelt werden.

In allen anderen Fällen befindet sich der Kraftwerksblock innerhalb der technischen Mindest- und Maximalleistung. Somit ist eine Leistungsänderung innerhalb der maximalen Leistungsänderungsgradienten und das Ende eines Rampenverlaufs erlaubt. Für die Einhaltung der Leistungsänderung wird durch Codeausschnitt **??** vorerst ermittelt, in welchem Leistungsband sich die Leistung für den nächsten Zustand bewegen darf.

```
  @inbounds power_t = power_levels[state_t]
  @inbounds power_previous_min = max(power_online_min[time_step_t - 1],
power_t - power_gradient)
  @inbounds power_previous_max = min(power_online_max[time_step_t - 1],
power_t + power_gradient)
  min_allowed_state = GetClosestStateFromPower(power_previous_min)
max_allowed_state = GetClosestStateFromPower(power_previous_max)
```

Codeausschnitt 4.16: Ermittlung des erlaubten Leistungsbandes unter Einhaltung des maximalen Leistungsänderungsgradienten.

Die Ermittlung der optimalen Leistung für den Vorzustand in Codeausschnitt 4.17 basiert auf der Berechnung der Grenzkosten, welche in der Funktion *GetOptimalPower(...)* implementiert ist.

```
optimal_state = GetClosestStateFromPower(GetOptimalPower(time_step_t - 1))
```

Codeausschnitt 4.17: Hinzufügen des optimalen Zustandes in das Array der erlaubten Vorzustände.

Andernfalls wird versucht dem optimalen Zustand so nahe wie nur möglich zu kommen[12]. In diesem Fall werden die minimale und maximal erlaubten Knotenpunkte in die Liste für Vorgänger, wie in Codeausschnitt 4.18 zu sehen ist, hinzugefügt.

```
push!(previous_possible_states, min_allowed_state)
push!(previous_possible_states, max_allowed_state)
```

Codeausschnitt 4.18: Falls der optimale Zustand außerhalb des Leistungsbandes liegt, darf der erlaubte Vorzustand maximal an der unteren bzw. oberen Grenze des Bandes liegen.

[12]Die Leistungsgrenzen durch die technische Mindest- und Maximalleistung, sowie die Änderung durch den Leistungsgradienten müssen allenfalls eingehalten werden.

Zusätzlich muss auch das Ende eines Rampenverlaufs erlaubt sein, da jederzeit eine Rampe gefahren werden darf. Codeausschnitt 4.19 fügt die Endwerte jedes Rampentyps in die Liste für erlaubte Vorgänger auf.

```
push!(previous_possible_states, GetStateFromPower(ramp_hot_values[
@LENGTH_RAMP_HOT() + 1]))
push!(previous_possible_states, GetStateFromPower(ramp_warm_values[
@LENGTH_RAMP_WARM() + 1]))
push!(previous_possible_states, GetStateFromPower(ramp_cold_values[
@LENGTH_RAMP_COLD() + 1]))
```

Codeausschnitt 4.19: Hinzufügen der Endwerte der Rampenverläufe, um einen Rampenverlauf jederzeit zu erlauben.

Der Endwert einer Abfahrrampe ist 0 und deshalb werden im Falle eines Stillstands immer zwei Vorgänger erlaubt, nämlich der Leistungswert der Abfahrrampe vor Stillstand und der Knotenpunkt für Stillstand selbst.

4.3.12 GetPathStates(...)

Nachdem der optimale Pfad durch die Funktion *GetOptimalSolution(...)* bereits ermittelt wurde, wird anhand von *GetPathStates(...)* der Pfad erneut durchlaufen und auf seine Gültigkeit überprüft. Dabei wird garantiert, ob minimale Stillstands- und Betriebsdauern eingehalten werden und insbesondere Rampenverläufe aufgrund ihres Typs korrekt abgearbeitet werden. Der Grundgedanke der Funktion *GetPathSates(...)* beruht darauf, dass der optimale Pfad überprüft wird und bei Verletzung der Rahmenbedingung automatisch den zweitbesten Pfad selektiert wird. Dies wird dadurch veranlasst, indem bestimmte Übergänge als nichtzulässig angenommen werden und ihre Übergangskosten auf unendlich gesetzt werden.

Der schematische Ablauf von *GetPathSates(...)* ist in Abbildung 4.7 zu sehen.

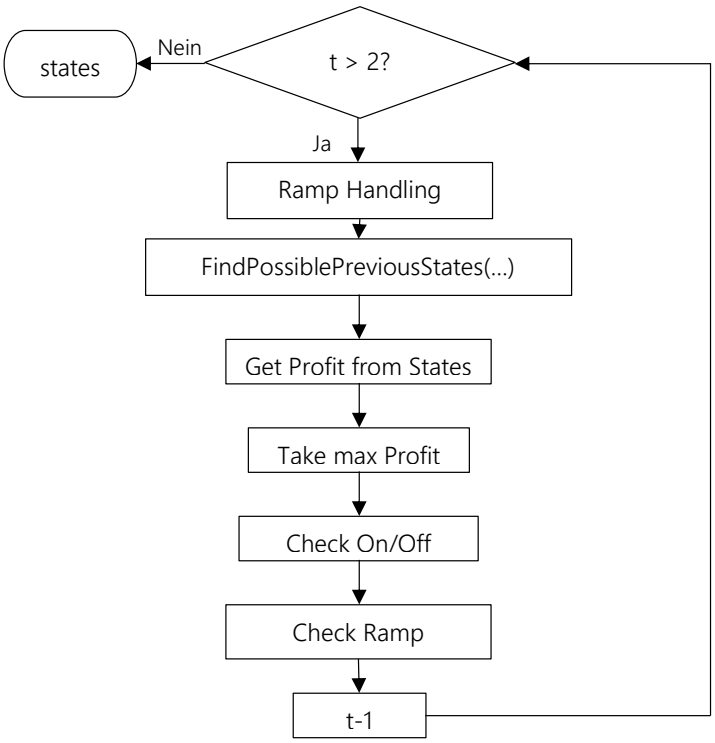

Abbildung 4.7: Schematischer Ablauf der Funktion GetPathStates(...) zur Erfassung des besten Lösungswegs. Im weiteren Verlauf wird der ermittelte Lösungspfad nochmals auf seine Gültigkeit überprüft, ob dieser auch alle Randbedingungen genügt.

Der Pfad wird erneut rückwärts, also vom aktuellen Zeitpunkt T, bis zum Startzeitpunkt T_0 in einer Schleife durchlaufen. Die Schleife beharrt an einem Zeitschritt, an dem eine Verletzung der Rahmenbedingungen festgestellt wurde solange, bis ein gültiger Übergang gefunden wurde. Bei Rampen trifft dies inbesondere zu, da in dem Fall mithilfe von temporären Hilfsvariablen alle Randdaten einer Rampe, wie zum Beispiel Beginn, Ende, Art und Dauer der Rampe gespeichert werden müssen, um zum jeweiligen Anfangs- bzw. Endzeitpunkt der Rampe springen zu können. Aus diesem Grund, muss erneut eine Suche nach möglichen Vorgängern bei fehlerhaften Stellen gemacht werden, um einen optimalen und gültigen Pfad sicherzustellen.

Im Standardfall wird anhand von Codeausschnitt 4.20 der optimale Knotenpunkt zu einem Zeitpunkt über den maximalen Profit erfasst, der bereits vorher durch *GetOptimalSolution(...)* berechnet wurde.

```
for j in 1:length(possible_previous_states)
      temp_profit[j] = path_memory[possible_previous_states[j], time_step_t-1]
end
path_states[1, time_step_t-1] = possible_previous_states[indmax(temp_profit)]
```

Codeausschnitt 4.20: Selektion des Knotenpunkts mit dem größten Profit.

Die Einhaltung von minimaler Stillstands- und Betriebsdauer wird anhand von Zählern im Hintergrund realisiert. Bei Verletzung wird der Kraftwerksblock gezwungen für eine bestimmte Dauer im Stillstand zu verweilen, bevor eine Rampe gestartet werden darf.

Sobald eine Rampe aktiv wird, werden Anfang, Ende, Dauer und Art der Rampe gespeichert. Nachdem die Rampe ihren Endwert erreicht hat, wird überprüft, ob alle Parameter den Modelleinschränkungen genügen.

Der Algorithmus wird primär einen Heißstart bevorzugen, da dieser mit den geringsten Startkosten verbunden ist und am kürzesten Zeit benötigt. Dabei kann es zu Nichteinhaltung der Stillstandsdauer kommen, da ein Heißstart vor dem Start eine maximal definierte Zeitdauer hat, in der der Kraftwerksblock still stehen darf. Sobald diese Zeit überschritten wird, wäre ein Heißstart unzulässig und ein Warmstart als nächstes zu bevorzugen. Im selbigen Fall kann dies zu einem Kaltstart führen, der unabhängig von der vorangegangen Stillstandsdauer ausgeführt werden kann[13].

Im Allgemeinen funktioniert die Prüfroutine von *GetPathStates(...)* nach dem zuvor erläuterten Prinzip. Falls kein Heißstart zulässig ist, wird je nach bestehender Stillstandsdauer ein Warm- bzw. Kaltstart ausgeführt. Der Knotenpunkt, der den Heißstart beginnen lässt, wird durch einen Übergang, der mit unendlich hohen Kosten verbunden ist, gesperrt. Dadurch sucht sich der Algorithmus automatisch den nächstbesseren Rampenverlauf. Abbildung 4.8 demonstriert dieses Verhalten dadurch, dass primär ein Heißstart präferiert wird. Bei der Überprüfung wird festgestellt, dass die angelaufene Stillstandszeit zu groß ist und der Kraftwerksblock nicht heiß genug für einen Heißstart ist. Deshalb wird als Alternative ein Warmstart vorgenommen, welcher sich innerhalb der zulässigen Grenzen befindet.

