Untersuchungen zur kooperativen Spieltheorie und Erweiterung des R-Pakets *CoopGame*

Masterarbeit

im Studiengang
Informatik

vorgelegt von

Johannes Anwander
Matr.-Nr.: 247620

am 17. Juli 2017
an der Hochschule Kempten

Aufgabensteller/Prüfer Prof. Dr. Jochen Staudacher
Anschrift des Verfassers Maler-Lochbihlerstr. 16, 87435 Kempten

Kurzfassung

Diese Masterarbeit beschreibt die Schritte und Überlegungen, die zur Weiterentwicklung des von Prof. Dr. Staudacher an der Hochschule Kempten initiierten R-Pakets *CoopGame* dahingehend durchgeführt wurden, um einen geeigneten Rahmen für eine zeitnahe Veröffentlichung auf *CRAN* zu schaffen.

Besonderes Augenmerk lag hierbei darauf, alle bisherigen Implementierungen im Hinblick ihrer korrekten Umsetzung erneut zu hinterfragen, fehlende Konzepte der kooperativen Spieltheorie zu untersuchen und gegebenenfalls das Paket um diese zu ergänzen. Als Ergebnis dieser Untersuchungen wurden nötige Modifikationen und Erneuerungen für ein in sich stimmiges Paket zur kooperativen Spieltheorie vorangetrieben, so dass neben den durch CRAN gestellten Qualitätsansprüchen auch dem Nutzen in der Lehre ausreichend Sorge getragen wurde. Insbesondere wurde das Paket – neben weiteren Spieleeigenschaften – um eine Vielzahl neuer punktwertiger Lösungsansätze, wie z.B. dem Public Help Index und verschiedenen Nucleolus Derivaten, ergänzt.

Die bereits im Paket enthaltenen Konzepte des Nucleolus und des Prenucleolus, die aus der Bachelorarbeit von GEBELE (2016) hervorgingen, wurden auf der Basis eines objektorientierten Ansatzes in R mit S4 Klassen überarbeitet und abgelöst. So wurde unter Berücksichtigung der Ergebnisse aus GEBELE (2016**)** ein Rahmenwerk für eine flexiblere und vereinfachte Implementierung von Nucleolus Derivaten geschaffen. Unter Nutzung eben dieses Frameworks wurden Derivate wie der Per Capita Nucleolus, der Proportional Nucleolus, der Disruption Nucleolus, der Modiclus und einer dessen Varianten, der Simplified Modiclus, realisiert.

Für das Lösungskonzept des Gately Punkts (GATELY 1974) wurde jedoch in dieser Arbeit die implizierte Eindeutigkeit widerlegt; mit der Konsequenz, dass ein neues eindeutiges Lösungskonzept als eigene Neuauslegung des Gately Punkts in Form des Lexical Gately Punkts geschaffen wurde. Dieses neue Konzept weist starke Ähnlichkeiten zum Disruption Nucleolus auf und minimiert lexikographisch die maximalen Störanfälligkeiten durch Einerkoalitionen (STAUDACHER 2017). Im nicht eindeutigen Fall macht sie dagegen willkürlich einen Punkt der Imputationsmenge als Lösung aus. Dieses Vorgehen soll aber zunächst nur eine erste Diskussionsgrundlage schaffen.

Neben in `CoopGame` erfolgten auch Beiträge und Überarbeitungen in den Paketen `PartitionGames` und `CommunicationGames`, die aus `CoopGame` heraus entstanden sind und jeweils eine Abgrenzung zwischen der reinen kooperativen Spieltheorie und partitiven sowie netzwerkbasierten kooperativen Ansätzen vollziehen. Zudem wurde mit dem Paket AuxiliaryTools ein Paket geschaffen, das nicht für die Veröffentlichung auf CRAN vorgesehen ist, aber Funktionalitäten für eine zentrale Pflege der drei übrigen Pakete bereitstellt.

Abstract

This master thesis describes the steps and considerations that have been pursued to further develop the existing R package `CoopGame` in order to provide a suitable framework for a near-term publication on CRAN. The package itself was earlier initiated by Prof. Dr. Jochen Staudacher at Hochschule Kempten.

Particular attention was given to all previous implementations with regard to their correct implementation. Moreover, missing concepts of the cooperative game theory were examined and, if reasonable, included in the package. As a result of all these investigations necessary modifications and renovations were carried out for a coherent package in the cooperative game theory, so that sufficient care was taken in respect of the CRAN quality requirements and the use in teaching.

In particular, the package was supplemented by a number of new point valued solution concepts, such as the Public Help Index and various Nucleolus variants.

The concepts of the Nucleolus and the Prenucleolus within `CoopGame`, established by GEBELE (2016), were revised and replaced based on an object-oriented approach in R with S4 classes. A framework for a more flexible and simplified implementation of nucleolus variants has been therefore created, considering the results from GEBELE (2016). By using this framework, also derivatives such as the Per Capita Nucleolus, the Proportional Nucleolus, the Disruption Nucleolus, the Modiclus and one of its adaptions the Simplified Modiclus were realized.

However, the implied uniqueness of the Gately Point (GATELY 1974) was refuted in this work. As consequence, a new unique solution concept – a kind of nucleolus derivative with similarities to the Disruption Nucleolus – was created by redefining the Gately point as the Lexical Gately Point. This new concept minimizes the maximal propensities to disrupt by all single coalitions but figures out any valid imputation at will in case of non-uniqueness (STAUDACHER 2017). This should serve as a basis for further discussion.

Aside from `CoopGame`, contributions and revisions were also made in the packages `PartitionGames` and `CommunicationGames`, which originated from `CoopGame`, and where each delineates a distinction between pure cooperative game theory and partitive as well as network-based cooperative approaches.

In addition, the package AuxiliaryTools was created as an internal package which is not intended to be released on CRAN, but provides functionalities for centralized maintenance of the three other packages.

Keywords: Cooperative game theory, CRAN, CoopGame, CommunicationGames, PartitionGames, Gately Point, Lexical Gately Point, Nucleolus variants, point-valued solution concepts

Inhaltsverzeichnis

Abbildungsverzeichnis

Codebeispielverzeichnis

Formelverzeichnis

Tabellenverzeichnis

Danksagung

An dieser Stelle möchte ich allen meinen Dank aussprechen, die am Zustandekommen dieser Arbeit Anteil hatten.

Dieser Dank gilt im Besonderen Herrn Prof. Dr. Jochen Staudacher für dessen Anregungen zu der vorliegenden Arbeit sowie für die Begleitung und Unterstützung während des Projekts.

1 Einleitung

„Die Spieltheorie stellt das formale Instrumentarium zur Analyse von Konflikten und Kooperation bereit“

Holler & Illing 2006: Vorwort zur ersten Auflage (1990)

Wirft diese Definition auf den ersten Blick doch viele Fragen auf, schafft sie es dennoch kurz und knapp die wesentlichen Charakterzüge der Spieltheorie zu vermitteln.

Hiernach bildet sie das mathematische Rahmenwerk für die Entscheidungstheorie und zwar gleich ob für soziale, ökonomische oder politischer Problemstellungen.

Dabei ist sie als solche noch ein recht junges Gebiet der Mathematik und beruht auf John von Neumann, welcher hierfür mit dem Beweis des Minimax-Theorem 1928 den ersten Grundstein schuf und zusammen mit Morgenstern später durch die Veröffentlichung des Klassikers *The Theory of Games and Economic Behavior* (1944), dieser zu großer Bekanntheit verhalf. In ihr sah er die Chance, soziale Erscheinungen anhand von mathematischen Modellen besonders im Hinblick darauf zu untersuchen, wie Entscheidungen getroffen werden sollten und – zu einem bestimmten Ausmaß – wie sie auch tatsächlich getroffen werden (DAVIS 2005:15).

Bei der Spieltheorie lässt sich dabei grundsätzlich zwischen der kooperativen und der nicht-kooperativen Spieltheorie unterscheiden.

Liegt bei der nicht-kooperativen Spieltheorie zum Großteil noch der Fokus auf den einzelnen Akteuren, die bestimmte Ziele verfolgen und dementsprechend agieren, zeichnet sich die kooperative Spieltheorie dagegen in erster Linie „durch die Möglichkeit verbindlicher Abmachungen, d.h. Kommunikation und exogene Durchsetzung [der Spieler] aus“ (HOLLER & ILLING 2000: 187).

Ist die anfängliche Euphorie über die Spieltheorie in den Geistes- und Wirtschaftswissenschaften zwar deutlich zurückgegangen, da sich immer stärker die Erkenntnis durchgesetzt hat, *„dass die Rationalitätsannahme den Menschen überfordert"* (SELTEN 2008), feiert sie in der Informatik umso mehr ihren Siegeszug.

Gerade in der künstlichen Intelligenz und bei der Modellierung von komplexen Multiagentensystemen in der Informatik erfreut sie sich großer Beliebtheit, da die Annahme der Rationalität hier in der Regel vollständig zutrifft (NEBEL 2009).

Aus diesem Grund ist die Spieltheorie auch in der Lehre nicht mehr wegzudenken. Mit dem Masterprojekt im Sommersemester 2015 an der Hochschule Kempten unter der Leitung von Prof. Dr. Staudacher wurde das Ziel verfolgt, ein R-Paket namens `CoopGame` im Bereich der Kooperativen Spieltheorie der breiten Öffentlichkeit zur Verfügung zu stellen. Gab es mit `GameTheory` (CANO-BERLANGA et al. 2015) und `TUGlab` (MIGUEL & RODRIGUEZ 2006**)** zwar

bereits schon zwei vergleichbare schlanke Pakete, sollte mit `CoopGame` die Anzahl der Konzepte möglichst breit gefächert sein und auch Restriktionen wie die Limitierung auf ausschließlich drei oder vier Spieler wegfallen.

Zudem wurde der Schwerpunkt beim Paket *`CoopGame`* auf punktwertige Lösungskonzepte wie den Nucleolus-Derivaten gelegt. Aufgrund der sich sehr stark ähnelnden Konzepte sollte insbesondere hier neben dem sonst verfolgten Rapid Prototyping – um fehleranfälligen redundanten Code zu vermeiden – ein objektorientierter Ansatz evaluiert werden, welcher auch aus softwaretechnischer Sicht zu überzeugen weiß.

Die vorliegende Masterarbeit setzt sich mit der Weiterentwicklung des bestehenden R-Pakets `CoopGame` auseinander und versucht einen geeigneten Rahmen für eine zeitnahe Veröffentlichung auf *CRAN* zu schaffen.

2 Grundlagen und Definitionen

Um sich eingehend mit den im Rahmen dieser Masterarbeit vorgestellten Konzepte befassen zu können, erfordert dies im Vorfeld die Klärung und Definition hierfür benötigter Begriffe.

Hierzu werden zuerst einige, für alle vorgestellten Konzepte wichtige, theoretischen Grundlagen der kooperativen Spieltheorie erörtert, gefolgt von Ausführungen zur Linearen Programmierung und der lexikographischen Ordnung, die später eine besondere Rolle bei der Vorstellung des Nucleolus und dessen Varianten (siehe Abschnitt 4.2) spielen werden.

2.1 Grundlagen der kooperativen Spieltheorie

In diesem Abschnitt wird auf die Grundlagen der kooperativen Spieltheorie näher eingegangen. Dabei soll kein Gesamteindruck über die kooperative Spieltheorie, sondern nur über die im Kontext dieser Masterarbeit wesentlichen Konzepte vermittelt werden.

2.1.1 n-Personen-Spiel

Handelt es sich um ein kooperatives n-Personenspiel (N, v) (siehe Formel 1), repräsentiert N die Menge aller am Spiel beteiligten n Spieler (Holler & Illing 2006: 267). Dabei muss n einer natürlichen Zahl entsprechen. Am Spiel sind hier mindestens zwei, jedoch höchstens endlich viele Spieler beteiligt. Die Mächtigkeit der Menge N liegt beim Wert n.

Bei einem n-Personen-Spiel obliegt es den Spielern eine Strategie auszuwählen und gemeinsam umzusetzen. Ein Zusammenschluss, der sich aus Spielern zum Zweck der Verfolgung einer gemeinsamen Strategie bildet, wird hierbei als Koalition bezeichnet und im folgenden Abschnitt vorgestellt, die Koalitionsfunktion v in Abschnitt 2.1.4.

Formel 1: Definition eines n-Personenspiel

$N = \{1,\ 2, \ldots, n\}\ Menge\ aller\ n\ Spieler$:

1. $n \in N$
2. $n \geq 2$
3. $n = |N| \neq \infty$

2.1.2 Koalitionen

Holler & Illing (2006: 267) beschreiben eine Koalition im Allgemeinen als jede nicht-leere Teilmenge der Spielermenge N. Hierbei lassen sich bestimmte Koalitionen näher charakterisieren.

Eine Einerkoalition stellt eine Koalition mit der Kardinalität von 1 dar, sie enthält genau einen der Spieler aus der Spielermenge N. Unter der großen Koalition hingegen wird eine Koalition verstanden, der alle Spieler der Spielermenge angehören. Der Nullkoalition wiederum gehört kein Spieler an.

2.1.3 Transferierbarer Nutzen

Nach Holler & Illing (2006: 270) ist es in der kooperativen Spieltheorie für ein Spiel elementar, ob der Nutzen transferierbar ist oder nicht.

Unter transferierbarem Nutzen wird hierbei verstanden, dass ein Spieler auf einen Teil seines erhaltenen Nutzens auch zugunsten seiner Mitspieler verzichten kann und ihnen diesen bei der Aufteilung zugesteht (Holler & Illing 2006: 270f).

Im Weiteren wird im Rahmen dieser Masterarbeit transferierbarer Nutzen für sämtliche behandelten Spiele implizit vorausgesetzt.

2.1.4 Koalitionsfunktion und. Spielevektor

Bei einer Koalitionsfunktion handelt es sich neben dem Spielevektor nach Holler & Illing (2006: 270) um eine mögliche Repräsentationsform eines Spiels (N, v). Eine Koalitionsfunktion v, die auch charakteristische Funktion genannt wird, bestimmt dabei für jede mögliche Koalition aus der Spielermenge N eine reelle Zahl, den sogenannten Koalitionswert. Dieser ermittelte Koalitionswert entspricht dabei dem Nutzen, der sich für die Mitglieder der Koalition ergibt. Für die Koalitionsfunktion muss hierbei gelten, dass die Funktion angewendet auf die Nullkoalition den Wert 0 liefert

Der in `CoopGame` verwendete Spielevektor A ist ein Tupel der Größe $2^n - 1$, das für alle Koalitionen (bis auf die Nullkoalition), welche sich aus der Potenzmenge ergeben, den jeweiligen Koalitionswert in einer fest vorgegebenen Reihenfolge enthält.