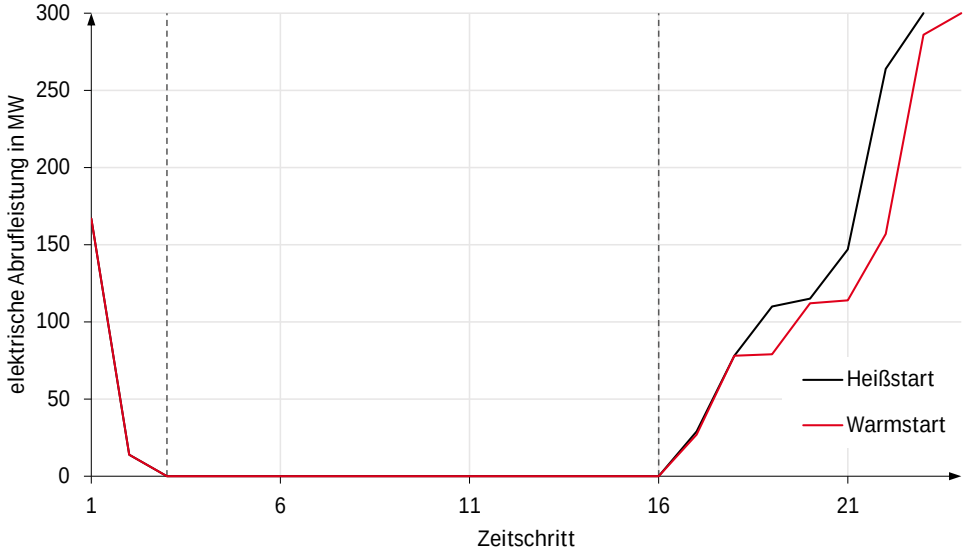

Abbildung 4.8: Die zuvor angelaufene Stillstandszeit entscheidet über die Art der Startrampe. Gegebenfalls kann es notwendig sein, den Typ der Startrampe zu adaptieren, damit die Randbedingungen des Modells nicht verletzt werden.

[13]Solange Heiß- und Warmstart bereits als unzulässig deklariert wurden.

An dieser Stelle sei nochmals hervorgehen, dass sich der beschriebene Ansatz zur Lösung von Fehlerstellen zwar in Worten einfach fassen lässt, aber mit einem hohen Grad an Komplexität in der Implementierung verbunden ist.

4.3.13 GetPowerDispatch(...)

Die Funktion *GetPowerDispatch(...)*, welche in Codeausschnitt 4.21 zu sehen ist, zieht als Quelle die bereits ermittelten Pfadzustände heran und ordnet ihnen die entsprechende elektrische Abrufleistung zu.

```
function     GetPowerDispatch(path_states)
      power = zeros(Int64, 1, T)
      for time_step_t in 1:T
          power[time_step_t] = power_levels[path_states[1, time_step_t]]
      end
      return  power
end
```

Codeausschnitt 4.21: Die Funktion GetPowerDispatch(...) dient zur Ermittlung der elektrischen Leistung für die entsprechenden Pfadzustände.

4.3.14 GetPathProfit(...)

Da nach Vollendung der Rekursion der maximale Profit bekannt ist, aber nicht der Profit zu einem bestimmten Zeitpunkt, berechnet *GetPathProfit(...)* alle Profite innerhalb des festgelegten Zeitraums zu jedem Zeitpunkt t. Als Basis werden die ermittelten Pfadzustände herangezogen. Dabei ist der Knotenprofit bereits durch den Knoten selbst festgelegt. Die Startkosten in Form von Übergangskosten müssen noch über eine Abfrage ergänzt werden.

```
function     GetPathProfit(path_states)
      path_profit = Array{Int32}(1,T)
      path_profit[1, 1] = 0

    for time_step_t in 2:T
          transition_cost = 0
          if (path_states[1, time_step_t - 1] == state_min && path_states[1,
  time_step_t] == GetStateFromPower(ramp_hot_values[2]))
              transition_cost = starting_cost_hot
          elseif (path_states[1, time_step_t - 1] == state_min &&  path_states
  [1, time_step_t] == GetStateFromPower(ramp_warm_values[2]))
              transition_cost = starting_cost_warm
          elseif (path_states[1, time_step_t - 1] == state_min && path_states
  [1, time_step_t] == GetStateFromPower(ramp_cold_values[2]))
              transition_cost = starting_cost_cold
      end
          node_profit = CalculateNodeProfit(time_step_t, path_states[1,
  time_step_t])
          path_profit[1, time_step_t] = round(path_profit[1, time_step_t - 1] +
   node_profit - transition_cost)
      end
      return  path_profit
```

```
end
```

Codeausschnitt 4.22: Ermittlung des Profits des optimalen Pfades.

Codeausschnitt 4.22 erzeugt eine Liste, die alle Knotenpunkte des optimalen Pfades enthält und somit eine exakte Nachverfolgung der elektrischen Abrufleistung ermöglicht. Sie ist die Basis für die Erstellung des Fahrplans für die elektrische Abrufleistung.

4.3.15 Aufbereitung der optimierten Entscheidungsvariablen

Nach dem Lösen des Optimierungsproblems steht als Ergebnis der optimale Fahrplan und der Gesamtprofit über den jeweiligen Zeitraum fest. Als weitere Ausgabevariable sind die variablen Durchschnittskosten von Interesse, die bereits in der Modellbeschreibung erläutert wurden. Sie können anhand von Gl. (4.43) berechnet werden.

$$\begin{aligned} AC_t &= p_{fuelallin,t} \cdot \frac{P_{thermisch,t}}{P_t} + p_{additional} \\ &= p_{fuelallin,t} \cdot \frac{a_{lin} + b_{lin} \cdot P_t}{P_t} + p_{additional} \end{aligned} \tag{4.43}$$

Dabei wird die Berechnung nur durchgeführt, wenn das Kraftwerk in Betrieb, also die elektrische Abrufleistung größer 0 MW ist, da die variablen Durchschnittskosten ansonsten unendlich hoch wären.

Die ermittelten Ergebnisse werden in einer Tabelle per „DataFrames“Format abgespeichert und in eine *.csv* Datei exportiert. Eine *.xls* Datei importiert die Information aus der *.csv* Datei und stellt sie grafisch dar.

5 Ergebnis

Das folgende Kapitel legt die Ergebnisse der zuvor beschriebenen Lösungsverfahren dar. Im Vordergrund stehen die berechneten optimalen Fahrpläne, die man als Lösung der Optimierung erhält und die Berechnungsdauer, die für die Berechnung des Optimums benötigt wird. Ein weiter essentieller Faktor ist die Beurteilung der Performance. Hierfür wird insbesondere dem Zusammenhang zwischen Schrittanzahl[1] und Berechnungsdauer Aufmerksamkeit gewidmet. Ziel ist es einen Überblick über die Ergebnisse zu erhalten, um die Grundlage für die Diskussion und Interpretation in den folgenden Abschnitten zu legen.

5.1 Ergebnis der gemischt-ganzzahlig linearen Programmierung in MatLab

5.1.1 Fahrplan auf Basis von MILP

Nach Lösung des Optimierungsproblems liefert *GUROBI* als Ergebnis einen Fahrplan für die elektrische Abrufleistung des thermischen Kraftwerksblocks.

Abbildung 5.1 zeigt den Fahrplan mit den relevanten Randdaten. Die elektrische Abrufleistung wird in Verbindung mit Marktpreis, Grenzkosten und variablen Durchschnittkosten dargestellt.

Bei hohen Marktpreisen wird versucht die elektrische Abrufleistung maximal zu halten, im regulären Fall gleich der technischen Maximalleistung. Im Fall einer Beschränkung der technischen Maximalleistung, muss auch die elektrische Abrufleistung rechtzeitig verringert werden. Dabei darf die elektrische Abrufleistung zu keinem Zeitpunkt die Grenze der technischen Maximalleistung überschreiten.

Die Einschränkung durch die technische Minimalleistung muss analog zur Maximalleistung eingehalten werden. Diese beiden Parameter spannen das Leistungsband auf, in dem sich die elektrische Abrufleistung verändern darf, solange sich der Kraftwerksblock im Online Status befindet. Im Falle einer Rampe muss dieses Leistungsband verlassen werden.

Sobald der Marktpreis unter die Grenzkosten fällt, wird die elektrische Abrufleistung verringert. Die Höhe der Verringerung der elektrischen Leistung ist abhängig von den

[1] Als Schrittanzahl wird die Anzahl der Schritte bezeichnet, die für den gewählten Zeitraum vorgesehen sind.

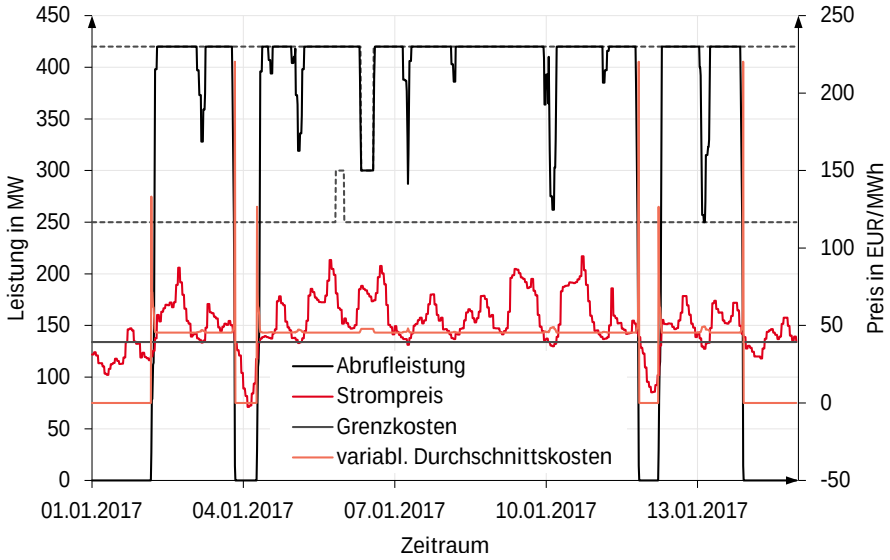

Abbildung 5.1: Fahrplan der elektrischen Abrufleistung auf Basis aller Randdaten, der durch die gemischt-ganzzahlige lineare Programmierung ermittelt wurde.

Grenzkosten zum jeweiligen Zeitpunkt. Bei niedrigen Marktpreisen entscheidet die Optimierung unter Umständen ein Abfahren des Kraftwerksblocks. Dadurch wird der Kraftwerksblock rechtzeitig in den Stillstand gebracht und es fallen keine laufenden Kosten an. Ein erneuter Start in Form einer Rampe verursacht Startkosten, die abhängig von der zuvor angelaufenen Stillstandszeit sind. Für die Optimierung ist dieser Aspekt von Bedeutung, da unter Umständen der Kraftwerksblock länger in Betrieb gehalten wird, um dadurch geringere Stillstandszeiten zu haben, um den Kraftwerksblock möglichst bald wieder starten zu können, welcher mit geringeren Startkosten verbunden ist.

Unter allen Umständen muss der maximale Leistungsänderungsgradient eingehalten werden. Als Ausnahme hierfür ist eine Rampe, die gefahren wird.

Abbildung 5.2 veranschaulicht den kumulierten Deckungsbeitrag des Kraftwerksblocks. Als sehr anschaulich lassen sich die Startkosten im Falle eines Startereignisses darstellen. Im Vergleich zu den korrelierenden Marktpreisen, kann deutlich erkannt werden, dass das Kraftwerk rechtzeitig heruntergefahren wird, um Profitverlust zu vermeiden.

5.1.2 Berechnungsdauer in MILP

Die Berechnungsdauer für die Optimierung mittels dem gemischt-ganzzahligen linearen Verfahren kann stark variieren. Einen großen Einfluss auf die Berechnungsdauer hat das sogenannte maximal erlaubte relative Gap. Das Gap beschreibt den prozentualen Unterschied zwischen oberer und unterer Schranke der Optimierung aufgrund des Branch-and-Bound Näherungsverfahrens. Da es sich um ein Näherungsverfahren handelt, liegt das Ergebnis näher am tatsächlichen Optimum, je kleiner das relative Gap ist. Im Regelfall wird mit einem relativen Gap von 0,1 % gearbeitet. Für Vergleichszwecke und um die Rechenzeit gering zu halten ist das relative Gap mit 2 % festgelegt. Daraus ergeben

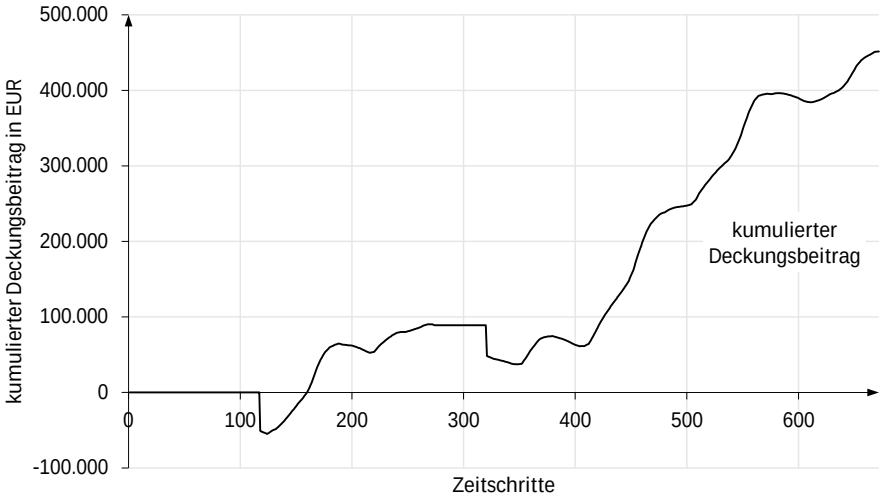

Abbildung 5.2: Kumulierter Deckungsbeitrag des Kraftwerksblocks als Ergebnis der gemischt-ganzzahligen linearen Programmierung.

sich die in Tabelle 5.1 dargestellten Berechnungsdauern und erwirtschafteten Profite für die jeweilige Schrittanzahl.

Tabelle 5.1: Berechnungsdauer der ganzzahlig-gemischten linearen Programmierung.

Zeitraum	Schritte	Dauer in Sek.	Total Profit in EUR
1 Tag	96	4,666	0
1 Woche	672	43,932	449 885
2 Wochen	1344	254,4	983 782
3 Wochen	2016	157	3 413 456
1 Monat	2688	350	5 228 315
5 Wochen	3360	557	5 959 106
6 Wochen	4032	988	6 741 175
7 Wochen	4704	1027	7 245 565
2 Monate	5376	1122	7 245 565
3 Monate	8064	2581	7 653 258

Aufgrund von relativ langen Rechenzeiten wurde für die gemischt-ganzzahlige lineare Programmierung als maximalen Zeitraum drei Monate festgelegt. Auffallend sind die Volatilitäten der Rechendauern. Wie zu Beginn des Abschnitts bereits angesprochen wurde, hängen die Rechendauern von der jeweiligen Marktlage bzw. der Markttiefe[2] des Kraftwerksblocks ab.

[2]Unter Markttiefe versteht man das Verhalten des Kraftwerks aufgrund von gegebenen Marktpreisen. Bei starker Markttiefe befindet sich der Marktpreis in einem Bereich, wo die Schwelle überschritten wird, um das Kraftwerk gerade in Betrieb zu nehmen und das Kraftwerk somit gestartet wird. Schwache Markttiefe bedeutet, dass der Zustand des Kraftwerks aufgrund der vorliegenden Marktpreise entweder Stillstand oder maximale Leistung ist.

In Abbildung 5.3 ist der Zusammenhang zwischen Schrittanzahl und Berechnungsdauer grafisch dargestellt. Die Trendlinie verdeutlicht den nichtlinearen Zusammenhang, was ein bekannter Umstand bei gemischt-ganzzahliger linearer Programmierung ist. Die vorhin erwähnte Thematik bezüglich der Markttiefe des Kraftwerksblocks lässt sich in der Grafik anhand von jenen Punkten, die von der Trendlinie stark abweichen, anschaulich erkennen. Tabelle 5.2 zeigt, dass bei niedrigen oder sehr hohen Strompreisen, die durch den Strompreisfaktor manipuliert werden können, die Rechendauer geringer ist, da der Kraftwerksblock mit großer Wahrscheinlichkeit still steht oder in Betrieb bei technischer Maximalleistung ist.[3]

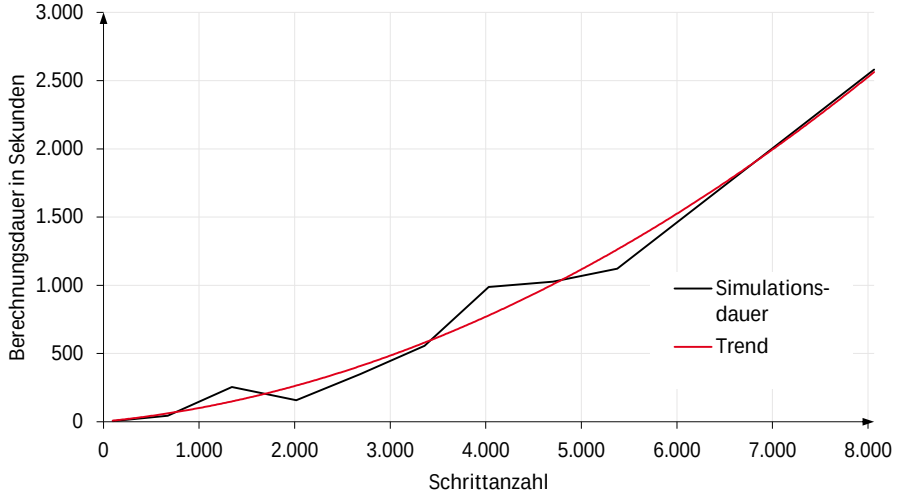

Abbildung 5.3: Berechnungsdauer in Abhängigkeit von der Schrittanzahl. Die Trendlinie verdeutlicht den nichtlinearen Zusammenhang.

Tabelle 5.2: Sensitivitätsanalyse für die Markttiefe des Kraftwerks für einen Zeitraum von sieben Tagen.

Strompreisfaktor in %	Rechendauer in Sek.
90	38,045
95	38,542
100	41,609
110	43,854
120	47,881
130	44,794
150	47,993
200	46,075
300	39,643

[3] In diesem Fall spricht man von schwacher Markttiefe.