Die Koalitionswerte werden hinsichtlich der Anzahl, der an ihnen beteiligten Spieler und deren Spielerindizes aufsteigend angeordnet.

Wird beispielsweise der Dreispielerfall herangezogen, liefert die Potenzmenge für $N = \{1,2,3\}$ acht verschiedene Teilmengen. Für die Anordnung werden alle Teilmengen bis auf die leere Menge betrachtet. Unter Berücksichtigung der Anzahl der beteiligten Spieler sowie deren Spielerindizes und unter Ausschluss der Nullkoalition, ergibt sich aufsteigend folgende Anordnung wie in Formel 2.

Formel 2: Beispiel Anordnung der Koalitionswerte im Spielevektor für den Dreispielerfall

$$A = (v(1), v(2), v(3), v(1,2), v(1,3), v(2,3), v(1,2,3))$$

Im Folgenden wird der charakteristische Vektor im Zusammenhang mit dem Hilfskonzept der Bitmatrix behandelt.

2.1.5 Charakteristische Vektor und die Bitmatrix

Für eine fest definierte Reihenfolge der Koalitionen entsprechend des Spielevektors A – auch hier ohne die Nullkoalition – beschreibt der charakteristische Vektor für jeden Spieler durch die Zuordnung des Werts 1, ob dieser in der jeweiligen Koalition mitbeteiligt ist; 0 spiegelt die Nichtbeteiligung wider.

Im Dreipersonenfall ergeben sich die in Formel 3 aufgeführten charakteristischen Vektoren CV_i für die Spieler $i \in N = \{1,2,3\}$.

Formel 3: Charakteristische Vektoren im Dreispielerfall

$$CV_1 = (1,0,0,1,1,0,1)$$
$$CV_2 = (0,1,0,1,0,1,0)$$
$$CV_3 = (0,0,1,0,1,1,0)$$

Werden für alle Spieler deren charakteristischen Vektoren als Spalten angeordnet, ergibt sich die Bitmatrix. Eine zusätzliche Spalte gibt darüber hinaus über den erzielten Koalitionsgewinn Aufschluss.

Index	**Bit-Zeile für S**	**Spieler $\in S$**	$v(S)$
1	1 0 0	{1}	$v(\{1\})$
2	0 1 0	{2}	$v(\{2\})$
3	0 0 1	{3}	$v(\{3\})$
4	1 1 0	{1,2}	$v(\{1,2\})$
5	1 0 1	{1,3}	$v(\{1,3\})$
6	0 1 1	{2,3}	$v(\{2,3\})$
7	1 1 1	{1,2,3}	$v(\{1,2,3\})$

Die Bitmatrix ist ein für `CoopGame` hilfreicher Ansatz, welcher dazu beiträgt, die Implementierungen von verschieden Konzepten oftmals leichter und effizienter zu gestalten.

Innerhalb des Pakets `CoopGame` übernimmt die Funktion `createBitMatrix` die Erzeugung einer entsprechenden Bitmatrix mit der Spalte namens `cVal` für die Koalitionsgewinne (siehe Codebsp. 1).

```
# Definition of game vector A and creation of corresponding bit matrix      #1
> A<-c(1,2,3,4,5,6,7)                                                      #2
> createBitMatrix(n=3,A)                                                   #3
     cVal                                                                  #4
[1,] 1 0 0   1                                                            #5
[2,] 0 1 0   2                                                            #6
[3,] 0 0 1   3                                                            #7
[4,] 1 1 0   4                                                            #8
[5,] 1 0 1   5                                                            #9
[6,] 0 1 1   6                                                            #10
[7,] 1 1 1   7                                                            #12
```

Codebsp. 1: Erzeugung Bit Matrix mit Funktionsaufruf von createBitMatrix

2.1.6 Auszahlungsvektor und Auszahlungsraum

Ein Auszahlungsvektor x ist ein Tupel, das jedem Spieler anhand einer Auszahlungsfunktion den Nutzen zuweist, den dieser schließlich zugestanden bekommt (HOLLER & ILLING 2006: 4).

„Die Menge aller möglichen Auszahlungsvektoren, die einem Spiel Γ entsprechen, wird durch den Nutzen- bzw. Auszahlungsraum […] charakterisiert“ (HOLLER & ILLING 2006: 42).

Aus der Menge aller zulässigen Nutzenkombinationen erschließt sich demnach der Auszahlungsraum X. Der Auszahlungsraum kann dabei unterschiedlich spezifiziert werden. Der in Abschnitt 0 beschriebene Kern und die im Zusammenhang mit Imputation und Präimputation (siehe Abschnitt 2.1.7) behandelte Imputations- sowie Präimputationsmenge stellen sehr bedeutsame Auszahlungsräume dar.

2.1.7 Imputation bzw. Präimputation

Eine Imputation (siehe Formel 1) wird nach HOLLER & ILLING (2006: 275) auch Zurechnung genannt und ist eine Auszahlung, die paretooptimal (siehe 1. Bedingung) und individuell rational (siehe 2. Bedingung) ist. Bei Paretooptimalität kann der Auszahlungsvektor nicht durch alle Spieler gemeinsam und bei individueller Rationalität nicht durch einen einzelnen Spieler verworfen werden.

Formel 4: Definition Imputation

$$\text{Eine Auszahlung } x = (x_1, \dots, x_n) \text{ heißt Imputation des Spiels } (N, v),$$
$$\text{wenn für alle Spieler } i \in N \text{ gilt:}$$

1. $\sum_{i \in N} x_i = v(N)$
2. $x_i \geq v(\{i\})$

Paretooptimalität bzw. kollektive Rationalität bedeutet, dass die Summe alle Komponenten eines Vektors x gleich dem Wert sein muss, welchen die Koalitionsfunktion angewandt auf die große Koalition liefert.

Nach der individuellen Rationalität muss hingegen der Auszahlungswert x_i eines jeden Spielers i mindestens so groß sein wie der Wert, den er auf sich allein gestellt in einer Einerkoalition erhält. Sind diese beiden Bedingungen erfüllt, ist der Auszahlungsvektor x eine Imputation.

Eine Präimputation unterscheidet sich von einer Imputation darin, dass die Forderung nach individueller Rationalität erfüllt sein kann, aber nicht zwingend muss. Daraus ergibt sich die in Formel 5 aufgeführte Definition für eine Präimputation.

Formel 5: Definition Präimputation

$$Eine\ Auszahlung\ x = (x_1, \dots, x_n)\ heißt\ Prämputation\ des\ Spiels\ (N, v),$$
$$wenn:$$
$$\sum_{i \in N} x_i = v(N)$$

Auf Grundlage der Definition der Imputation (siehe Formel 4) und der Präimpution (siehe Formel 5) lassen sich die beiden Auszahlungsräume der Imputations- und Präimputationsmenge definieren.

Die Imputationsmenge (siehe Formel 6) beschreibt hierbei den Auszahlungsraum, in der alle Imputationen liegen (PETERS 2015: 159).

Formel 6: Definition der Imputationsmenge

$$I(N, v) = \{x \in \mathbb{R}^n : x(N) = v(N), x_i \geq v(i)\ \forall i\ \in N\}$$

Dementsprechend charakterisiert die Präimputationsmenge (siehe Formel 7) den Auszahlungsraum, in welchem es sich bei allen Auszahlungen um Präimputationen handelt (PETERS 2015: 347).

Formel 7: Definition der Präimputationsmenge

$$I^*(N, v) = \{x \in \mathbb{R}^n : x(N) = v(N)\}$$

Beide Auszahlungsräume erfahren später im Kontext des Nucleolus und des Prenucleolus besondere Bedeutung (siehe Abschnitt 4.2.1).

2.1.8 Der Kern

Beim Kern handelt es sich nach HOLLER & ILLING (2006: 280f) um die Menge aller nicht-dominierten Imputationen, die individuell-, koalitions- und gruppenrational sind.

Der Kern greift dabei als mengenwertiger Lösungsansatz das Konzept der Imputationsmenge auf und bildet durch die zusätzliche Forderung nach Koalitionsrationalität eine Teilmenge daraus ab. Die individuelle und gruppenspezifische Rationalität ist beim Kern bereits durch die Imputationsmenge gedeckt.

Bei der Koalitionsrationalität (siehe Formel 8) handelt es sich indes um eine Ausweitung der individuellen Rationalität auf alle Koalitionen. In jeder Koalition S aus N kann keine höhere Auszahlung erreicht werden, als in der großen Koalition (FROMEN 2004: 168f**)**.

Formel 8: Definition der Koalitionsrationalität

$$\sum_{i \in S} x_i \geq v(S), \text{für alle Koalition } S \subseteq N$$

Der Kern eines Spiels (N, v) lässt sich im Weiteren unter Berufung auf die Imputationsmenge und die Koalitionsrationalität wie in Formel 9 nach HOLLER & ILLING (2006: 280) definieren.

Formel 9: Definition des Kerns C(N,v)

$$C(N, v) = \{x \in I(N, v) | v(S) - \sum_{i \in S} x_i \leq 0, \text{für alle } S \subseteq N\}$$

Die Elemente des Kerns sind dabei intern stabil, da keine Imputation innerhalb des Kerns dominiert wird. Externe Stabilität ist dagegen nicht erfüllt, weil es Auszahlungsvektoren außerhalb des Kerns gibt, die von keinem Element des Kerns dominiert werden (HOLLER & ILLING 2006: 280f).

Die Eigenschaft der Balanciertheit und das Bondareva-Shapley Theorem, das Aussagen über die Existenz des Kerns zulässt, werden später in Abschnitt 3.3 erläutert.

2.1.9 Einmütigkeitsspiele

Ein Einmütigkeitsspiel ist ein einfaches Spiel (N, v), das durch eine nichtleere Menge T von N charakterisiert ist. Bei den Spielern aus T handelt es sich hierbei ausschließlich um Vetospieler. Diese sind für eine Gewinnkoalition unentbehrlich.

Formel 10: Definition eines Einmütigkeitsspiels

Ein Einmütigkeitsspiel ist ein Spiel mit einer nichtleeren Teilmenge T von N, für das gilt:

$$u_T(K) = \begin{cases} 1, & K \supseteq T \\ 0, & sonst \end{cases}$$

Die Einmütigkeitsspiele sind später noch bei der Vorstellung der Einmütigkeitskoeffizienten in Abschnitt 3.4 von essentieller Bedeutung.

2.1.10 Wesentliche Spiele

Wesentliche Spiele sind dadurch charakterisiert, dass sich bei Bildung der großen Koalition ein Mehrwert gegenüber den Einerkoalitionen ergibt.

In einem wesentlichen Spiel (siehe Formel 11) ist die Summe der Einerkoalitionswerte demnach kleiner als der Wert der großen Koalition.

Formel 11: Definition eines wesentlichen Spiels

$$Sei\ (N, v)\ ein\ wesentliches\ Spiel, so\ gilt:$$

$$\sum_{i \in N} v(i) < v(N)$$

Den wesentlichen Spielen wird später im Zusammenhang der Definition eines eindeutigen Nucleolus (siehe Abschnitt 4.2) große Bedeutung zu Teil, weisen diese doch die überaus positive Äquivalenzbeziehung auf, dass für sie die Imputations- sowie Präimputationsmenge nie leer ist (PETERS 2015: 159).

2.1.11 Gewichtetes Abstimmungsspiel

Nach Wiese (2005: 94) ist ein gewichtes Abstimmungsspiel ein Spiel, bei dem es zu Koalitionsbildung kommt, um gemeinschaftlich eine erforderliche Quote zu erreichen. Jeder Spieler i einer Koalition K besitzt hierbei ein persönliches Stimmgewicht w_i, das er in die Koalition miteinbringt.

Ein gewichtetes Abstimmungsspiel ist hierbei durch das Tupel mit den Stimmgewichten und der zu erreichenden Quote charakterisierbar (siehe Formel 12).

Formel 12: Definition eines gewichteten Abstimmungsspiels

$$Für\ ein\ gewichtetes\ Abstimmungsspiel,$$
$$mit\ dem\ Tupel\ [q; w_1, \dots w_n]\ und\ den\ Stimmgewichten\ w_i\ der$$
$$Spieler\ i \in N\ sowie\ der\ zu\ erreichenden\ Quote\ q, gilt:$$

$$v(K) = \begin{cases} 1, & \sum_{i \in K} w_i \geq q \\ 0, & \sum_{i \in K} w_i < q \end{cases}$$

Als Beispiel für ein Abstimmungsspiel kann ein Geschworenengericht in den USA bzw. England angeführt werden. In beiden Ländern werden zwölf Geschworene von Staatsanwaltschaft und Strafverteidigung berufen, die über die Schuldfrage des Angeklagten in einem Strafprozess entscheiden.

Muss in den USA über Schuldfrage zwingend einstimmig entschieden werden, reicht es in England dagegen aus, wenn sich acht der zwölf Geschworene einig sind. Für die USA ergibt sich somit eine zu erfüllende Quote von zwölf ($q^{USA} = 12$), in England von acht Geschworenen ($q^{E} = 8$); jeder Geschwore besitzt ein einfaches Stimmgewicht ($w_i = 1$).

2.2 Lineare Programmierung

Bei Linearer Programmierung handelt es sich um eine relative einfache Optimierungstechnik. Sie wird beispielsweise in Operational Research, beim Internet Routing und in vielen weiteren Anwendungsgebieten genutzt, um unter Berücksichtigung unterschiedlicher Restriktionen auf möglichst effiziente Weise einen optimalen Lösungswert zu bestimmen.