5.2 Ergebnis der rückwärts dynamischen Programmierung in Julia

5.2.1 Fahrplan auf Basis von DP

Die rückwärts dynamische Programmierung liefert als Ergebnis einen Fahrplan für den Kraftwerksblock, welcher in Abbildung 5.4 zu sehen ist.

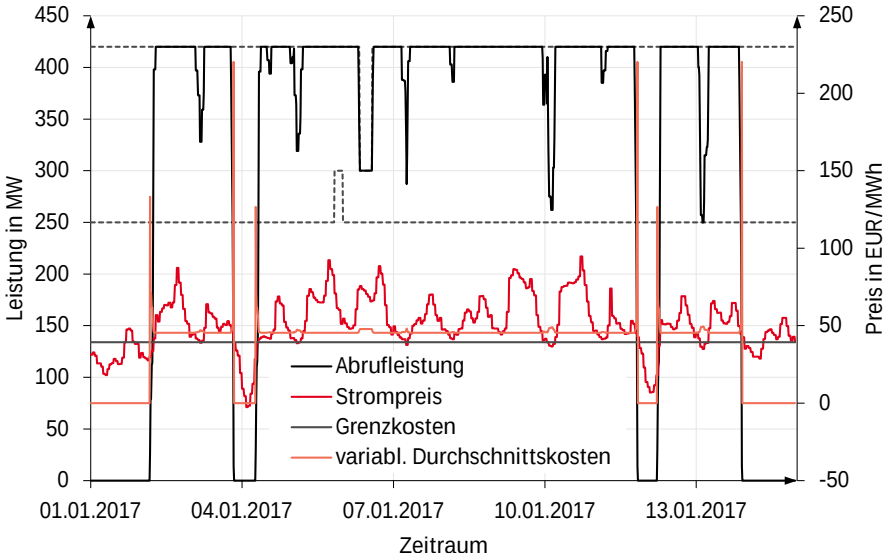

Abbildung 5.4: Fahrplan der elektrischen Abrufleistung auf Basis aller Randdaten, der durch die rückwärts dynamische Programmierung ermittelt wurde.

Der Fahrplan wurde mit den selben Parametern, wie bei der gemischt-ganzzahligen linearen Programmierung berechnet. Die Eigenschaften des Fahrplans bzw. die Einschränkungen sind ident mit jenen der gemischt-ganzzahligen linearen Programmierung. Daraus kann man schließen, dass die Fahrpläne auch ident sind. Die Einhaltung des erlaubten Leistungsbandes, wie es auch in der gemischt-ganzzahligen linearen Programmierung realisiert wurde, ist in Abbildung 5.5 zu sehen. Als weiteres Merkmal, welches in der gemischt-ganzzahligen linearen Programmierung nicht berücksichtigt wurde, wird die Vorgeschichte des Kraftwerksblocks miteinbezogen. Befindet sich der Kraftwerksblock in einem Rampenverlauf bereits vor dem Startzeitpunkt, so muss dieser bis zu seinem Endzustand fertig gefahren und zusätzlich die minimale Betriebsdauer eingehalten werden, bevor er wieder zum Stillstand gebracht werden darf. Dieses Verhalten ist in Abbildung 5.6 ersichtlich.

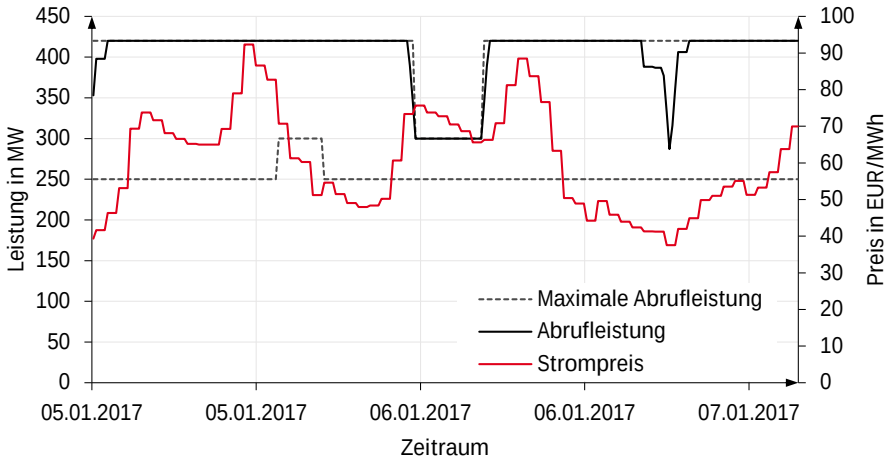

Abbildung 5.5: Verringerung der elektrischen Abrufleistung aufgrund der reduzierten technischen Maximalleistung.

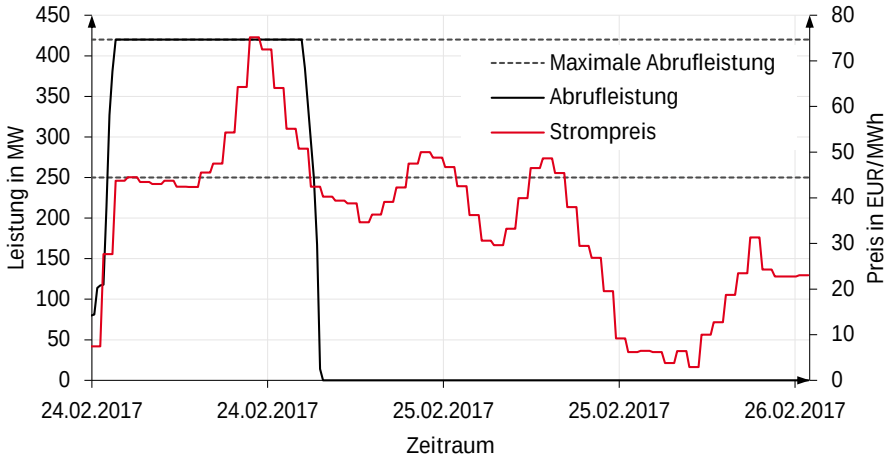

Abbildung 5.6: Fertigstellung eines Rampenverlaufs und Einhaltung der minimalen Betriebsdauer, bevor der Kraftwerksblock heruntergefahren werden darf.

5.2.2 Berechnungsdauer und Arbeitsspeicherbedarf in DP

Die Berechnungsdauer des Modells mit rückwärts dynamischer Programmierung verhält sich in Abhängigkeit der Schrittanzahl immer linear. Die Datenlage hat keinen Einfluss auf die Rechendauer des Optimierungsproblems. Des Weiteren existiert auch kein Duality Gap, wie es in der gemischt-ganzzahlig linearen Programmierung der Fall ist. Dennoch handelt es sich um kein exaktes Optimum, da für die Berechnung Vereinfachungen getroffen werden, um die Rechenzeit gering zu halten. Ein Beispiel für solche Abstraktionen ist das Runden von Daten, um in weiteren Berechnungen einen Zeitvorteil zu generieren. Im nächsten Kapitel wird auf die getroffenen Vereinfachungen näher eingegangen.

Tabelle 5.3 zeigt alle wichtigen Ergebnisse der rückwärts dynamischen Programmierung.

Tabelle 5.3: Berechnungsdauer und Arbeitsspeicherbedarf der rückwärts dynamischen Programmierung.

Zeitraum	Schritte	Dauer in Sek.	Speicher in MB	Total Profit in EUR
1 Tag	96	0,18	13	0
1 Woche	672	0,25	27	455 783
2 Wochen	1344	0,29	45	941 895
3 Wochen	2016	0,30	49	3 455 730
1 Monat	2688	0,35	61	5 287 260
5 Wochen	3360	0,40	83	6 027 900
6 Wochen	4032	0,44	101	6 898 790
7 Wochen	4704	0,48	118	7 376 080
2 Monate	5376	0,59	160	7 360 440
3 Monate	8064	0,86	284	7 773 110
4 Monate	10 752	1,14	422	8 085 080
6 Monate	16 128	1,67	664	8 716 790
8 Monate	21 504	2,15	850	10 415 500
9 Monate	24 192	2,33	953	10 973 900
1 Jahr	35 040	3,72	1515	16 604 600

Im Gegensatz zur gemischt-ganzzahligen linearen Programmierung wurde der Zeitraum für die Berechnung von drei Monaten auf ein Jahr erhöht, da dieser Zeitraum ohne Probleme unter realistischen Zeitbedingungen simulierbar ist.

Die dynamische Programmierung besitzt einen linearen Zusammenhang zwischen Rechenzeit und Schrittanzahl. Abbildung 5.7 veranschaulicht dieses Verhalten.

Ein weiterer Aspekt, der bei der Untersuchung der Ergebnisse von Interesse ist, ist der benötigte Arbeitsspeicher der Optimierung. Für die gemischt-ganzzahlige lineare Programmierung liegen keine Messergebnisse vor. Der Arbeitsspeicherbedarf der rückwärts dynamischen Programmierung lässt sich durch sogenannte Makros in der Entwicklungsumgebung von Julia messen. Abbildung 5.8 zeigt den linearen Zusammenhang zwischen Arbeitsspeicherbedarf und Schrittanzahl bei rückwärts dynamischer Programmierung.

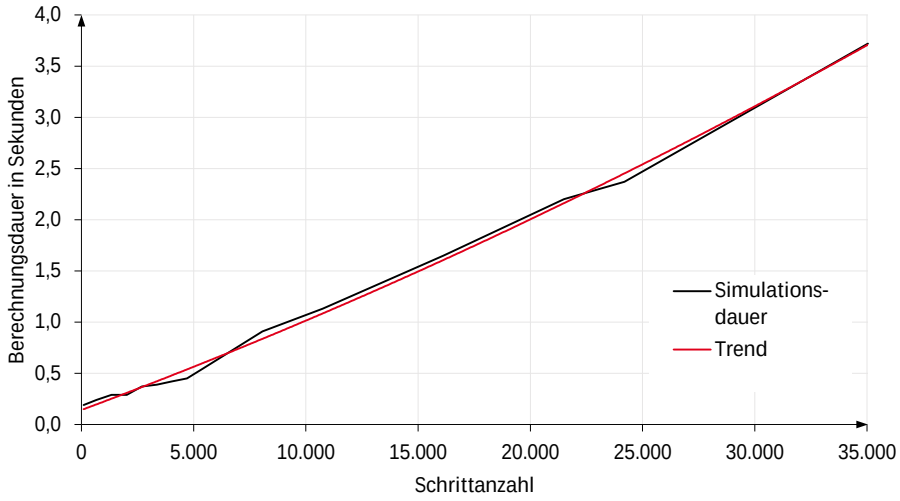

Abbildung 5.7: Berechnungsdauer in Abhängigkeit der Schrittanzahl der rückwärts dynamischen Programmierung. Die Trendlinie verdeutlicht den linearen Zusammenhang.

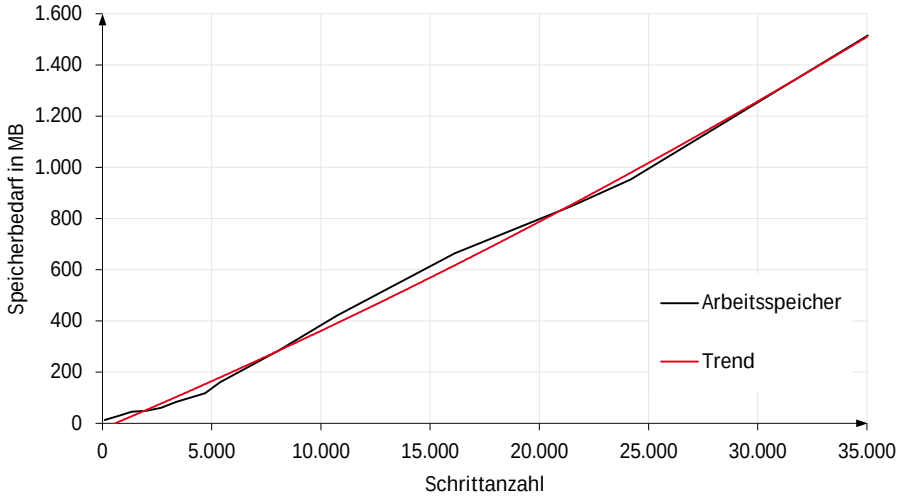

Abbildung 5.8: Arbeitsspeicherbedarf in Abhängigkeit der Schrittanzahl der rückwärts dynamischen Programmierung.

6 Synthese

In diesem Kapitel werden die zuvor dargestellten Ergebnisse miteinander verglichen und eine Aussage hinsichtlich Vor- und Nachteile der jeweiligen Lösungsvariante getroffen. Im ersten Abschnitt werden die Verfahren für die Lösung des Optimierungsproblems verglichen. Im zweiten Abschnitt wird auf die eingangs erwähnte Forschungsfrage, nämlich die Untersuchung, ob Julia in der dynamischen Programmierung eine ähnliche Performance wie Java aufweist, eingegangen.

6.1 Vergleich gemischt-ganzzahlige lineare Programmierung mit rückwärts dynamischer Programmierung

Beide Verfahren zur Lösung von Optimierungsproblemen wurden im Kapitel „analytische Methode“bereits detailliert beschrieben. Tabelle 6.1 fasst die gemessenen Berechnungsdauern zusammen und stellt eine Grundlage für den Vergleich für gemischt-ganzzahlige lineare Programmierung mit rückwärts dynamischer Programmierung dar.

Es steht außer Zweifel, dass die rückwärts dynamische Programmierung einen signifikanten Vorteil in der Berechnungszeit gegenüber der gemischt-ganzzahlig linearen Programmierung besitzt. Vor allem bei steigender Schrittanzahl wird die zeitliche Differenz deutlich größer. Zusätzlich skaliert die Berechnungsdauer bei gemischt-ganzzahlig linearer Programmierung mit steigender Schrittanzahl überproportional mit. Dadurch muss die Größe des Problems klein bzw. die Zeitgranularität möglichst groß gehalten werden.

Die Datenlage hat wesentlichen Einfluss auf die Dauer der Berechnung. Im Zuge dieser Arbeit konnte beobachtet werden, dass je nach Marktpreis die Berechnungsdauer stark variieren kann. In Zeiten von extremen Marktsituationen, beispielsweise durch sehr niedrige oder sehr hohe Marktpreise verursacht, lässt sich das Modell relativ schnell lösen. Der Grund liegt darin, da für den Algorithmus fest steht, dass der optimale Fahrplan entweder Stillstand oder technische Maximalleistung vorschreibt. In Zeiten, wo sich der Marktpreis im Bereich der Grenzkosten befindet, benötigt die Optimierung mehr Zeit, um zu entscheiden, welcher Leistungsschritt der optimalste wäre.[1] Als Beispiel lassen sich die Zeiträume von sieben Wochen und zwei Monaten der gemischt-ganzzahligen linearen Programmierung anschaulich vergleichen. Ihre Berechnungsdauern unterscheiden sich marginal. Ihr errechneter Profit ist völlig ident. Dieser Umstand ist auf einen niedrigen

[1] Starke bzw. schwache Markttiefe

Tabelle 6.1: Vergleich der Berechnungsdauer der Lösungen über gemischt-ganzzahliger linearer Programmierung und rückwärts dynamischer Programmierung.

Auswertung Zeitraum	Schritte	MILP Dauer in Sekunden	DP
1 Tag	96	4,67	0,18
1 Woche	672	43,93	0,25
2 Wochen	1344	254,4	0,29
3 Wochen	2016	257	0,30
1 Monat	2688	350	0,35
5 Wochen	3360	557	0,40
6 Wochen	4032	988	0,44
7 Wochen	4704	1027	0,48
2 Monate	5376	1122	0,59
3 Monate	8064	2581	0,86
4 Monate	10 752	n.v.	1,14
6 Monate	16 128	n.v.	1,67
8 Monate	21 504	n.v.	2,15
9 Monate	24 192	n.v.	2,33
1 Jahr	35 040	n.v.	3,72

Marktpreis zurückzuführen. Der Kraftwerksblock steht in der letzten Woche der zwei Monate still und sein Fahrplan lässt sich in der Optimierung simpel berechnen. Dadurch ist der berechnete Profit ident und die Rechendauer unterscheidet sich nur marginal, obwohl eine Zeitraumdifferenz von einer Woche besteht. Als weiteres bekräftigendes Beispiel lassen sich die Zeiträume von drei Wochen und einem Monat anschaulich vergleichen. Hier ist ein deutlicher Unterschied in der Berechnungsdauer und im errechneten Profit zu erkennen. Das liegt daran, dass der Kraftwerksblock permanent im Betrieb ist und die Optimierung mehr Rechenaufwand benötigt, obwohl wiederum nur eine Zeitraumdifferenz von einer Woche besteht.

Durch Erweiterung des Modells anhand von Rampenverläufen, steigt der Grad an Komplexität des Modells weiter an und dies hat starke Auswirkungen auf die Berechnungsdauer des Modells. Im Vergleich zu rückwärts dynamischer Programmierung, wo Erweiterungen häufig nur einen geringeren Mehraufwand in der Berechnung erfordern. Bestimmte Anforderungen lassen sich nicht über gemischt-ganzzahlige lineare Programmierung formulieren oder nur sehr schwierig implementieren. Als Beispiel sei die Kombination aus einem Kaltstart und einer reduzierten technischen Maximalleistung angeführt. Falls sich die technische Maximalleistung unterhalb des Leistungsendwerts eines Kaltstarts befindet, ist dieser nicht erlaubt und somit auch nicht fahrbar. Allerdings könnte dies sehr wohl eine optimale Lösung darstellen.

In der rückwärts dynamischen Programmierung lässt sich dieser Umstand implementieren, in dem der Kaltstart ab der technischen Maximalleistung limitiert wird und diese Zustände zulässig sind. Abbildung 6.1 veranschaulicht diesen Sachverhalt.