Mathematisch ist ein Lineares Programm (siehe EICHHORN ET. AL. 2016: 179) mit m Nebenbedingungen, n Variablen und einer zu maximierenden (oder zu minimierenden) Zielfunktion $F: \mathbb{R}^s \rightarrow \mathbb{R}$ und den reellen Konstanten c_k, b_j und a_{jk}in allgemeiner Form wie folgt definiert:

Formel 13: Definition eines Linearen Programms in allgemeiner Form

$$Maximiere\ (oder\ minimiere):$$

$$z = F(x_1, \dots, x_n) = c_1 x_1 + \cdots + c_s x_s$$

$$a_{j1} x_1 + \cdots + a_{js} x_s \leq b_j\ (j = 1, \dots, m_1)$$

$$Unter\ Einhaltung\ der\ Nebenbedingungen:$$

$$a_{j1} x_1 + \cdots + a_{js} x_s \leq b_j\ (j = 1, \dots, m_1)$$

$$a_{j1} x_1 + \cdots + a_{js} x_s \geq b_j\ (j = m_1 + 1, \dots, m_2)$$

$$a_{j1} x_1 + \cdots + a_{js} x_s = b_j\ (j = m_2 + 1, \dots, m_3)$$

Für die Lösung eines so definierten Linearen Programms haben sich verschiedene Algorithmen etabliert. An dieser Stelle wird jedoch nur auf jeweils zwei bekannte Verfahrensklassen kurz eingegangenen: das Simplex und das Innere Punkte Verfahren.

Die Grundidee der Simplex Methode wurde von DANTZIG (1947) vorgestellt und in zahlreichen Varianten fortlaufend verbessert. Inzwischen zählt sie zu einem der wichtigsten Algorithmen in der Linearen Optimierung. Zwar benötigt das Verfahren in Extremfällen exponentielle Laufzeit, ist im praktischen Einsatz meist aber in polynomieller Laufzeit lösbar und äußerst effizient (PATIL Et al.2016: 388; DANTZIG 1987).

Daneben gibt es mit der Klasse der Innere Punkte Verfahren weitere Alternativen zur Lösung linearer Programme, welche anders wie der Simplex Algorithmus garantiert polynomiell bezüglich der Laufzeit sind. Dabei garantieren Innere Punkte-Verfahren nicht nur immer theoretisch polynomielle Laufzeit, sondern rechnen auch tatsächlich bei vielen Problemen schneller als Simplex-Typ-Verfahren (BORGWARDT 2001: 141).

Für die Repräsentation eines Linearen Programms gibt es viele unterschiedliche Darstellungsformen, wie die Standard- und die kanonische Form (BORGWARDT 2001: 32**)**. Diese haben aber im Wesentlichen viel gemein, basieren sie doch alle auf linearen Ungleichungssystemen und sind transformationsidentisch – sprich ineinander überführbar (BORGWARDT 2010: 30).

Die Berechnung Linearer Programme erfolgt für `CoopGame` über das R-Packet `glpkAPI`, welches seinerseits eine Schnittstelle auf die ANSI C Bibliothek `GLPK` bereitstellt.

2.3 Lexikographische Ordnung

Bei den später behandelten Nucleolus Derivaten (siehe Abschnitt 4.2) ist die lexikographische Ordnung ein wesentlicher und unabdingbarer Bestandteil für deren Verständnis, daher wird sie in diesem Abschnitt nach PETERS (2015: 345) definiert und einige ihrer Eigenschaften kurz aufgezeigt.

Hierbei wird vorab auf die Begriffe partielle, schwache und lineare Ordnung eingegangen, gefolgt von den eigentlichen Ausführungen zur lexikographischen Ordnung. Alle Begrifflichkeiten bauen dabei auf den Definitionen von PETERS (2015: 345) sukzessiv auf.

An erster Stelle steht die Definition der partiellen Ordnung (siehe Formel 14) für eine zweistellige Relation mit den beiden Eigenschaften der Reflexivität sowie der Transitivität.

Die Reflexivität beschreibt hierbei, dass jedes Element des Vektorraums auch in Relation zu sich selbst steht. Dagegen stellt die Transitivität die Beziehung dreier beliebiger Elemente x, y, z aus dem Vektorraum $\mathbb{R}^k$ zueinander dar. Demnach steht das Element x über y in Relation mit z, wenn x mit y und y mit z in Beziehung steht.Die Notation $x \succ y$ indiziert hierbei $x \succcurlyeq y \ \wedge x \neq y$.

Formel 14: Definition einer partiellen Ordnung

$Eine\ partielle\ Ordnung\ x \succcurlyeq y\ auf\ dem\ reellen\ Vektorraum\ \mathbb{R}^k$
$der\ Dimension\ k\ ist\ eine\ zweistellige\ Relation, welche$
$die\ Eigenschaften\ der\ Reflexivität\ und\ der\ Transitivität\ erfüllt.$

(1) *Reflexivität:*

$Für\ alle\ \ x \in \mathbb{R}^k\ gilt\ x \succcurlyeq x$

(2) *Transitivität:*

$Für\ alle\ x, y, z \in \mathbb{R}^k mit\ x \succcurlyeq y\ \wedge\ y \succcurlyeq z\ folgt\ x \succcurlyeq z$

Bleibt zusätzlich bei der partiellen Ordnung die Eigenschaft der Vollständigkeit gewahrt, kann die zugrundeliegende Relation als schwache Ordnung bezeichnet werden.

Unter Vollständigkeit wird hier verstanden, dass jedes Element x mit jedem anderen beliebigen Element y aus dem Vektorraum $\mathbb{R}^k$ vergleichbar ist (bzw. y mit x).

Formel 15: Definition einer schwachen Ordnung

$Eine\ partielle\ Ordnung\ ist\ eine\ schwache\ Ordnung, wenn\ zusätzlich\ die$
$Vollständigkeit\ erfüllt\ ist.$

Vollständigkeit:
$Für\ alle\ x, y \in \mathbb{R}^k\ mit\ x \neq y\ folgt\ x \succcurlyeq y\ \vee\ y \succcurlyeq x$:

Aufbauend auf der schwachen Ordnung ist eine lineare Ordnung gegeben, wenn zusätzlich die Antisymmetrie erfüllt ist und somit für den Fall, dass x in Relation mit y steht, nicht zugleich die Umkehrung gilt, außer falls x und y gleich sind.

Formel 16: Definition einer linearen Ordnung

$$Eine\ schwache\ Ordnung\ ist\ eine\ lineare\ Ordnung, wenn\ zus\ddot{a}tzlich\ die\ Antisymmetrie\ erf\ddot{u}llt\ ist.$$

$$\underline{Antisymmetrie}:$$
$$F\ddot{u}r\ alle\ x, y \in \mathbb{R}^k\ mit\ x \succcurlyeq y\ \wedge\ y \succcurlyeq x\ gilt\ x = y$$

Letztendlich ist eine lineare Ordnung eine lexikalische Ordnung $\succcurlyeq_{lex}$ auf $\mathbb{R}^k$, falls für zwei Vektoren $x, y \in \mathbb{R}^k$ x lexikographisch größer oder gleich y ist (Notation: $x \succcurlyeq_{lex} y$), wenn entweder $x = y$ oder $x \neq y$. Für $i = \min\{j \in \{1, \dots, k\} | x_j \neq y_j\}$ ist $x_i > y_i$.

Mit diesem Verständnis der lexikalischen Ordnung lassen sich die später in Abschnitt 4.2.1 vorgestellten Überschüsse des Nucleolus bzw. die zu minimierenden Strukturvariablen derer Derivate im Allgemeinen in eine Ordnung versetzen.

3 Implementierte Spieleigenschaften und spezielle Spiele

In diesem Abschnitt wird eine Auswahl der im Rahmen der Masterarbeit implementierten Spieleeigenschaften und speziellen Spiele aufgeführt.

3.1 Duales Spiel

Ein Spiel (N, v) korrespondiert mit einem sich daraus abgeleiteten dualen Spiel (N, v^*), welches viele Äquivalenzen sowie gemeinsame Eigenschaften aufzeigt. So ist ein Kostenspiel (N, c) genau dann konvex, wenn dessen dualer Pendant konkav ist und umgekehrt (BILBAO 2000: 106). Auch teilt sich ein (Profit-)Spiel der Form (N, v) mit dessem korrespondierendem dualem Spiel (es handelt sich hierbei um ein Kostenspiel) denselben identischen Kern, so gilt nach BILBAO (2000: 4) $C(N, v) = C(N, v^*)$.

Ein duales Spiel ist hierbei nach PELEG & SUDHÖLTER (2007: 125) wie in Formel 17 formal definiert.

Formel 17: Definition eines dualen Spiels

$$Sei\ (N, v)\ ein\ Spiel, dann\ gilt\ für\ das\ zugehörige\ duale\ Spiel\ (N, v^*):$$

$$v^*(S) = v(N) - \mathrm{v(N\backslash S)}, \text{für alle S} \subseteq \mathrm{N}$$

Die Koalitionsfunktion des dualen Spiels $v^*(S)$ ordnet für ein wesentliches Spiel jeder Koalition den Wert zu, der ihr dadurch entgeht, dass sie sich nicht an der großen Koalition N mitbeteiligt.

Aufgrund der Äquivalenzen und gemeinsamen Eigenschaften beider Spiele wird das duale Spiel oft zur Analyse und für die Umsetzung verschiedener Konzepte in `CoopGame` verwendet.

3.2 Kosten- vs. Kostenersparnisspiel

Bei einem Kostenspiel (N, c) handelt es sich um ein Spiel, das nicht durch eine Koalitionsfunktion v, sondern durch eine Kostenfunktion c charakterisiert wird (PELEG & SUDHÖLTER 2007: 14).

Wie auch die Koalitionsfunktion, ordnet auch die Kostenfunktion jeder Koalition einen reellen Wert zu, nur handelt es sich in diesem Fall nicht um einen erwirtschaften Ertrag, sondern um Kosten, die jede Koalition zu stemmen hat.

Ein solches Kostenspiel (N, c) lässt sich jedoch in ein Spiel der Form (N, v) überführen, welches wieder durch eine Koalitionsfunktion v bestimmt ist; hierzu werden die Ersparnisse bei Koalitionsbildungen betrachtet (PELEG&SUDHÖLTER 2007: 14).

Formel 18: Definition der Transformation eines Kosten- in ein Kostenersparnisspiel

$$v(S) = \sum_{i \in S} c(\{i\}) - c(S), \text{für alle } S \subseteq N$$

Ein einfaches Beispiel ist ein Kostenspiel mit zwei Hausbesitzern, die beide 1000 Liter Heizöl benötigen. Kauft jeder für sich allein das Heizöl hat er, mit den Transportkosten von 100 € und 50 Cent pro Liter, Kosten von 600 € zu tragen. Im Verbund kostet sie aber das Öl und der Transport 1100€. Für dieses Kostenspiel (N, c) ergibt sich somit ein Kostenvektor von $c = (600{,}600{,}1100)$.

Durch Verwendung von Formel 18 lässt sich das Kostenspiel (N, c) allerdings in ein Kostenersparnisspiel mit Koalitionsfunktion transformieren (siehe Formel 19).

Formel 19: Beispiel für Transformation eines Kosten- in ein Kostenersparnisspiel

$$v(\{1\}) = \sum_{i \in \{1\}} c(\{i\}) - c(\{1\}) = 600 - 600 = 0$$

$$v(\{2\}) = \sum_{i \in \{2\}} c(\{i\}) - c(\{2\}) = 600 - 600 = 0$$

$$v(\{1{,}2\}) = \sum_{i \in \{1,2\}} c(\{i\}) - c(\{1{,}2\}) = (600 + 600) - 1100 = 100$$

Durch die Schritte in Formel 19, liegt mit $v(\{1\}) = v(\{2\}) = 0$ und $v(\{1{,}2\})$ nun ein Kostenersparnisspiel der Form (N, v) vor, auf welches sich alle in `CoopGame` implementierten Lösungskonzepte anwenden lassen. Für die Form (N, c) war dies zuvor nicht möglich.

3.3 Balanciertheit

Bei der Balanciertheit eines Spiels handelt es sich um eine wichtige Spieleigenschaft, gibt sie doch darüber Aufschluss, ob der Kern (siehe Abschnitt 0) eines Spiels leer ist.

Die Eigenschaft beruht auf dem Bondareva-Shapley Theorem, welche die notwendigen Anforderungen eines Spiels für einen nichtleeren Kern beschreibt.

Ein Spiel mit einem nichtleeren Kern wird als balanciert bezeichnet (PELEG & SUDHÖLTER 2007: 27)

Das folgende Theorem wurde dabei unabhängig voneinander durch BONDAREVA (1963) sowie SHAPLEY (1967) aufgestellt (PELEG & SUDHÖLTER 2007: 30) und wird im Weiteren kurz beschrieben.

3.3.1 Das Bondareva-Shapley Theorem

Bevor der Begriff der Balanciertheit in Gänze erklärt werden kann, muss zuerst der in dem Zusammenhang verwendete Begriff des ausgewogenen Mengensystems (engl. balanced collection) erörterte werden – siehe hierzu die Definition nach PELEG & SUDHÖLTER (2007: 38) in Formel 20.

Formel 20: Definition eines ausgewogenen Mengensystems

Sei $B \subset 2^N \backslash \{\emptyset\}$ eine Mengensystem aus nicht leerer Koalitionen
der Spielermenge N.
Dieses Mengensystem ist balanciert,
falls Koeffizienten $\lambda_S > 0$ für $S \in B$ existieren,
so dass für jeden Spieler $i \in N$

$\sum_{S \in B: i \in S} \lambda_S = 1$

$erf\ddot{u}llt\ ist.$

Nach der Definition in Formel 20 wird für jeden Spieler $i \in N$ ein Mapping durchgeführt, bei der jeder Koalition, an der Spieler i beteiligt ist, ein positives Gewicht zugeteilt wird (andernfalls 0).

Bei den durchgeführten Mappings muss es sich außerdem um balancierte Zuordnungen handeln, d.h. die Summe der Gewichte beläuft sich für jedes Mapping auf 1.

Mithilfe des Begriffs des ausgewogenen Mengensystems wird im Folgenden (siehe Formel 21) das Bondareva-Shapley-Theorem nach PELEG & SUDHÖLTER (2007: 39) vorgestellt.

Formel 21: Definition Bondareva-Shapley Theorem

Für jedes Spiel $v \in G^N\ gilt$: $C(v) \neq \emptyset$ g. d. w. jede ausgewogene

Mengensystem (balanced collection) B $\subset 2^N$

mit Koeffizienten $\{\lambda_S | S \in B\}, die\ f\ddot{u}r\ Ausgewogenheit\ sorgen,$

$$\sum_{S \in B} \lambda_S v(S) \leq v(N)$$

$erf\ddot{u}llt.$

Ein Spiel ist demnach balanciert und verfügt über einen nichtleeren Kern, wenn für jedes ausgewogene Mengensystem (siehe Formel 20) die Summe aus den Balance Koeffizienten multipliziert mit den Koalitionswerten kleiner gleich dem Ertrag der großen Koalition ist.

Die Balanciertheit soll im Weiteren an einem einfachen Beispiel mit nichtleerem Kern veranschaulicht werden.

3.3.2 Beispiel für ein balanciertes Spiel

Gegeben sei ein Dreipersonenspiel mit $N = \{1, 2, 3\}$ und der Koalitionsfunktion v, die wie folgt definiert ist:

Formel 22: Spielvektor für Balanciertheit Beispiel

Werte der Einerkoalitonen:

$$v(1) = 1$$

$$v(2) = v(3) = 2$$

Werte der Zweierkoalitionen:

$$v(1,2) = 4;\ v(1,3) = 3;\ v(2,3) = 4$$

Wert der Großen Koalition:

$$v(123) = 6$$

Nach der Formel 20 für ein ausgewogenes Mengensystem, lässt sich für jeden Spieler i über den charakteristischen Vektor (siehe Abschnitt 2.1.5) ein allgemeines Mapping M_i spezifizieren. Die Spalten der Bitmatrix entsprechen hier jeweils den charakteristischen Vektoren der einzelnen Spieler.

Tabelle 1: Beispiel für die Bestimmung eines ausgewogenen Mengensystems

Index	**Bit-Zeile für S**	**Spieler $\in S$**	M_1	M_2	M_3	$v(S)$
1	1 0 0	{1}	λ_1	0	0	1
2	0 1 0	{2}	0	λ_2	0	2
3	0 0 1	{3}	0	0	λ_3	2
4	1 1 0	{1,2}	λ_4	λ_4	0	4
5	1 0 1	{1,3}	λ_5	0	λ_5	3
6	0 1 1	{2,3}	0	λ_6	λ_6	4
7	1 1 1	{1,2,3}	λ_7	λ_7	λ_7	6

Für ein balanciertes Mapping muss die Summe der Koeffizienten 1 ergeben. Hiermit gelten die in Formel 23 beschriebenen Gleichheitsnebenbedingungen

Formel 23: Gleicheitsnebenbedingungen für ausgewogenes Mengensystem im Dreipersonenfall

$$\begin{array}{cccccccccc} \lambda_1 & +0 & +0 & +\lambda_4 & +\lambda_5 & +0 & +\lambda_7 & = & 1 \\ 0 & +\lambda_2 & +0 & +\lambda_4 & +0 & +\lambda_6 & +\lambda_7 & = & 1 \\ 0 & +0 & +\lambda_3 & +0 & +\lambda_5 & +\lambda_6 & +\lambda_7 & = & 1 \end{array}$$

Unter Berücksichtigung der Gleichheitsnebenbedingungen aus Formel 23 gilt es, abgeleitet von Formel 21 zum Bondareva-Shapley Theorem, das Maximum des Terms $\sum_{S\in B} \lambda_S v(S)$ als Zielfunktion durch ein Lineares Programm zu identifizieren.

Hieraus ergibt sich für das Beispiel und dessen Koalitionswerte folgende zu maximierende Zielfunktion.

Formel 24: Zielfunktion für Bestimmung Balanciertheit

$$Maximiere$$

$$z = F(\alpha_1, \dots, \alpha_n) = \lambda_1 + 2\lambda_2 + 2\lambda_3 + 4\lambda_4 + 3\lambda_5 + 4\lambda_6 + 6\lambda_7$$

Nachdem der Zielfunktionswert z durch Lineare Programmierung bestimmt wurde, müssen entsprechend $\sum_{S\in B} \lambda_S v(S) \leq v(N)$ aus Formel 21 mit $z \leq v(N)$ sowie $z > v(N)$ zwei Fälle gesondert unterschieden werden.

Für den ersten Fall $z \leq v(N)$ zeigt das Bondareva-Shapley Theorem, dass der Kern eines solchen Spiels nichtleer und das Spiel balanciert ist. Statt auf $z \leq v(N)$ kann aber hier auch lediglich auf $z = v(N)$ geprüft werden, da $z < v(N)$ nicht eintreten kann – ist $\lambda_1 = \lambda_2 = \lambda_3 = \lambda_4 = \lambda_5 = \lambda_6 = 0$ und $\lambda_7 = 1$ doch für alle Spieler i ein zulässiges balanciertes Mapping, das zu $z = v(N)$ führt.

Für den anderen Fall $z > v(N)$ lehrt das Bondareva-Shapley Theorem, dass das Spiel nicht balanciert und der Kern leer ist.

Im vorliegenden Beispiel wird $z = 6 = v(N)$ unter Berücksichtigung der Gleichheitsnebenbedingungen in `R` über das Paket `glpkAPI` (siehe Abschnitt 5.3.1.1) bestimmt, das Spiel ist somit balanciert.

Auf die Umsetzung der Prüfung der Balanciertheit in `R` wird hierbei noch nicht eingegangen. Diese erfolgt aber später an geeigneter Stelle im Zusammenhang mit der Vorstellung des exemplarischen Einsatzes von `GLPK` und `glpkAPI` zur Lösung Linearer Programme (siehe Abschnitt 5.3.1.2).

3.4 Einmütigkeitskoeffizienten

Die in Abschnitt 2.1.9 kurz erörterten Einmütigkeitsspiele können anstatt der Einheitsspiele (siehe Formel 25) bei $T \neq \emptyset$ als Basis des Spielvektorraums G_N mit $\mathbb{R}^{2^n-1}$ herangezogen werden WIESE (2005: 123).

Formel 25: Definition Einheitsspiel

$$Ein\ Einheitsspiel\ ist\ ein\ Spiel\ mit\ einer\ nichtleeren\ Teilmenge\ T\ von\ N\ f\ddot{u}r\ das\ gilt:$$

$$v_T(S) = \begin{cases} 1, & T = S \\ 0, & T \neq S \end{cases}$$

Für den Beweis, dass die Einmütigkeitsspiele tatsächlich eine Basis des $\mathbb{R}^{2^n-1}$ bilden, sei hierbei auf WIESE (2005: 123) verwiesen.

WIESE (2005: 124) verweist bei der Herleitung der Formel für die Einmütigkeitskoeffizienten des Weiteren weiter auf SHAPLEY (1953).

Formel 26: Definition Einmütigkeitskoeffizienten

$$\lambda_T(v) = \sum_{K \in 2^T \setminus \{\emptyset\}} (-1)^{t-k} v(K), f\ddot{u}r\ t = |T| und\ k = |K|$$

Ein einfaches Beispiel (siehe Formel 27) zu den Einmütigkeitskoeffizienten wurde aus SLIKKER & NOUWELAND (2001: 7) entnommen, vgl. hierzu auch WIESE (2005: 124f).

Formel 27: Spielevektor für Einmütigkeitskoeffizienten Beispiel

Werte der Einerkoalitionen:

$$v(1) = v(2) = v(3) = 0$$

Werte der Zweierkoalitionen:

$$v(1,2) = 60; v(1,3) = 48;$$
$$v(2,3) = 30$$

Werte der Großen Koalition:

$$v(1,2,3) = 72$$

Für dieses Spiel werden die Einmütigkeitskoeffizienten im Folgenden entsprechend mit Formel 26 ermittelt (siehe Formel 28).

Formel 28: Beispiel für Berechnung der Einmütigkeitskoeffizienten

Einmütigkeitskoeffizienten für $\lambda_{\{1\}}, \lambda_{\{2\}}, \lambda_{\{3\}}$:

$$\lambda_{\{1\}}(v) = \lambda_{\{2\}}(v) = \lambda_{\{3\}}(v) = 0$$

Einmütigkeitskoeffizienten für $\lambda_{\{2,3\}}$:

$$\lambda_{\{2,3\}}(v) = \sum_{K \in 2^{\{2,3\}} \setminus \{\emptyset\}} (-1)^{|\{2,3\}|-|K|} v(K)$$
$$= (-1)^{2-1} v(\{2\}) + (-1)^{2-1} v(\{3\}) + (-1)^{2-2} v(\{2,3\})$$
$$= 0 + 0 + 30 = 30$$

Einmütigkeitskoeffizienten für $\lambda_{\{1,2\}}$:

$$\lambda_{\{1,2\}}(v) = \sum_{K \in 2^{\{1,2\}} \setminus \{\emptyset\}} (-1)^{|\{1,2\}|-|K|} v(K)$$
$$= (-1)^{2-1} v(\{1\}) + (-1)^{2-1} v(\{2\}) + (-1)^{2-2} v(\{1,2\})$$
$$= 0 + 0 + 60 = 60$$

Einmütigkeitskoeffizienten für $\lambda_{\{1,3\}}$:

$$\lambda_{\{1,3\}}(v) = \sum_{K \in 2^{\{1,3\}} \setminus \{\emptyset\}} (-1)^{|\{1,3\}|-|K|} v(K)$$
$$= (-1)^{2-1} v(\{1\}) + (-1)^{2-1} v(\{3\}) + (-1)^{2-2} v(\{1,3\})$$
$$= 0 + 0 + 48 = 48$$

Einmütigkeitskoeffizienten für $\lambda_{\{1,2,3\}}$:

$$\lambda_{\{1,2,3\}}(v) = \sum_{K \in 2^{\{1,2,3\}} \setminus \{\emptyset\}} (-1)^{|\{1,2,3\}|-|K|} v(K)$$
$$= (-1)^{3-1} v(\{1\}) + (-1)^{3-1} v(\{2\}) + (-1)^{3-1} v(\{3\}) + (-1)^{3-2} v(\{1,2\}) + (-1)^{3-2} v(\{1,3\}) + (-1)^{3-2} v(\{2,3\}) + (-1)^{3-3} v(\{1,2,3\})$$
$$= 0 + 0 + 0 - 60 - 48 - 30 + 72 = -66$$

Auf Grundlage der bestimmten Einmütigkeitskoeffizienten ergibt sich der Spielevektor als Linearkombination mit den Einmütigkeitsvektor-Basen und den bestimmten Koeffizienten.

Formel 29: Spiel als Linearkombination mit den Basen aus den Einmütigkeitskoeffizienten

$$\lambda_{\{1,2\}}(v)v(\{1,2\}) + \lambda_{\{1,3\}}(v)v(\{1,3\}) + \lambda_{\{2,3\}}(v)v(\{1\}) + \lambda_{\{1,2,3\}}(v)v(\{1\})$$
$$= 60 \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} + 48 \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} + 30 \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} - 66 \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 60 \\ 48 \\ 30 \\ 72 \end{pmatrix}$$

Für das Paket `CoopGame` wurde im Rahmen dieser Masterarbeit mit der Funktion `linearCombinationUnanimity` die Funktion für die Bestimmung der Einmütigkeitskoeffizienten zu einem Spiel geschaffen. Dieser Basenwechsel kam bei mehreren Implementierungen in `CommunicationGames` zum Einsatz.

```
> A<-c(0,0,0,60,48,30,72)                                    #1
> linearCombinationUnanimity(A)                              #2
[1]  0  0  0 60 48 30 -66                                    #3
```

Codebsp. 2: Bestimmung der Einmütigkeitskoeffizienten

4 Punktwertige Lösungskonzepte für Kooperationsspiele

Für Koalitionsspiele lassen sich zahlreiche Lösungskonzepte anführen, wobei sich prinzipiell zwischen punkt- und mengenwertigen Ansätzen unterscheiden lässt.

Dienen Mengenansätze dazu, die Auszahlungsmenge auf eine Teilmenge mit bestimmten Charakteristika zu reduzieren, streben Wertansätze die Herleitung einer eindeutigen Lösung an.

Mengenansätze geben den Spielern dabei rationale Entscheidungsrahmen mit einem mehr oder weniger großen Maß an Unbestimmtheit vor, wohingegen Wertansätze den Spielern eindeutige Handlungsdirektiven liefern.

Im Rahmen dieser Arbeit wurden zu den bereits bestehenden – wie dem Kern sowie der Weber- und Imputationsmenge – keine neuen Mengenansätze erarbeitet, da der Schwerpunkt gezielt auf die Erweiterung des Pakets `CoopGame` um neue Wertansätze gelegt wurde.

Mit dem Banzhaf, Deegan Packel, Gately, Johnston, Public Good, Public Help, Shapley und τ −Wert stehen im Packet zum jetzigen Entwicklungsstand zahlreiche einfache punktwertige Konzepte zur Verfügung, die auf Korrektheit überprüft bzw. neu aufgenommen wurden.

Des Weiteren enthält das Paket mit Abschluss der Masterarbeit die in der Berechnung aufwendigeren Konzepte wie den Nucleolus und verwandter Ansätze wie den Disruption Nucleolus, Modiclus, Per Capita Nucleolus, Prenucleolus, Proportional Nucleolus und Simplified Modiclus.

Auch das unter Berücksichtigung der Ergebnisse von GEBELE (2015) erstellte Framework zur Berechnung der Nucleolus Derivate und deren Implementierung wird in diesem Kapitel noch thematisiert; hier sei auf das Kapitel 5 verwiesen.

Im nächsten Abschnitt wird auf die einfachen punktwertigen Konzepte eingegangen und anschließend im darauffolgenden Unterkapitel auf das anspruchsvollere Konzept des Nucleolus und dessen Derivaten.

4.1 Einfache Punktwertige Konzepte

Mit dem Banzhaf, Deegan Packel, Gately, Johnston und Public Help Index standen zahlreiche Machtindizes in `CoopGame` zur Verfügung, die aus einem Projekt unter der Leitung von Prof. Dr. Staudacher im Wintersemester 2015/2016 hervorgingen.

Diese Konzepte galt es im Rahmen dieser Masterarbeit hinsichtlich der vorgesehenen Veröffentlichung auf `CRAN` erneut zu prüfen und wo nötig zu modifizieren.

Die Machtindizes wurden im Rahmen dieser Masterarbeit zusätzlich als punktwertige Ansätze umgedeutet. Wie auch beim Shapley Wert können auch deren Ergebnisse als Erwartungswert interpretiert werden.

Darüber hinaus sollten auch weitere neue Konzepte wie der Public Help und τ-Wert fest in `CoopGame` verankert werden.