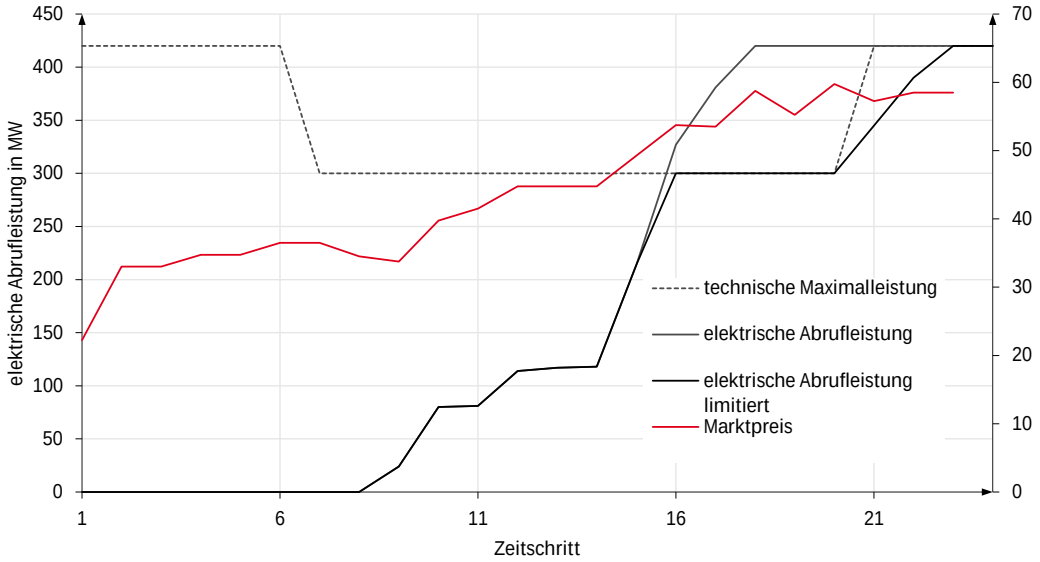

Abbildung 6.1: Abänderung der Kaltstartrampe aufgrund der Limitierung der elektrischen Abrufleistung durch die technische Maximalleistung.

Aufgrund der zuvor angelaufenen Stillstandszeit ist nur eine Kaltstartrampe eine Option. Allerdings wird das Ende des Rampenverlaufs durch die technische Maximalleistung beschränkt. Die elektrische Abrufleistung muss unter allen Umständen unterhalb dieser Limitierung sein und kann somit den Kaltstartvorgang nicht beenden und verweilt an der technischen Maximalleistung. Nach Aufhebung der Limitierung kann der Kraftwerksblock wieder seinen normalen Betrieb aufnehmen.

Es darf nicht unerwähnt bleiben, dass ein Energieversorger solche Marktsituationen manuell handhaben wird, da die Optimierung einen nicht optimalen Fahrplan liefert. Im Falle der dynamischen Programmierung, wäre kein manuelles Eingreifen erforderlich. Es steht außer Zweifel, dass eine manuelle Überprüfung immer noch notwendig ist, da nicht davon ausgegangen werden kann, dass der Algorithmus zu allen Marktsituationen jederzeit fehlerfrei arbeitet und stets einen optimalen Fahrplan garantiert.

6.2 Vergleich der Rechenergebnisse der rückwärts dynamischen Programmierung in Julia und Java

Das Ziel der Arbeit ist ein Performancevergleich zwischen der relativ jungen Programmiersprache Julia und Java auf Basis von rückwärts dynamischer Programmierung. Die im Folgenden gezeigten Ergebnisse dienen als Basis für den Vergleich. Die angeführten Zahlenwerte von der Java-Applikation wurden von der EVN zur Verfügung gestellt.

6.2.1 Vergleich der Berechnungsdauer

Die Ergebnisse der Berechnungsdauer zwischen Java und Julia ist Tabelle 6.2 zu entnehmen.

Tabelle 6.2: Vergleich der Berechnungsdauer von rückwärts dynamischer Programmierung in Julia und Java.

Zeitraum	Java	Julia
1 Woche	0,03	0,18
1 Monat	0,49	0,35
2 Monate	0,99	0,59
3 Monate	1,52	0,86
4 Monate	2,08	1,14
5 Monate	2,64	1,37
6 Monate	3,20	1,67
7 Monate	3,84	1,90
8 Monate	4,48	2,15
9 Monate	4,94	2,33
10 Monate	6,17	2,98
12 Monate	7,40	3,72

Julia löst das Optimierungsproblem schneller als Java. In diesem Sinne hat Julia im direkten Vergleich zu Java auf Basis von rückwärts dynamischer Programmierung einen Performancevorteil. Insbesondere bei steigender Schrittanzahl bewährt sich die Umsetzung in Julia und die Berechnungsdauer halbiert sich. Abbildung 6.2 veranschaulicht die Rechendauer in Abhängigkeit der Schrittanzahl.

An dieser Stelle wird darauf hingewiesen, dass zwar bei beiden Lösungsvarianten die rückwärts dynamische Programmierung angewendet wurde, jedoch der Algorithmus in Julia leicht abgewandelt wurde. Der Algorithmus, welcher in Java implementiert wurde, legt zuerst den gerichteten Graphen an und bestimmt für jeden einzelnen Zustand die möglichen Vorgänger bzw. Nachfolger. Dadurch ergibt sich für jeden Zustand bereits vor Beginn der Berechnung eine Liste von erlaubten Übergängen. Durch diese Vorgehensweise wird bereits vor der eigentlichen Berechnung Zeit benötigt alle Zustände exakt zu erfassen. Der Vorteil dieser Herangehensweise besteht darin, dass sich bei geringfügiger Änderung der Eingangsdaten, sich nur ein kleiner Teil des Graphen ändert.

Der Algorithmus, welcher in Julia implementiert wurde, arbeitet sehr ähnlich. Ein bedeutender Unterschied liegt darin, dass der gerichtete Graph nicht bereits vor der Berechnung des Optimums definiert wird, sondern in erster Linie nur Platzhalter angelegt werden. Die zulässigen Vorgänger bzw. Nachfolger werden erst zur Laufzeit ermittelt. Es ist klar ersichtlich, dass dadurch weitaus weniger Rechenaufwand betrieben wird und die Rechenzeit stark gesenkt wird. Dies erfordert jedoch einen effizienten und zugleich intelligenten Algorithmus, der zur Laufzeit die richtigen bzw. optimalen Entscheidungen trifft. Des Weiteren wurde der Algorithmus um eine weitere Komponente, die sich als Prioritätenliste verstehen lässt, erweitert. Sie legt in abstrakter Weise die Priorität der

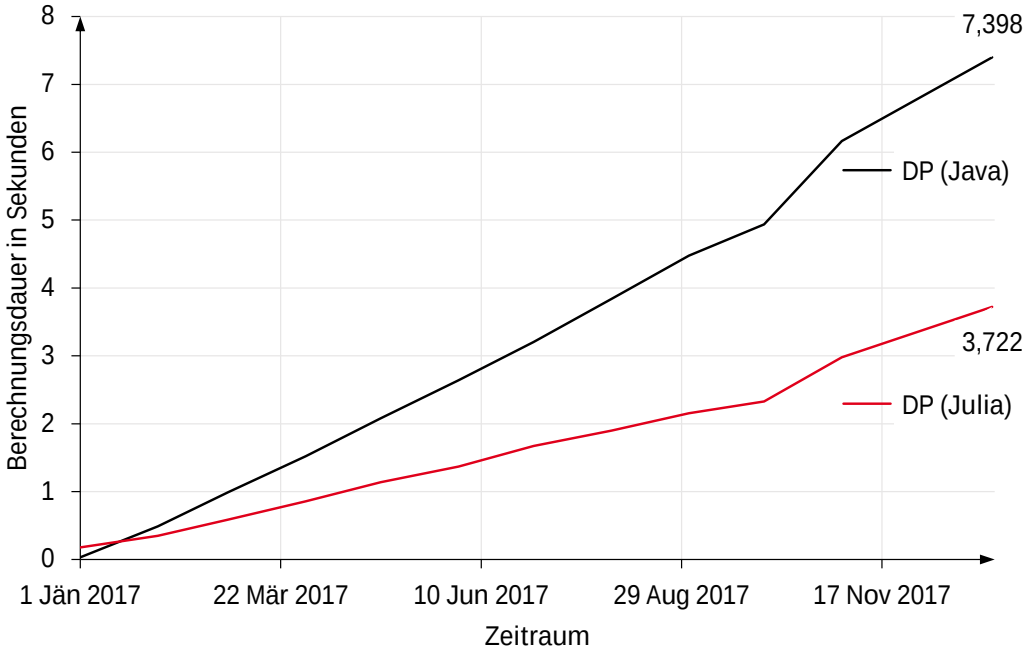

Abbildung 6.2: Vergleich der Berechnungsdauer der rückwärts dynamischen Programmierung in Julia und Java in Abhängigkeit der Schrittanzahl.

erlaubten Vorgänger fest und zwar in dem Sinne, dass sie in der Reihenfolge abgearbeitet werden, die am wahrscheinlichsten zum optimalen Pfad führen. Dadurch wird der optimale Vorgänger quasi geschätzt und als erstes berechnet. Falls dieser immer lokale Optima liefert, so bedeutet dies auch, dass sie in Summe ein globales Optimum nach Bellman[2] bilden. Dadurch werden alle anderen möglichen Vorgänger, die eine niedrigere Priorität haben nicht mehr berechnet, um Zeit einzusparen. Ein weiteres wichtiges Merkmal der dynamischen Programmierung ist die Speicherung von Zwischenergebnissen, welche bereits in den vorangegangenen Abschnitten erläutert wurde.