Im Folgenden werden die bereits im Paket vorhandenen Lösungskonzepte des Shapley-Werts im besonderen Hinblick auf seine Sonderstellung in der Spieltheorie und der Public Good hinsichtlich der Analogien zu dem ebenfalls behandelten Public Help Wert aufgeführt.

Im Anschluss folgt eine Vorstellung des τ-Werts und des Gately-Punkts.

4.1.1 Shapley Wert

Beim Shapley-Wert handelt es sich um ein sehr weit verbreitetes Lösungskonzept in der kooperativen Spieltheorie.

Er wurde durch SHAPLEY (1953) vorgestellt und findet seither auch im praktischen Umfeld vermehrt Verwendung. Hierbei stößt er durchaus auf positive Resonanz.

Nach HOLLER & ILLING (2006: 305f) betrachtet das Maß dabei im Wesentlichen nicht Koalitionen als solche, sondern die verschiedenen Permutationen der Spieler, welche das Eintreten von Spielern in bestimmten Entscheidungssituationen widerspiegelt. In diesen spezifischen Situationen wird der Einfluss des Spielers auf das Spiel untersucht. Bezogen auf alle Spieler und Entscheidungssituationen ergibt sich durch Berücksichtigung der unterschiedlichen Einflussstärken der Spieler der Shapley-Wert nach der im Folgenden vorgestellten Definition.

4.1.1.1 Festlegung des Shapley Wert

Der Shapley-Value eines Spielers $i \in N$ und das Spiels v ist wie in Formel 30 (HOLLER & ILLING 2006: 304) definiert.

Formel 30: Definition Shapley Value

$$\phi_i(v) = \sum_{K \ni i; K \subset N} \frac{(k-1)!\,(n-k)!}{n!} [v(K) - v(K - \{i\})]$$

$$mit \sum \phi_i(v) = v(N)$$

Hierbei drückt k die Anzahl der an der Koalition K beteiligten Spieler $k = card(K)$ und n die insgesamt involvierten Spieler $n = card(N)$ aus.

Der Ausdruck $[v(K) - v(K - \{i\})]$ mit $i \in K$ beschreibt den marginalen Beitrag des Spielers zur Koalition k, also den Mehrwert, den die Koalition durch diesen Spieler erfährt.

Wie oft die Koalition K innerhalb der $n!$ Permutationen auftritt, gibt der Term $\frac{(k-1)!(n-k)!}{n!}$ wieder. Dies soll an einem kleinen Beispiel hergeleitet werden.

Gegeben sei ein Fünfpersonenspiel (n=5) mit K=$\{k_1, k_2, k_3\}$, in dem für Spieler i=3 untersucht werden soll, wie oft dieser an einer solchen Konstellation teil hat.

Zur Veranschaulichung werden hier die drei Elemente aus K als k_1, k_2, k_3 und die nicht an K beteiligten Spieler als r_1, r_2 bezeichnet.

Tabelle 2 Permutationsbeispiel a)

Platz 1	Platz 2	Platz 3	Platz 4	Platz 5
k_1	k_2	$k_{i=3}$	r_1	r_2

Ist $k_{i=3}$ immer an dritter Stelle vertreten, ist es bei den anderen Spielern k_1, k_2 für das Zustandekommen der Koalition K unerheblich, in welcher Reihenfolge sie die Plätze 1 und 2 belegen und auch analog für die Spieler r_1, r_2 bezüglich der Plätze 5 und 7.

Hier ergeben sich für die Elemente k_1, k_2 nach $(k-1)! = (3-1)! = 2$ sowie für r_1, r_2 nach $(n-k)! = (5-3)! = 2$ jeweils zwei verschiedene Anordnungen und somit insgesamt $2 * 2 = 4$ mögliche Permutationen, die zur Koalition K führen.

Tabelle 3: Permutationsbeispiel b)

Platz 1	Platz 2	Platz 3	Platz 4	Platz 5
k_1	k_2	$k_{i=3}$	r_1	r_2
k_1	k_2	$k_{i=3}$	r_2	r_1
k_2	k_1	$k_{i=3}$	r_1	r_2
k_2	k_1	$k_{i=3}$	r_2	r_1

Bei einer Anzahl möglicher Permutationen von $n! = 5! = 120$ erschließt sich für die Koalition K ein Anteil von $\frac{1}{30}$ mit $\frac{(k-1)!(n-k)!}{n!} = \frac{(3-1)!(5-3)!}{5!} = \frac{4}{120} = \frac{1}{30}$.

Für jeden Spieler bildet sich der Shapley Wert durch Summenbildung von seinen gewichteten marginalen Beiträgen zu den Koalitionen, an denen er beteiligt ist.

4.1.1.2 Beispiel für Shapley Wert

Für das Public Good Wert Beispiel dient ein Abstimmungsspiel (siehe Abschnitt 2.1.11) mit vier Spielern und ihren Gewichten $w = (35,21,15,15)$ sowie der zu erfüllenden Quote von $q = 51$.

Hieraus ergeben sich die in Formel 31 beschriebenen Koalitionswerte.

Formel 31: Spielevektor für das Shapley Wert Beispiel

Werte der Einerkoalitonen:

$$v(1) = v(2) = v(3) = v(4) = 0$$

Werte der Zweierkoalitionen:

$$v(1,2) = 1; v(1,3) = v(1,4) = 0$$
$$v(2,3) = v(2,4) = 0$$
$$v(3,4) = 0$$

Werte der Dreierkoalitionen:

$$v(1,2,3) = v(1,2,4) = v(1,3,4) = 1$$
$$v(2,3,4) = 1$$

Wert der Großen Koalition:

$$v(1,2,3,4) = 1$$

Die 24 verschiedenen Permutationen, die sich im Vierspielerfall ergeben und die marginalen Beiträge MB_i für jeden Spieler und für jede Permutation lassen sich, in Anlehnung an das Dreipersonenbeispiel von HOLLER & ILLING (2006: 308), wie folgt (siehe Tabelle 4) veranschaulichen.

Tabelle 4: Permutationen und marginale Beiträge

Index	**Permutation**	$\boldsymbol{MB_1}$	$\boldsymbol{MB_2}$	$\boldsymbol{MB_3}$	$\boldsymbol{MB_4}$
1	1 2 3 4	–	**1**	–	–
2	1 2 4 3	–	**1**	–	–
3	1 3 2 4	–	**1**	–	–
4	1 3 4 2	–	–	–	**1**
5	1 4 2 3	–	**1**	–	–
6	1 4 3 2	–	–	**1**	–
7	2 1 3 4	**1**	–	–	–
8	2 1 4 3	**1**	–	–	–
9	2 3 1 4	**1**	–	–	–
10	2 3 4 1	–	–	–	**1**
11	2 4 1 3	**1**	–	–	–
12	2 4 3 1	–	–	**1**	–

13	3 1 2 4	–	**1**	–	–
14	3 1 4 2	–	–	–	**1**
15	3 2 1 4	**1**	–	–	–
16	3 2 4 1	–	–	–	**1**
17	3 4 1 2	**1**	–	–	–
18	3 4 2 1	–	**1**	–	–
19	4 1 2 3	–	**1**	–	–
20	4 1 3 2	–	–	**1**	–
21	4 2 1 3	**1**	–	–	–
22	4 2 3 1	–	–	**1**	–
23	4 3 1 2	**1**	–	–	–
24	4 3 2 1	–	**1**	–	–
	Zahl des Pivots	8	8	4	4
	Shapley-Wert	$\frac{8}{24} = \frac{1}{3}$	$\frac{8}{24} = \frac{1}{3}$	$\frac{4}{24} = \frac{1}{6}$	$\frac{4}{24} = \frac{1}{6}$

Da es sich bei einem Abstimmungsspiel um ein einfaches Spiel handelt, bei dem der marginale Beitrag MB_i eines Spielers i entweder auf 1 oder 0 zu beziffern ist, reicht es für die Bestimmung des Shapley Punkts in diesem einfachen Beispiel aus, in Erfahrung zu bringen, wie oft Spieler i als Pivotspieler in Erscheinung tritt. Bei einem Pivotspieler handelt es sich dabei um einen kritischen Spieler, der durch seinen Beitritt zu einer Koalition K ausschlaggebend dafür ist, ob aus einer Verlust- ($v(K) = 0$) eine Gewinnkoalition ($v(K) = 1$) wird (vgl. hierzu spätere formale Definition in Formel 32).

Der Shapley Value ergibt sich in einem einfachen Spiel für einen Spieler durch Division der Anzahl seiner Pivots durch die Gesamtanzahl der Permutationen.

Für das gegebene Beispiel und mit der Anzahl der Spielerpivots von (8,8,4,4), ergeben sich die Shapley Values von $\phi_1 = \phi_2 = \frac{1}{3}$ und $\phi_3 = \phi_4 = \frac{1}{6}$.

```
> A<-generateGameVector(cFuncQuota,n=4,w=c(35,21,15,15),q=51)          #1
> shapleyValue(A)$shapleyValue                                        #2
[1] 0.3333333 0.3333333 0.1666667 0.1666667                           #3
```

Codebsp. 3: Shapley Wert Beispiel

4.1.2 Public Good Wert

Beim Public Good Wert (PGV) handelt es sich um ein eigenes – anhand des Public Good Index (PGI) definiertes – punktwertiges Lösungskonzept.

Der PGI ist ein Machtindex, der jedem Spieler einen Wert zwischen 0 und 1 beimisst, wobei sich die Summe der Machtindizes aller Spieler auf 1 beläuft.

Dem PGI Konzept liegt hierbei die Kollektivgutannahme zugrunde (Holler & Illing 2006: 325). Die einzelnen Präferenzen der jeweiligen Spieler werden dabei vernachlässigt, da im Fokus allein die Ermittlung des jeweiligen Machtumfangs des Spielers und nicht die Erfüllung seiner Interessen steht.

Holler & Illing (2006: 325) beschreiben hierbei ein Kollektivgut als ein Gut, das uneingeschränkt allen Spielern einer Koalition in Gänze und gleichermaßen nutzt. Hier entfällt die sonst übliche Aufteilungsproblematik vollständig. Als Beispiele können hierbei Abstimmungen in Parlamenten und Gremien herangezogen werden.

Als Konsequenz der Kollektivgutannahme (Holler & Illing 2006: 325) bezieht sich der PGI ausschließlich auf Minimumgewinnkoalitionen $M(v)$. Er wird im Folgenden kurz spezifiziert.

4.1.2.1 Festlegung des Public Good Index

Vor der Umdisponierung des PGI zum Wertkonzept wird in diesem Abschnitt erläutert, wie sich dieser zusammensetzt.

Weil dem PGI die Minimumgewinnkoalitionen und Gewinnkoalitionen zu Grunde liegen, sind diese für das PGI Konzept unabdingbar und werden kurz charakterisiert.

Bei Gewinnkoalitionen handelt es sich nach WIESE (2005: 91) um sämtliche Koalitionen K, die bei einem einfachen Spiel (z.B. einem Abstimmungsspiel) einen Koalitionswert von 1 aufweisen, bei Koalitionen K mit $v(K) = 0$ entsprechend um Verlustkoalitionen. Als verallgemeinerte Abwandlung der Definition einer Gewinnkoalition v für einfache Spiele nach WIESE (2005: 91), lässt sich die Menge aller Gewinnkoalitionen – bezogen auf alle Spiele (N, v) – wie in Formel 32 definieren.

Formel 32: Definition Gewinnkoalitionen

$$W(v) = \{K \subseteq N \mid v(K) > 0\}$$

Der Ausdruck $W_i(v)$ beschreibt hierbei die Untermenge der Gewinnkoalitionen, an denen der Spieler $i \in N$ beteiligt ist.

Formel 33: Definition Gewinnkoalitionen mit Beteiligung des Spielers $i \in N$

$$W_i(v) = \{K \subseteq N \mid v(K) > 0 \ \wedge i \in K\}$$

Minimumgewinnkoalitionen sind Koalitionen, die ausschließlich aus kritischen Spielern zusammengesetzt sind. Fällt hier nur ein einzelner Spieler weg, wird aus der Gewinnkoalition eine Verlustkoalition, bei der kein gemeinsamer Wert erwirtschaftet wird. Minimumgewinnkoalitionen lassen sich wie in Formel 34 definieren (Holler & Illing 2006: 323).

Formel 34: Definition Minimumgewinnkoalitionen

$$M(v) = \{K \subseteq N \mid v(K) > 0 \ und \ v(L) = 0, \forall L \subset K\}$$

Die Summe aller Koalitionswerte v(K) für alle $K \in M(v)$ und den Spieler $i \in K$ wird im Folgenden nach Formel 35 als $c_i(v)$ notiert.

Formel 35: Summe aller Koalitionswerte $v(K)$ der Minimumgewinnkoalitionen bei Public Good Index

$$\mathrm{c_i(v)} = \sum_{\mathrm{K \ni i; K \in M(v)}} \mathrm{v(K)}$$

Aus der Kollektivgutannahme und der getroffenen Beschränkung auf Minimumgewinnkoalitionen leitet sich der PGI daher nach der Formel 34 und Formel 35, wie in Formel 36 definiert, ab (Holler & Illing 2006: 325f).

Formel 36: Definition Public Good Index

$$PGI_i(v) = \frac{c_i(v)}{\sum_{j \in N} c_j(v)} \text{ und } \sum_{j \in N} PGI_i(v) = 1$$

In Formel 36 ist für jeden Spieler $i \epsilon N$ das Verhältnis der Summe aller Koalitionswerte aus Minimumgewinnkoalitionen, an denen er beteiligt ist, zu der Summe sämtlicher Koalitionswerte aus Minimumgewinnkoalitionen definiert.

4.1.2.2 Der Public Good Index als Grundlage für ein Wertkonzept

Entsprechend dem Shapley Wert ϕ mit $\sum \phi_i(v) = v(N)$ ist die kollektive Rationalität gefordert. Analog hierzu wird dieselbe Forderung (siehe Formel 37) auch an den Public Good Value PGV formuliert.

Formel 37: Forderung nach kollektiver Rationalität für Public Good Value

$$\sum PGV_i(v) = v(N)$$

Abgeleitet von der Forderung nach kollektiver Rationalität (siehe Formel 37), lässt sich der Public Good Wert dann durch Skalierung der Public Good Indizes mit dem Koalitionswert der großen Koalition N definieren (siehe Formel 38).

Formel 38: Definition Public Good Value

$$PGV_i(v) = PGI_i(v) * v(N)$$

Eine solche Umdeutung eines Machtindex als punktwertiges Lösungskonzept birgt jedoch zahlreiche Fallstricke und gibt durchaus Anlass zur Diskussion; so widerspricht eine solche Deutung prinzipiell gerade der Kollektivgutannahme, welcher der Public Good Index unterliegt.

Wie bereits im vorherigen Abschnitt erörtert, wird hier bei der Kollektivgutannahme fest davon ausgegangen, dass es eben keinerlei Interesse an einer späteren Aufteilung des Gemeinguts in Privatgüter für jedes Koalitionsmitglied gibt.

Je nach den genauen Umständen und Art der Modellierung des Spiels kann aber auch ein solches Vorgehen seine Rechtmäßigkeit haben, wird beispielsweise die Lösung als Erwartungswert und nicht als Aufteilung interpretiert.

Im Weiteren wird ein Beispiel für die Verwendung des Public Good Value vorgestellt.

4.1.2.3 Beispiel für Public Good Wert

Für das Public Good Wert Beispiel dient das bereits im Zusammenhang des Shapley Value erörterten Abstimmungsspiel (siehe Abschnitt 2.1.11) mit den jeweiligen Gewichten der Spieler $w = (35{,}21{,}15{,}15)$ sowie der zu erfüllenden Quote von $q = 51$.

Das Spiel und dessen Struktur lässt sich in folgender Form (siehe Tabelle 5) veranschaulichen. Neben dem jeweiligen Koalitions-Index werden zusätzlich die Spielermenge von S als Bit-Vektor sowie als Menge dargestellt. Die rechte Seite zeigt den von der Koalition S erzielten Wert $v(S)$ auf.

Tabelle 5: Spielstruktur für Public Good Value Beispiel mit Abstimmungsspiel $w = (35{,}21{,}15{,}15)$ und $q = 51$

Index	**Bit-Zeile für S**	**Spieler $\in S$**	$v(S)$
1	1 0 0 0	{1}	0
2	0 1 0 0	{2}	0
3	0 0 1 0	{3}	0

4	0 0 0 1	{4}	0
5	1 1 0 0	{1,2}	1
6	1 0 1 0	{1,3}	0
7	1 0 0 1	{1,4}	0
8	0 1 1 0	{2,3}	0
9	0 1 0 1	{2,4}	0
10	0 0 1 1	{3,4}	0
11	1 1 1 0	{1,2,3}	1
12	1 1 0 1	{1,2,4}	1
13	1 0 1 1	{1,3,4}	1
14	0 1 1 1	{2,3,4}	1
15	1 1 1 1	{1,2,3,4}	1

Bei allen Einträgen von Tabelle 5 mit $v(S) = 1$ handelt es sich um Gewinnkoalitionen. Diese werden entsprechend in Tabelle 6 aufgeführt.

Tabelle 6: Gewinnkoalitionen für Public Good Value Beispiel mit Abstimmungsspiel w=(35,21,15,15) und q=51

Index	**Bit-Zeile für S**	**Spieler $\in S$**	$v(S)$
5	1 1 0 0	{1,2}	1
11	1 1 1 0	{1,2,3}	1
12	1 1 0 1	{1,2,4}	1
13	1 0 1 1	{1,3,4}	1
14	0 1 1 1	{2,3,4}	1
15	1 1 1 1	{1,2,3,4}	1

Nach Eliminierung der Gewinnkoalitionen aus Tabelle 6, bei denen nicht jeder Spieler ein kritischer Spieler ist, ergibt sich Tabelle 7 mit den Minimumgewinnkoalitionen.

Exemplarisch soll an den Einträgen mit Index 5 und 12 aus Tabelle 6 vorgeführt werden, wieso es sich im ersten Fall um eine Minimumgewinnkoalition (siehe Formel 34) handelt, im zweiten Fall aber nicht.

Für das Abstimmungsspiel muss eine Quote von 51 erreicht werden. Wird hier nun die Koalition mit Index 5 betrachtet, setzt diese sich aus den Spielern 1 und 2 zusammen. Diese beiden Spieler weisen die Stimmgewichte 35 und 21 auf, können also gemeinsam mit 56 Stimmgewichten die Quote von 51 erfüllen. Sollte aber einer der beiden Spieler abspringen, kann die Quote nicht mehr erfüllt werden und so wird aus der Gewinn- eine Verlustkoalition. Bei beiden handelt es sich um kritische Spieler, die Voraussetzungen für eine Minimumgewinnkoalition sind somit für die Spielervereinigung {1,2} erfüllt.

Anders gelagert ist der Fall bei der Koalition mit Index 12 und deren Spielern j ∈ {1,2,4}. Die Gewichte der Spieler sind hier (35,21,15). Ohne weiteres kann daher mit einem gemeinsamen Stimmgewicht von 71 die Quote übererfüllt werden.

Spieler 1 nimmt dabei die Position eines kritischen Spielers ein, während Spieler 2 und 3 mit 36 die Quote nicht annähernd erreichen.

Auch Spieler 2 ist kritisch, die Spieler 1 und 2 verfehlen nämlich ohne ihn – knapp um ein Stimmgewicht – die erforderliche Quote.

Bei Spieler 3 hingegen handelt es sich keineswegs um einen kritischen Spieler. Durch seinen Wegfall ergibt sich die zur Koalition mit Index 5 identische Spielermenge, für die bereits gezeigt wurde, dass für diese die Quote kein Problem darstellt.

Hiermit handelt es sich in diesem Fall um keine Minimumgewinnkoalition.

Analog zum Vorgehen für die Koalitionen 5 und 12 wird für die anderen Gewinnkoalitionen verfahren. Hieraus erschließt sich Tabelle 7 mit allen für das Spiel vorliegenden Minimumgewinnkoalitionen.

Tabelle 7: Minimumgewinnkoalitionen M(v) für Public Good Value Beispiel mit Abstimmungsspiel $w = (35,21,15,15)$ *und* $q = 51$

Index	**Bit-Zeile für S**	**Spieler ∈ S**	$v(S)$
5	1 1 0 0	{1,2}	1
13	1 0 1 1	{1,3,4}	1
14	0 1 1 1	{2,3,4}	1

Aus den Minimumgewinnkoalitionen berechnet sich der Public Good Index anhand der Formel 35 sowie Formel 36 nun wie folgt:

Im ersten Schritt wird mit Formel 35 die Summe aller Koalitionswerte v(K) mit K ∈ M(v) und i ∈ K für alle Spieler aus N berechnet.

Formel 39: Summe aller Koalitionswerte der Minimumgewinnkoalitionen für das Public Good Value Beispiel

Für Spieler 1:

$$c_1(v) = \sum_{K \ni 1; K \in M(v)} v(K) = v(\{1,2\}) + v(\{1,3,4\}) = 1 + 1 = 2$$

Für Spieler 2:

$$c_2(v) = \sum_{K \ni 2; K \in M(v)} v(K) = v(\{1,2\}) + v(\{2,3,4\}) = 1 + 1 = 2$$

Für Spieler 3:

$$c_3(v) = \sum_{K \ni 3; K \in M(v)} v(K) = v(\{1,3,4\}) + v(\{2,3,4\}) = 1 + 1 = 2$$

Für Spieler 4:

$$c_4(v) = \sum_{K \ni 4; K \in M(v)} v(K) = v(\{1,3,4\}) + v(\{2,3,4\}) = 1 + 1 = 2$$

Im Anschluss kann mit Formel 36 der Public Good Index für alle Spieler bestimmt werden.

Formel 40: Public Good Indizes aller Spieler für das Public Good Value Beispiel

Für Spieler 1:

$$PGI_1(v) = \frac{c_1(v)}{\sum_{j \in N} c_j(v)} = \frac{c_1(v)}{c_1(v) + c_2(v) + c_3(v) + c_4(v)} = \frac{2}{8} = \frac{1}{4}$$

Für Spieler 2:

$$PGI_2(v) = \frac{c_2(v)}{\sum_{j \in N} c_j(v)} = \frac{c_2(v)}{c_1(v) + c_2(v) + c_3(v) + c_4(v)} = \frac{2}{8} = \frac{1}{4}$$

Für Spieler 3:

$$PGI_3(v) = \frac{c_3(v)}{\sum_{j \in N} c_j(v)} = \frac{c_3(v)}{c_1(v) + c_2(v) + c_3(v) + c_4(v)} = \frac{2}{8} = \frac{1}{4}$$

Für Spieler 4:

$$PGI_4(v) = \frac{c_4(v)}{\sum_{j \in N} c_j(v)} = \frac{c_4(v)}{c_1(v) + c_2(v) + c_3(v) + c_4(v)} = \frac{2}{8} = \frac{1}{4}$$

$$\Longrightarrow PGI_1(v) = PGI_2(v) = PGI_3(v) = PGI_4(v) = \frac{1}{4}$$

Hier fällt auf, dass die Machtanteile – trotz unterschiedlich hoher Gewichte – auf alle Spieler mit $\frac{1}{4}$ gleichmäßig verteilt sind.

Formel 38 mit $PGV_i(v) = PGI_i(v) * v(N)$ beschreibt hierbei, wie aus den bestimmten Machtindizes ein punktwertiger Lösungswert gewonnen werden kann.

In diesem Beispiel entsprechen aber der Public Good Value und Public Good Index einander, da der Skalarwert sich in diesem Fall mit $v(N) = 1$ bei der Multiplikation mit $PGI_i(v)$ neutral verhält.

```
> A<-generateGameVector(cFuncQuota,n=4,w=c(35,21,15,15),q=51)      #1
> publicGoodValue(A)                                              #2
 [1] 0.25 0.25 0.25 0.25                                          #3
```

Codebsp. 4 Public Good Value

4.1.3 Public Help Wert

Beim Public Help Wert (PHV) handelt es sich um ein eigenes, anhand des Public Help Index (PHI) definiertes, punktwertiges Lösungskonzept.

Hierbei ähneln sich sowohl das im Vorfeld behandelte Konzept des Public Good und das des Public Help Index sehr stark und weisen dabei zahlreiche unverkennbare Gemeinsamkeiten, wie beispielsweise die Kollektivgutannahme, auf. Im Grunde handelt es sich beim Public Help Index um eine Modifizierung des Public Good Index, bei der sämtliche Gewinnkoalitionen miteinfließen und nicht nur die Minimumgewinnkoalitionen (BERTINI et al. 2008: 83). Die Ausweitung auf nicht nur ausschließlich Minimumgewinnkoalitionen kann dabei durch verschiedene Beweggründe motiviert sein, ein besonders plausibler sind aber zu schwache Minimumgewinnkoalitionen (BERTINI et al. 2008: 83).

4.1.3.1 Festlegung des Public Help Index

Bevor der PHI zum Wertkonzept umdisponiert wird, wird in diesem Abschnitt erläutert wie er sich zusammensetzt.

Bei der Ermittlung der Machtindizes berücksichtigt das Konzept des Public Help Index alle Gewinnkoalitionen eines Spiels. Der hierfür benötigte Begriff der Gewinnkoalition wurde bereits bei der Spezifizierung des Public Good Index in Abschnitt 4.1.2.1 erörtert.

Der Public Help Index ist nach BERTINI et al. (2008: 83) und wie in Formel 41 definiert.

Formel 41: Definition Public Help Index

$$\mathrm{PHI_i(v)} = \frac{\overline{W_i}}{\sum_{j \in N} \overline{W_j}} \; \forall i \in N \wedge \; \overline{W_i} = \mathrm{card}(W_i)$$

Der Public Help Index vernachlässigt im Gegensatz zum Public Good Index sämtliche Koalitionswerte. Entscheidend ist bei diesem Konzept grundsätzlich nur die Anzahl der Gewinnkoalitionen.

Für jeden Spieler ergibt sich dessen Machtindex durch das Verhältnis von Gewinnkoalitionen, an denen dieser beteiligt ist, zu der gesamten Anzahl von Gewinnkoalitionen.

4.1.3.2 Der Public Help Index als Grundlage für ein Wertkonzept

Analog zum Public Good Value (siehe Abschnitt 4.1.2.2) wird auch beim Public Help Value verfahren und für das neue Lösungskonzept kollektive Rationalität gefordert.

Hiermit ergibt sich die in Formel 42 spezifizierte Auffassung des Public Help Value PHV in Abhängigkeit vom Public Help Index.

Formel 42: Definition Public Help Value

$$PHV_i(v) = PHI_i(v) * v(N)$$

Auf mögliche Kritikpunkte eines solchen Vorgehens wird an dieser Stelle nicht erneut eingegangen, da diese bereits im Zusammenhang mit dem Public Good Value in Abschnitt 4.1.2.2 entsprechend diskutiert wurden.

Im Weiteren wird ein Beispiel für die Verwendung des Public Help Value vorgestellt.

4.1.3.3 Beispiel für Public Help Wert

Für das Public Help Wert Beispiel dient erneut das bereits im Zusammenhang mit dem Shapley und Public Good Value erörterte Abstimmungsspiel (siehe Abschnitt 4.1.1.2), sowie den jeweiligen Gewichten der Spieler $w = (35,21,15,15)$ und der zu erfüllenden Quote von $q = 51$.

Die bereits bekannten Gewinnkoalitionen sind erneut in Tabelle 8 aufgeführt.

Tabelle 8: Gewinnkoalitionen für Public Good Value Beispiel mit Abstimmungsspiel w=(35,21,15,15) und q=51

Index	**Bit-Zeile für S**	**Spieler $\in S$**	$v(S)$
5	1 1 0 0	{1,2}	1
11	1 1 1 0	{1,2,3}	1
12	1 1 0 1	{1,2,4}	1
13	1 0 1 1	{1,3,4}	1
14	0 1 1 1	{2,3,4}	1
15	1 1 1 1	{1,2,3,4}	1

Auf Grundlage dieser Gewinnkoalitionen wird für die Spieler $i \in N$ jeweils deren Public Help Index berechnet.

Hierfür ist es notwendig, im Vorfeld für jeden Spieler i die Anzahl der Gewinnkoalitionen $\overline{W_i}$ zu bestimmen, an denen dieser beteiligt ist.