6.2.2 Vergleich der Erträge

Durch die eingangs erwähnten Unterschiede im Algorithmus, liefern beide Ansätze unterschiedliche Lösungen. Die Fahrpläne sind relativ ident und weichen nur marginal voneinander ab. Ein Grund dafür liegt darin, dass die Lösung in Julia mit ausschließlich gerundeten Werten rechnet, um die Berechnung schneller ausführen zu können. Des Weiteren erlaubt der Algorithmus in Julia auch das tatsächliche Anspringen eines jeden Leistungswertes. Die Lösung in Java hingegen arbeitet mit einem festgelegten Gitter an Leistungswerten, welche angesprungen werden können, was sich als Nachteil erwiesen hat. Unterschiede im generierten Profite können auch daher kommen, dass zu manchen Zeitabschnitten der Kraftwerksblock auch bei niedrigen Marktpreisen in Betrieb gehalten und erst später heruntergefahren wird. Der Sinn dahinter liegt in den

[2]Optimalitätsprinzip nach Bellman

geringer anfallenden Startkosten, da unter Umständen statt eines Kaltstarts ein Warmstart vorgenommen werden kann. Tabelle 6.3 zeigt den Vergleich des generierten Profits beider Lösungsvarianten für den Zeitraum von einem Jahr.

Tabelle 6.3: Vergleich des berechneten Profits und der resultierenden Differenz zum abgeschätzten optimalen Abrufplan.

Vergleich des Profits	Total Profit in EUR	Differenz zu optimalen Abrufplan in %
Java	16 863 020	0,40
Julia	16 604 615	0,49
Differenz	258 405	0,09

Da beide Lösungen reine Näherungsverfahren sind, lässt sich nur durch eine Abschätzung eine Aussage über die Abweichung zum tatsächlichen optimalen Fahrplan treffen. Auf Basis der Grenzkosten und Vernachlässigung des Leistungsänderungsgradienten kann ein optimaler Abrufplan errechnet werden. Zweifellos ist dieser in der Realität aufgrund von technischen Einschränkungen nicht fahrbar. Allerdings wird versucht durch qualitativ möglichst gute Näherungen dem optimalen Abrufplan unter Berücksichtigung der technischen Restriktionen sehr nahe zu kommen. Anhand dieser Berechnung lässt sich eine Aussage über die Qualität der Optimierung treffen, in dem man die Abweichung zu jedem einzelnen Zeitpunkt berechnet und über alle Zeitschritte mittelt. Dadurch kann man erkennen, dass die Lösungen in Java und Julia für den betrachteten Zeitraum von einem Jahr sich sehr ähneln.

6.2.3 Vergleich des Codeaufwands

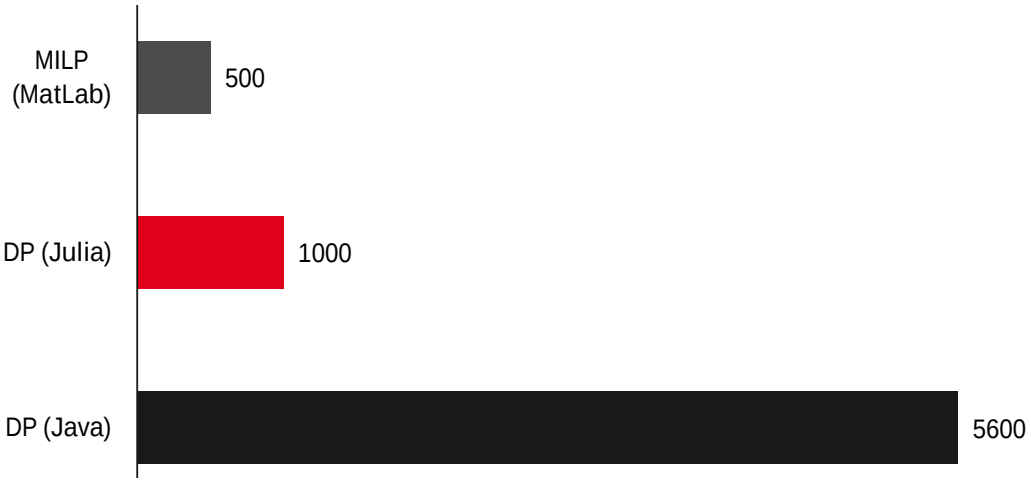

Abbildung 6.3: Vergleich der Lines-of-Code der verschiedenen Lösungsmethoden.

Da wie bereits erwähnt der Algorithmus von beiden Lösungen verschieden ist, ist auch ihr Codeaufwand unterschiedlich. Abbildung 6.3 verdeutlicht den Vergleich der einzelnen

Lösungen und stellt die Anzahl der Lines-of-Code grafisch dar.

Die Implementierung des Modells in MatLab und Lösung mittels *GUROBI* schlägt sich in der Menge der programmierten Zeilen nieder. Durch die einfache Beschreibung des Modells, lässt sich der Codeaufwand reduzieren, im Gegensatz zu rückwärts dynamischer Programmierung. Die rückwärts dynamische Programmierung kann auf verschiedene Weisen implementiert werden und hängt in erster Linie vom Programmierer selbst ab. Die Anzahl an benötigten Codezeilen in der Realisierung in Java wurde von der EVN zur Verfügung gestellt.

6.2.4 Vergleich des Arbeitsspeicherbedarfs

Eine weitere wichtige Kenngröße ist der Speicherbedarf der Optimierung im Arbeitsspeicher. Für die Ermittlung der Werte wurden Makros benutzt, die die Entwicklungsumgebung zur Verfügung stellt. Abbildung 6.4 veranschaulicht den Vergleich des Arbeitsspeicherbedarfs der Optimierung in Julia und Java.

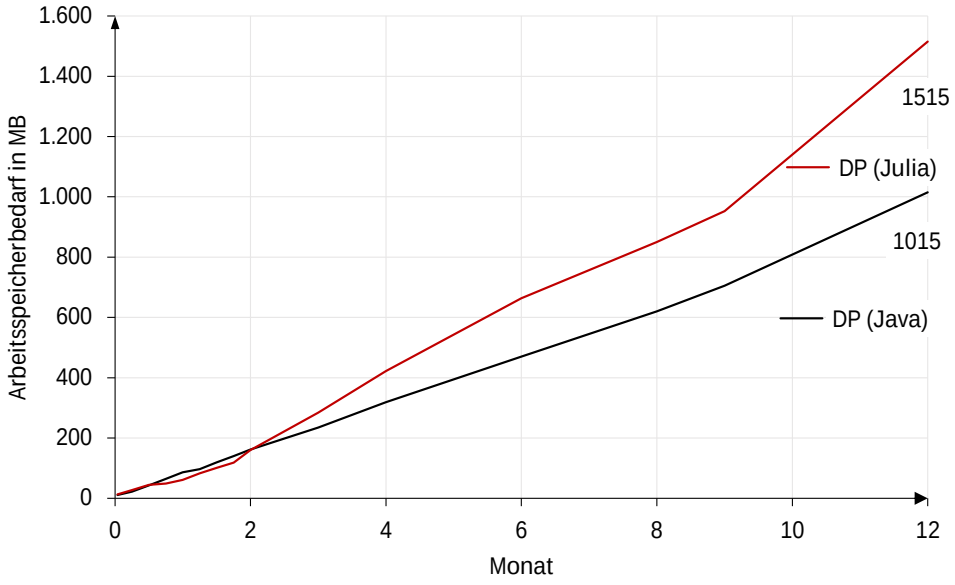

Abbildung 6.4: Vergleich des Arbeitsspeicherbedarfs der rückwärts dynamischen Programmierung in Julia und Java.

Durch die effiziente Programmierung fällt der Platzbedarf im Arbeitsspeicher sehr gering aus und kann als vernachlässigbar behandelt werden.

6.2.5 Kritik am Vergleich

An dieser Stelle sei darauf hingewiesen, dass der Performance-Vergleich von Java und Julia mittels rückwärts dynamischer-Programmierung innerhalb dieses Projekts keine eindeutige Aussage zulässt. Die im vorangengenen Abschnitt erläuterte Thematik, dass

der Algorithmus in Julia die Optimierung unterschiedlich vornimmt, als der Algorithmus in Java, steht im Gegensatz zu einem Vergleich mit komplett identen Randbedingungen. Für einen direkten Vergleich müsste die zurzeit bestehende Java-Applikation in Julia übersetzt oder die entwickelte Julia-Applikation in Java übersetzt werden. Dies hätte einen fundierten Vergleich mit eindeutiger Aussage zu Folge.

7 Schlussfolgerung

In diesem Kapitel wird eine abschließende Aussage der Arbeit getroffen und in Folge auf Thematiken eingegangen, die im weiteren Forschungsverlauf von Interesse sind, auf die jedoch im Rahmen der Diplomarbeit nicht näher eingegangen werden konnte.

Zusammenfassend lässt sich sagen, dass die implementierte Software Lösung einen Prototyp darstellt. Für eine hinreichend qualitative Beantwortung der Forschungsfrage ist dieser vollkommen ausreichend. Es steht außer Zweifel, dass für eine alltägliche Verwendung in der Fahrplanerstellung Weiterentwicklungen und Adaptierungen notwendig sind. Um in erster Linie eine fehlerfreie Verwendungen der Software Lösung mit Nutzern zu gewährleisten, bedarf es Fehlererkennungsmechanismen im Programm. Diese wurden vernachlässigt behandelt, da der Entwickler selbst als einziger Nutzer bzw. Tester die Applikation bedient. Als grundlegendes Fundament für Fehlererkennung sind zusätzliche Codesequenzen wie zum Beispiel if-Abfragen und try-catch-error Routinen notwendig. Da im Rahmen des Projekts alleine die Performance als Kernelement angesehen wird, wurde gänzlich auf fehlerabfangende Maßnahmen verzichtet. Unter realen Marktbedingungen sind alle notwendigen Informationen in Form von Zeitreihen auf Datenbanken gespeichert. Somit wäre für eine alltägliche Benutzung eine Datenbankanbindung von Nöten. Des Weiteren muss davon ausgegangen werden, dass der Entwickler selbst nicht der Nutzer der Applikation sein wird. Dadurch muss das Hinzufügen eines Nutzer Interfaces in Form einer grafischen Oberfläche in Betracht gezogen werden. Solch ein User Interface hat hohen Programmieraufwand zu Folge und konnte somit nicht im Rahmen der Diplomarbeit realisiert werden, was auch kein festgelegtes Ziel der Arbeit war.