Formel 43 Anzahl der Gewinnkoalitionen mit Beteiligung des Spielers $i \in N$

Für Spieler 1:

$$\overline{W_1} = \text{card}(\{1,2\},\{1,2,3\},\{1,2,4\},\{1,3,4\},\{1,2,3,4\}) = 5$$

Für Spieler 2:

$$\overline{W_2} = \text{card}(\{1,2\},\{1,2,3\},\{1,2,4\},\{2,3,4\},\{1,2,3,4\}) = 5$$

Für Spieler 3:

$$\overline{W_3} = \text{card}(\{1,2,3\},\{1,3,4\}\{2,3,4\},\{1,2,3,4\}) = 4$$

Für Spieler 4:

$$\overline{W_4} = \text{card}(\{1,2,4\},\{1,3,4\},\{2,3,4\},\{1,2,3,4\}) = 4$$

Im Anschluss kann auf Grundlage der bestimmten Kennzahlen (siehe Formel 43) mit der Definition aus Formel 42 der Public Help Index für alle Spieler bestimmt werden.

Formel 44: Public Help Indizes für das Public Help Beispiel

Für Spieler 1:

$$PHI_1(v) = \frac{\overline{W_1}}{\sum_{j\in N}\overline{W_j}} = \frac{\overline{W_1}}{\overline{W_1}+\overline{W_2}+\overline{W_3}+\overline{W_4}} = \frac{5}{5+5+4+4} = \frac{5}{18}$$

Für Spieler 2:

$$PHI_2(v) = \frac{\overline{W_1}}{\sum_{j\in N}\overline{W_j}} = \frac{\overline{W_1}}{\overline{W_1}+\overline{W_2}+\overline{W_3}+\overline{W_4}} = \frac{5}{5+5+4+4} = \frac{5}{18}$$

Für Spieler 3:

$$PHI_3(v) = \frac{\overline{W_1}}{\sum_{j\in N}\overline{W_j}} = \frac{\overline{W_1}}{\overline{W_1}+\overline{W_2}+\overline{W_3}+\overline{W_4}} = \frac{4}{5+5+4+4} = \frac{4}{18}$$

Für Spieler 4:

$$PHI_4(v) = \frac{\overline{W_1}}{\sum_{j\in N}\overline{W_j}} = \frac{\overline{W_1}}{\overline{W_1}+\overline{W_2}+\overline{W_3}+\overline{W_4}} = \frac{4}{5+5+4+4} = \frac{4}{18}$$

Hier fällt im Vergleich zum Public Good Value auf, dass sich die unterschiedlich hohen Stimmgewichte etwas schwächer auf die Machtverteilung auswirken. Spieler 1 und 2 bekommen mit $\frac{5}{8}$ gleich hohe Machtanteile trotz unterschiedlich hoher Stimmgewichte.

Formel 42 mit $PHV_i(v) = PHI_i(v) * v(N)$ beschreibt auch hier, wie aus dem Public Help Machtindex ein punktwertiger Lösungswert gewonnen werden kann.

In diesem Beispiel entsprechen aber Public Good Value und Public Good Index einander, da der Skalarwert mit $v(N) = 1$ sich bei der Multiplikation mit $PGI_i(v)$ neutral verhält.

```
> A<-generateGameVector(cFuncQuota,n=4,w=c(35,21,15,15),q=51)        #1
> publicHelpValue(A)                                                #2
[1] 0.2777778 0.2777778 0.2222222 0.2222222                         #3
```

Codebsp. 5: Public Help Value Beispiel

4.1.4 τ-Wert

Beim τ-Wert handelt es sich um ein von TIJS (1981) vorgestelltes punktwertiges Lösungskonzept, das für quasi-balancierte Spiele eine garantiert eindeutige Lösung bietet.

Trotz der Beschränkung auf nur quasi-balancierte Spiele erfüllt der τ-Wert dennoch durchaus die notwendigen Voraussetzungen für die praktische Akzeptanz eines Lösungskonzepts. So überzeugt er durch leichte Kommunizierbarkeit und ist anders als das Nucleolus sowie das Shapley Konzept rein gedanklich sehr leicht nachzuvollziehen (ZELEWSKI 2009: 56).

FROMEN (2004: 218f) beschriebt hierbei die Grundidee, zwei Schranken als untere bzw. obere Grenze für eine mögliche Aufteilung des Spielwerts aufzustellen. Unter Berücksichtigung der beiden Schranken und weiterer geltender Restriktionen erfolgt hier die Bestimmung einer eindeutigen Imputation.

Im Weiteren wird auf die Eigenschaft der Quasi-Balanciertheit, die Schranken und die weiteren für den τ-Wert geltenden Einschränkungen eingegangen.

4.1.4.1 Die obere und untere Schranke für den τ-Wert

Bei der oberen Schranke handelt es sich hierbei um den sogenannten Utopiapayoff-Vektor (siehe Formel 45), der den marginalen Beitrag aller Spieler für die große Koalition N beschreibt und die bestmögliche Aufteilung bei der für den τ-Wert geforderten Quasi-Balanciertheit (siehe Abschnitt 4.1.4.3) repräsentiert.

Formel 45: Definition des Utopiapayoffs

$$M_i(N, v) = v(N) - v(N \backslash \{i\})$$

Der Vektor für die minimalen Rechte der Spieler (siehe Formel 46) stellt gleichzeitg die untere Schranke dar. Hier werden für jeden Spieler i alle Koalitionen S betrachtet in denen der Spieler i involviert ist. Für alle Koalitionen S wird festgestellt, welchen Wert Spieler i maximal erhalten kann – im Fall, dass alle anderen an der Koalition beteiligten Spieler j sich ihren maximalen Betrag (siehe Formel 45) sichern. Dieser maximaler Wert, den sich Spieler i unter den beschriebenen Umständen sichern kann, repräsentiert die minimalen Rechte des Spielers i (BRANZEI et. al. 2008: 20).

Formel 46: Definition der Minimalen Rechte (minimal rights) eines Spielers i

$$m_i(N,v) = \max_{S:i\in S} \left\{ v(S) - \sum_{j\in S\setminus\{i\}} M_j \right\}$$

4.1.4.2 Quasi-Balanciertheit

Der in diesem Abschnitt vorgestellte Begriff der Quasi-Balanciertheit wurde von TIJS (1981: 127) zur Beschreibung der Spieleklasse entwickelt, für die durch das τ-Wert Konzept eine eindeutige Lösung ermittelt werden kann (JENE 2015: 226).

Die hier verwendete Notation der Quasi-Balanciertheit (siehe Formel 47) richtet sich hierbei nach BRANZEI et. al. (2008: 31).

Formel 47: Definition der Quasi Balanciertheit

1. $\mathrm{m_i}(N,v) \leq M_i(N,v)\ \forall i \epsilon N$
2. $\sum_{i\in N} m_i(N,v) \leq v(N) \leq \sum_{i\in N} M_i(N,v)$

Die minimalen Rechte aller Spieler müssen demnach immer kleiner gleich deren Utopiapayoff-Werte sein (siehe Bedingung 1 der Formel 47) und der Wert der großen Koalition muss sich zwischen der Summe der Werte für die minimalen Rechte und der für die Utopiapayoffs bewegen (siehe Bedingung 2 der Formel 47). Auf Spiele, welche diese Eigenschaft nicht erfüllen, ist das τ-Wert Konzept nicht anwendbar (JENE 2015: 226).

Das Konzept lässt sich hierbei immer auf konvexe Spiele anwenden, da diese über einen nichtleeren Kern verfügen. Somit sind konvexe Spiele sowohl balanciert als auch quasi-balanciert (DRIESSEN, TIJS 1985: 231).

Es existieren aber andererseits auch wesentliche Spiele, die nicht quasi-balanciert sind (DRIESSEN, TIJS 1983: 253)

4.1.4.3 Der $\tau - Wert$

In einem quasi-balancierten Spiel mit den minimalen Rechten m und dem Utopiapayoff-Vektor M für $\alpha \in [0,1]$, ist der τ-Wert nach BRANZEI et al. (2008: 32) wie in Formel 48 definiert.

Formel 48: Definition τ-Wert für $\alpha \in [0,1]$

$$\tau(v) = \alpha m(N,v) + (1-\alpha)M(N,v)$$
$$Unter\ der\ Nebenbedingung:$$
$$\sum_{i\in N} \tau_i = v(N)$$

Der τ-Wert bewegt sich hierbei auf der Strecke zwischen den beiden Punkten $m(N,v)$ und $M(N,v)$.

Dies wird direkt nach Umformung der ersten Bedingung aus Formel 48 ersichtlich.

Formel 49: Umformung der Definition des τ-Wert in Parameterform

$$(1)\ \tau(v) = \alpha m(v) + (1-\alpha)M(N,v)$$

$$(2)\ \tau(v) = \alpha m(v) + M(N,v) - \alpha M(N,v)$$

$$\Rightarrow Parameterform\colon \tau(v) = M(N,v) + \alpha(m(v) - M(N,v))$$

Dies ergibt sich aus $\alpha \in [0,1]$und der 1. Bedingung von Formel 48, die an sich eine Geradengleichung in Parameterform mit $M(N,v)$ als Aufpunkt und $(m(v) - M(N,v))$ als Richtungsvektor widerspiegelt.

Auf dieser Strecke erfüllt genau ein Punkt die nach der 2. Bedingung von Formel 48 geforderte kollektive Rationalität. Somit gibt es in einem quasi-balancierten Spiel genau einen Punkt als Lösungswert, bei dem es sich zusätzlich unter den geforderten Kriterien um eine Imputation handelt.

4.1.4.4 Ein Beispiel für den τ-Wert

In folgendem Beispiel soll in einem Dreipersonenspiel mit $N = \{1,2,3\}$ und der Koalitionsfunktion v, die wie in Formel 50 definiert ist, der zugehörige τ-Wert berechnet werden.

Formel 50: Spielvektors für t-Wert Beispiel

Werte der Einerkoalitonen:

$$v(1) = v(2) = 0$$
$$v(3) = 5$$

Werte der Zweierkoalitionen:

$$v(1,2) = 4;\ v(1,3) = 8;\ v(2,3) = 9$$

Wert der Großen Koalition:

$$v(123) = 12$$

Das Spiel und dessen Struktur lässt sich in folgender Form (siehe Tabelle 9) veranschaulichen. Neben dem jeweiligen Koalitions-Index, werden zusätzlich die Spielermenge von S als Bit-Vektor sowie als Menge dargestellt. Die rechte Seite zeigt den von der Koalition S erzielten Wert $v(S)$ auf.

Tabelle 9: Spielstruktur für t-Wert Beispiel

Index	**Bit-Zeile für S**	**Spieler $\in S$**	$v(S)$
1	1 0 0	{1}	0
2	0 1 0	{2}	0
3	0 0 1	{3}	5
4	1 1 0	{1,2}	4
5	1 0 1	{1,3}	8
6	0 1 1	{2,3}	9
7	1 1 1	{1,2,3}	12

Bevor der τ-Wert berechnet werden kann, muss das Spiel erst auf Quasi-Balanciertheit (siehe Formel 47) geprüft werden. Hierzu müssen die Schranken für den minimalen (siehe Formel 46) und maximalen (siehe Formel 45) Anteil am Effizienzgewinn aller Spieler ermittelt werden.

Die Tabelle 10 zeigt hier mit $M(N,v) = (3,4,8)$ den maximalen Anteil aller Spieler am Effizienzgewinn in Form des sogenannten Utopiapayoffs $M(N,v)$ und die Art und Weise dessen Berechnung.

Tabelle 10: Maximaler zurechenbarer Anteil M_i am Effizienzgewinn für alle Spieler im t-Wert Beispiel

Spieler i	$\boldsymbol{v(N) - v(N\backslash\{i\})}$	$\boldsymbol{M_i(N,v)}$
1	$v(N) - v(N\backslash\{1\}) = v(\{1,2,3\}) - v(\{2,3\}) = 12 - 9 = 3$	3
2	$v(N) - v(N\backslash\{2\}) = v(\{1,2,3\}) - v(\{1,3\}) = 12 - 8 = 4$	4
3	$v(N) - v(N\backslash\{2\}) = v(\{1,2,3\}) - v(\{1,2\}) = 12 - 4 = 8$	8

Nach dem maximalen, gilt es auch den minimalen Anteil (siehe Formel 46) zu ermitteln. Dies geschieht in der Tabelle 11.

Tabelle 11: Minimal zurechenbarer Anteil m_i am Effizienzgewinn für alle Spieler im t-Wert Beispiel

Spieler i	$\boldsymbol{v(S) - \sum_{j \in S\backslash\{i\}} M_j}$	$\boldsymbol{m_i(N,v)}$
1	$v(\{1\}) - 0 = 0 - 0 = 0$ $v(\{1,2\}) - (M_2) = 4 - 4 = 0$ $v(\{1,3\}) - (M_3) = 8 - 8 = 0$ $v(\{1,2,3\}) - (M_2 + M_3) = 12 - 4 - 8 = 0$	0
2	$v(\{2\}) - 0 = 0 - 0 = 0$	1

	$v(\{1,2\}) - (M_1) = 4 - 3 = 1$ $v(\{2,3\}) - (M_3) = 9 - 8 = 1$ $v(\{1,2,3\}) - (M_1 + M_3) = 12 - 3 - 8 = 1$	
3	$v(\{3\}) - 0 = 5 - 0 = 5$ $v(\{1,3\}) - (M_1) = 8 - 3 = 5$ $v(\{2,3\}) - (M_2) = 9 - 4 = 5$ $v(\{1,2,3\}) - (M_1 + M_2) = 12 - 3 - 4 = 5$	5

Eine Aufstellung unterer und oberer Schranken aller Spieler sowie deren jeweiliger Summenwert zeigt entsprechend Tabelle 12.

Tabelle 12: Übersicht minimaler und maximaler Anteile am Effizienzgewinn im t-Wert Beispiel

Spieler i / **Schranke**	**1**	**2**	**3**	**Summe**
$\boldsymbol{M_i(N,v)}$	3	4	8	15
$\boldsymbol{m_i(N,v)}$	0	1	5	6

Im Anschluss wird anhand der Tabelle 12 überprüft, ob die Anforderungen an die unteren und oberen Schranken gemäß Formel 47 erfüllt sind (siehe Formel 51).