Die entwickelte Applikation versucht stets den besten Fahrplan, sprich den optimalen Leistungswert zu einem bestimmten Zeitpunkt zu berechnen. In der Realität kann es jedoch aufgrund von verschiedenen Einflüssen vorkommen, dass der vorgegebene Leistungswert nicht gefahren werden kann. Die einzige Information, die zum Beispiel bei einem Rampenverlauf für die Optimierung als relevant eingestuft wird, ist die zuvor angelaufene Stillstandszeit. Der Algorithmus hat keine ausführliche Auskunft über die aktuellen Verfügbarkeiten des Kraftwerksblocks. In einem optimalen Fahrplan wird es im regulären Fall keine Probleme machen alle Stillstandszeiten einzuhalten und den jeweiligen Starttyp auszuwählen. Diese Information gibt der Lastverteiler den Kraftwerksbetreibern weiter. Dieser hat allerdings keine Zugriffe bzw. Information auf den tatsächlichen Betriebszustand des Kraftwerks. So kann es aufgrund von technischen Schwierigkeiten vorkommen, dass ein Kraftwerksbetreiber den vorgeschriebenen Fahrplan um ein paar Minuten verzögert anwenden kann. Beispielsweise würde der vorgeschriebene Fahrplan einen Heißstart vorschreiben, der Kraftwerksbetreiber jedoch kann diesen Fahrplan nicht

minutengenau befolgen und nimmt einen Warmstart vor. Aufgrund dieser Sachlage ist eine stetige und gute Kommunikation zwischen Lastverteiler und Kraftwerksbetreiber notwendig, da somit eine Aktualisierung der Eingabedaten bei der Optimierung stattfinden kann. Die Applikation, welche die Optimierung vornimmt, arbeitet mit dem aktuellen Betriebszustand des Kraftwerks. In diesem Sinne wäre eine Erweiterung der Applikation um zusätzliche Kraftwerksparameter sinnvoll, um unerwünschte Zustände vermeiden zu können.

Wie die Arbeit gezeigt hat, wäre eine rekursive Lösung mittels rückwärts dynamischer Programmierung stets zu bevorzugen. In der Arbeit wurde nachgewiesen, dass bei der Berechnung der Fahrpläne für größere Zeiträume mit feinerer Zeitgranularität die rückwärts dynamische Programmierung der gemischt-ganzzahligen linearen Programmierung eindeutig überlegen ist. Andererseits jedoch muss in Betracht gezogen werden, dass die Entwicklung eines Modells, welches das Verfahren zur dynamischen Programmierung nutzt, deutlich komplexer ausfällt, als ein Modell, das sich mittels eines Solvers lösen lässt. Der große Vorteil in der Nutzung eines Solvers wie *GUROBI* liegt in der Einfachheit der Modellimplementierung. Die Entscheidung, welches Lösungsverfahren verwendet wird, hängt von der Art der Applikation und des Problems ab. In Falle der vorliegenden Problemstellung ist ein dynamisches Verfahren mit minimaler Berechnungszeit zu bevorzugen, da die Optimierung sehr häufig aufgerufen wird. Zusätzlich ist ihr Ergebnis zeitkritisch, da es für Entscheidungen am realen Markt herangezogen wird und es von Bedeutung ist, dass ein Optimierungsvorgang eine möglichst kurze Berechnungsdauer hat.

Wie die Ergebnisse gezeigt haben, bringt die Linearisierung der quadratisch inversen Produktionsfunktion eine relativ große Zeitersparnis. Der Grund dafür liegt in der Reduzierung des eigentlich quadratischen Optimierungsproblems zu einem linearen. *GUROBI* ist im Stande ein quadratisches Problem zu lösen. Für eine exakte Lösung wäre eine quadratische Formulierung notwendig. Hier wäre zu fragen, ob solch eine mathematisch aufwendigere Formulierung weitaus bessere Ergebnisse im Hinblick auf Genauigkeit liefert, wenn auch der Umstand der Berechnungsdauer eine Rolle spielt. Ein zusätzlicher Nachteil in Form einer Vereinfachung in der Realisierung mit gemischt-ganzzahliger linearer Programmierung ist die Kopplung an die Formate der Parameter. Beispielhaft sei hier die Dauer einer Rampe erwähnt, welche nicht beliebig verändert werden kann, da ansonsten Teile des Programmcodes adaptiert werden müssten. Die Leistungspunkte einer Rampe können sehr wohl ohne Komplikationen verändert werden. Die rückwärts dynamische Programmierung ist entkoppelt von den Formaten der Parameter und ist deshalb vielseitiger einsetzbar. Die Einschränkungen beim Start-, sowie dem Endpunkt haben zwar sehr geringe Auswirkungen auf das Ergebnis, sind jedoch weitere Vereinfachungen, die das Ergebnis bezüglich Genauigkeit negativ beeinflussen.

Eine Frage, die durch diese Arbeit nicht geklärt werden konnte, ist die Einbindung von Speichern. In der Realität kommen in der Einsatzplanung häufig Speicher, welche im Zuge dieser Arbeit nicht eingebunden wurden. Es ist unbestritten, dass die Implementierung von Speichern den Grad der Komplexität noch weiter steigern. Somit ist die Einbindung von Speichern eine lohnenswerte Aufgabe für zukünftige Untersuchungen.

Bei der Entwicklung des Algorithmus wurden spezielle Fälle erfasst, die durch intelligente Erweiterungen eine weitere Steigerung hinsichtlich Performance liefern könnten. So könnte beispielsweise die Prioritätenliste, die den optimalen Vorgänger schätzt noch optimiert werden. Es wäre grundlegend möglich anhand eines Schrittes nicht nur den nächsten Vorgänger, sondern mehrere zu schätzen, um somit komplette Pfadelemente schneller abarbeiten zu können. Bei einer Marktlage von sehr hohen oder sehr niedrigen Marktpreisen wäre die bevorzugte elektrische Abrufleistung bereits bekannt und somit könnten gleich mehrere Zeitschritte übersprungen werden. Als weiteren interessanten Aspekt sei die Einführung von neuronalen Netzen erwähnt. Für bestimmte Programmabschnitte würde sich ein Ersatz durch neuronale Netzen grundlegend anbieten. Um diese Frage eindeutig beantworten zu können, bedarf es weiterer Untersuchungen.

Da Julia noch relativ jung in der Welt der Programmiersprachen ist, ist sie im Vergleich zu anderen Programmiersprachen (noch) relativ fehlerbehaftet. Die Entwicklungsarbeit die an Julia geleistet wird ist groß und täglich kommen neue Packages und Komponenten hinzu, gestärkt durch den Open-Source Gedanken. Zweifellos bedarf es noch einiger Entwicklungsarbeit und -zeit, um mit anderen bereits etablierten Programmiersprachen auf Augenhöhe konkurrieren zu können. Die Arbeit hat gezeigt, dass die Modellierung ohne weiteres in Julia implementiert werden kann. Die Lösung des Modells gestaltet sich genau so komplex, wie in anderen Programmiersprachen. Allerdings zeigt Julia großes Potential im Hinblick auf Performance und Bedienbarkeit. Deshalb wird das Projekt von der EVN mit großer Wahrscheinlichkeit weiter verfolgt versucht mehr Problemstellungen mit Hilfe von Julia zu lösen.

Abbildungsverzeichnis

Literatur

[1] S. B. Aruoba,J. Fernandez-Villaverde, 5 August 2014. Available: https://www.sas.upenn.edu/ jesusfv/comparison_languages.pdf [Zugriff am 22. November 2018]

[2] JuliaLang, The Julia Programming Language. Available: https://julialang.org/ [Zugriff am 22. November 2018]

[3] Gawlik W., Energieversorgung, 2015

[4] Different approaches on power-based Unit Commitment formulation, Putz D., Gumhalter M., Technische Universität Wien - Energy Economics Group, [2018]

[5] Weisstein E., Taylor Series. Available: http://mathworld.wolfram.com/TaylorSeries.html/ [Zugriff am 13. Dezember 2018]

[6] GUROBI, GUROBI Optimization. Available: http://www.gurobi.com/resources/gettingstarted/mip-basics [Zugriff am 03. Oktober 2018]

[7] Aumüller Martin, Effiziente Algorithmen, TU Ilmenau. Available: https://www.tu-ilmenau.de/fileadmin/public/iti/Lehre/Eff.Alg/SS12/EA-SS12-Kapitel5.pdf/ Zugriff am 03. Oktober 2018]

[8] Auer H., Energiemodelle und Analysen - Kapitel 03 - Lineare Optimierung, 2017

Eidesstattliche Erklärung[1]

Hiermit erkäre ich, dass die vorliegende Arbeit gemäß dem Code of Conduct, insbesondere ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel, angefertigt wurde. Die aus anderen Quellen oder indirektübernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Wien, _____ _____

Datum Unterschrift

Statutory Declaration

I declare that I have authored this thesis according the Code of Conduct independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Wien, _____ _____

Date Signature

[1] Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008