Formel 51: Prüfung auf Quasi-Balanciertheit im t-Wert Beispiel

1. $\mathrm{m_i}(N,v) \leq M_i(N,v)\ \forall i \epsilon N$

$\Rightarrow erf\ddot{u}llt, da$:

$0 \leq 3 \ \wedge 1 \leq 4 \wedge 5 \leq 8$

2. $\sum_{i\in N} m_i(N,v) \leq v(N) \leq \sum_{i\in N} M_i(N,v)$

$\Rightarrow erf\ddot{u}llt, da$:

$6 \leq 12 \leq 15$

Da die beiden Kriterien an Quasi-Balanciertheit im Fall des spezifizierten Spiels erfüllt sind, kann mit der Berechnung des τ-Werts fortgefahren werden.

Die Berechnung des τ-Werts erfolgt nach Formel 48. Hier wird der Definition des τ-Werts in Parameterform (siehe Formel 49) mit $\tau(v) = M(N,v) + \alpha(m(v) - M(N,v))$ für $\sum_{i\in N} \tau_i = v(N)$ das entsprechende lineare Gleichungssystem aufgestellt.

Formel 52 Bestimmung des α-Werts im t-Wert Beispiel

$Schritt$ 0: Allgemeines Lineares Gleichungssystem
für $\tau -$ Wert im Dreispielerfall:

$$
\begin{array}{llllll}
(1) & \tau_1 & & + & (M_1 - m_1)\alpha & = & M_1 \\
(2) & & \tau_2 & + & (M_2 - m_2)\alpha & = & M_2 \\
(3) & & & \tau_3 + & (M_3 - m_3)\alpha & = & M_3 \\
(4) & \tau_1 & +\tau_2 \quad +\tau_3 & & & = & v(N)
\end{array}
$$

Schritt 1: Lineares Gleichungssystem
mit eingesetzten unteren und oberen Schranken:

$$
\begin{array}{llllll}
(1) & \tau_1 & & + & 3\alpha & = & 3 \\
(2) & & \tau_2 & + & 3\alpha & = & 4 \\
(3) & & & \tau_3 + & 3\alpha & = & 8 \\
(4) & \tau_1 & +\tau_2 \quad +\tau_3 & & & = & 12
\end{array}
$$

Schritt 2: Auflösen des Lineares Gleichungssystem
nach τ:

$$
\begin{array}{llllll}
(1') & \tau_1 & & + & = & 3 - 3\alpha \\
(2') & & \tau_2 & + & = & 4 - 3\alpha \\
(3') & & & \tau_3 + & = & 8 - 3\alpha \\
(4) & \tau_1 & +\tau_2 \quad +\tau_3 & & = & 12
\end{array}
$$

Schritt 3: *Einsetzen von* $(1'), (2')$ *und* $(3')$ *in* (4):

$$
\begin{array}{llll}
(3 - 3\alpha) & +(4 - 3\alpha) & +(8 - 3\alpha) & = \quad 12 \\
15 & -9\alpha & & = \quad 12 \\
\Rightarrow & \alpha & & = \quad 1/3
\end{array}
$$

Nach dem Auflösen des Gleichungssystems nach τ (siehe Schritt 2) und dem Einsetzen der τ-Komponentenwerte in die 4. Gleichung zur kollektiven Rationalität, ergibt sich für α ein Wert von $1/3$.

Für die τ-Komponenten (siehe Formel 53) resultieren mit $\alpha = 1/3$ daher die Werte $(2, 3, 7)$.

Formel 53: Bestimmung des τ-Werts im Beispiel

Einsetzen $\alpha = 1/3$:

$$
\begin{array}{llllll}
(1') & \tau_1 & & + & = & 3 - 3(1/3) \\
(2') & & \tau_2 & + & = & 4 - 3(1/3) \\
(3') & & & \tau_3 + & = & 8 - 3(1/3)
\end{array}
$$

Resultierende $\tau -$ *Komponenten*:

$$
\begin{array}{llllll}
(1') & \tau_1 & & + & = & 2 \\
(2') & & \tau_2 & + & = & 3 \\
(3') & & & \tau_3 + & = & 7
\end{array}
$$

Das gleiche Ergebnis für den τ-Werts im Beispiel mit dem Spielvektor A = (0,0,5,4,8,9,12) kann auch mittels der Methode `tauValue` in `CoopGame` berechnet werden und wird in Codebsp. 6 kurz veranschaulicht.

```
>A<-c(0,0,0,5,4,8,9,12)                                    #1
> tauValue(A)                                              #2
2 3 7                                                      #3
```

Codebsp. 6: Berechnung des τ-Werts in `CoopGame`

Im Folgenden wird die Implementierung zum τ-Wert innerhalb des Pakets `CoopGame` in den Grundzügen skizziert.

4.1.4.5 Umsetzung des τ-Werts in R

Bei der Implementierung des τ-Wertes (siehe Codebsp. 7) wird zuerst, wie auch im vorangegangenen Beispiel (siehe Abschnitt 4.1.4.4), das Spiel in Zeile 5 auf Quasi-Balanciertheit (siehe Formel 47) geprüft.

```
tauValue<-function(A){                                                          #1
        paramCheckResult=getEmptyParamCheckResult()                             #2
        initialParamCheck_tauValue(paramCheckResult,A)                          #3
        retVal=NULL                                                             #4
        if(!isQuasiBalanced(A)){                                                #5
                print("Game is not quasi balanced therefore no tau value can be retrieved") #6
        }else{                                                                  #7
                n=getNumberOfPlayers(A)                                         #8
                N=length(A)                                                     #9
                mc=utopiaPayoff(A=A)                                            #10
                diffM_m=mc["M",]-mc["m",]                                       #11
                tDiagMatrix=diag(1,nrow=n,ncol=n)                               #12
                coeffMat=cbind(tDiagMatrix,-diffM_m)                            #13
                coeffMat=rbind(coeffMat,c(rep(1,n),0))                          #14
                retVal=solve(coeffMat,c(mc["m",],A[N]))                         #15
                unname(retVal)                                                  #16
        }                                                                       #17
        return(retVal[1:n])                                                     #18
}                                                                               #19
```

Codebsp. 7: Funktion zur Berechnung des τ-Werts

Ist das Spiel quasi-balanciert, wird das aus der Formel 48 abgeleitete Gleichungssystem (siehe Formel 54) mit der internen R Funktionalität `solve` gelöst (siehe Zeile 15) und der ermittelte τ-Wert anschließend zurückgegeben (siehe Zeile 19).

Formel 54: Zur Bestimmung des τ-Werts verwendete Gleichungssystems

$$\begin{array}{ccccccccccc}
t_1 & & & & & & + & (M_1 - m_1) & = & M_1 \\
 & t_2 & & & & & + & (M_2 - m_2) & = & M_2 \\
 & & t_3 & & & & + & (M_3 - m_3) & = & M_3 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & & \vdots & \vdots & \vdots \\
 & & & & t_n & + & (M_n - m_n) & = & M_n \\
t_1 & +t_2 & +t_3 & +\cdots+ & t_n & & & = & v(N)
\end{array}$$

Die Berechnung der minimalen Rechte und des Utopiapayoffs (siehe Zeile 10) für alle Spieler $i \epsilon N$, sowie der Überprüfung auf Balanciertheit (siehe Zeile 5) ist hierbei in die beiden Funktionen `utopiaPayoff` und `isBalancedGame` ausgelagert.

4.1.5 Gately Punkt

Beim Gately Punkt handelt es sich um ein punktwertiges Lösungskonzept, das von GATELY (1974) vorgestellt worden ist und sowohl in die Monographie von ZELEWSKI (2009) als auch in die Lehrbücher von STRAFFIN (1993) sowie NARAHARI (2014) Eingang gefunden hat. Darüber hinaus wurde das Konzept nach Google Scholar in mindestens 183 weiteren Publikationen zitiert (Stand 24.06.2017). Die Zielsetzung dieses Lösungkonzept besteht darin, bei der Kooperation ein oder mehrerer Partner, eine Kosten- bzw. Gewinnaufteilung derart vorzunehmen, dass das Risiko für das Abspringen eines der Partner möglichst gering gehalten wird.

Seinen geschichtlichen Hintergrund hat der Gately Punkt in einer spieltheoretischen Betrachtung für die Investitionsplanung einer gemeinschaftlichen elektrischen Infrastruktur zwischen vier Regionen im südlichen Teils Indiens – Andhra Pradesh, Kerala, Mysore und Tamil Nadu (GATELY 1974: 195), wobei die beiden Regionen Kerale und Mysore vereinfacht als ein Gebiet betrachtet wurden (GATELY 1974: 196).

Auf Grundlage einer für alle akzeptablen Kostenaufteilung, sollte – entgegen dem bisherigen Trend von nur regional vorangetriebenen und nichtkooperativen Investitionen – hier für alle das gegenseitige Interesse geschaffen werden, miteinander zu kooperieren.

Als Voraussetzung für ein derartiges Bestreben war es im Vorfeld notwendig ein hinreichendes Messkriterium für die Störanfälligkeit – *Propensity To Disrupt* – zu schaffen, die für jede Region sowohl das Verhältnis der eigenen zu den Gesamtkosten aller berücksichtigt (GATELY 1974: 200).

Für das Lösungskonzept des Gately Punkts wird jedoch in der vorliegenden Masterarbeit die von GATELY (1974) implizierte Eindeutigkeit widerlegt und darauf mit einer eigenen Abwandlung, dem später vorgestellten Lexical Gately Punkt reagiert (siehe Abschnitt 4.2.7).

Im Weiteren folgt eine Vorstellung des für den Gately Punkt definierten Maßes für Störanfälligkeit und ein veranschaulichendes sowie einfaches Beispiel in einem eindeutigen Fall.

Nachdem beim Leser hierdurch ein besserer Zugang zum Konzept geschaffen wurde, thematisieren die darauffolgenden Abschnitte, wie die Nichteindeutigkeit zu begründen und sogar konstruierbar ist.

Die dort beschriebenen Überlegungen fließen dabei alle in die Implementierung mit ein, die zum Abschluss kurz angerissen wird.

4.1.5.1 Die Propensity To Disupt im Hinblick auf den Gately Punkt

Ein solches Maß für die Störanfälligkeit eines Spielers i lässt sich für ein Spiel (N, v) mit $N = \{1, \dots, n\}$, eine Imputation x und eine Koalitionsfunktion v wie in Formel 55 definieren.

Formel 55: Definition der Propensity To Disrupt

$$d_i(x) = \frac{\sum_{j \neq i} x_j - v(N \backslash \{i\})}{x_i - v(\{i\})}$$

Der Zähler mit $\sum_{j \neq i} x_j - v(N\{i\})$ bringt hierbei den marginalen Beitrag des Spielers i bezüglich der großen Koalition zum Ausdruck.

Ist der Betrag des Nenners 0 handelt es sich bei dem Spieler i hinsichtlich der großen Koalition N um einen Nullspieler (vgl. Definition Nullspieler WIESE 2005: 194), der keinen Beitrag zur großen Koalition leistet.

Bei Negativität des Zählers wirkt sich der Spieler i sogar negativ auf die anderen Spieler aus, da eine Kooperation mit diesem Spieler i den Gewinn aller anderen schmälert.

Im Zähler spiegelt sich auch hier bei Nichtnegativität der persönliche Nutzen für Spieler i wieder, an der großen Koalition teilzuhaben. Für den Wert von 0 verhält es sich neutral

Ein Wert echt kleiner 0 für $x_i - v(\{i\})$ ist indes nicht zulässig, da $x_i - v(\{i\}) < 0$ gegen das Kriterium der individuellen Rationalität verstößt, welches bei Imputationen gewahrt bleiben muss. Zudem ist für $x_i - v(\{i\}) = 0$ die *Propensity To Disrupt* aufgrund der Division durch 0 nicht definiert.

Aus diesem Grund gilt es im Weiteren die eigens definierte strikte individuelle Rationalität $x_i > v(\{i\})$ für das Gately Konzept zu fordern.

Formel 56: Definition der strikten individuellen Rationalität

$$x_i > v(\{i\}) f\ddot{u}r\ \forall\ i \in N$$

Je geringer die *Propensity To Disrupt* mit $d_i(x)$ für den Spieler i demnach ausfällt, umso größer ist dessen Verlangen, in der großen Koalition zu verbleiben.

Die Zielsetzung des Gately Punktes besteht daher darin, für ein möglichst großes Maß an Stabilität hinsichtlich des Wegfalls nur einzelner Spieler zu sorgen und die maximale *Propensity To Disrupt* für alle Einerkoalitionen zu minimieren.

Tatsächlich ergibt sich dieser Punkt für den Fall, dass das Maß für die Störanfälligkeit eines jeden Spielers identisch ist, der sogenannten *Equal Propensity To Disrupt*.

Formel 57: Definition der Equal Propensity To Disrupt

$$d_1(x) = \cdots = d_n(x) \qquad \textit{für } n=card(N)$$

Nach LITTLECHILD und VAIDYA (1976: 153) lässt sich der Wert für die *Equal Propensity To Disrupt* mit folgender Formel bestimmen, wobei der Parameter k hier die Koalitionsgröße beschreibt, für die das Maß bestimmt werden soll.

Formel 58: Definition der allgemeinen Formel zur Bestimmung der Equal Propensity To Disrupt

$$d^*(k,x) = \frac{\frac{(n-1)!}{k!(n-k-1)!}v(N) - \sum_{S:\,|S|=k} v(N-S)}{\frac{(n-1)!}{(k-1)!(n-k)!}v(N) - \sum_{S:\,|S|=k} v(S)}$$

Durch entsprechende Umformungen und unter Berücksichtigung einer Koalitionsgröße von $k = 1$, ergibt sich im Fall des Gately Punkts eine einfachere und umgänglichere Berechnungsformel.

Formel 59: Umformung der Formel zur Bestimmung der Equal Propensity To Disrupt für k=1

$$d^*(k=1,x) = \frac{\frac{(n-1)!}{1!(n-1-1)!}v(N) - \sum_{S:\,|S|=1} v(N-S)}{\frac{(n-1)!}{(1-1)!(n-1)!}v(N) - \sum_{S:\,|S|=1} v(S)} = \frac{(n-1)v(N) - \sum_{S:\,|S|=1} v(N-S)}{v(N) - \sum_{S:\,|S|=1} v(S)}$$

Diese Formel findet später bei der Bestimmung des Gately Punkts Verwendung.

4.1.5.2 Ein eindeutiges Gately Punkt Beispiel

Gegeben sei ein Dreipersonenspiel mit $N = \{1, 2, 3\}$ und der Koalitionsfunktion v, die wie folgt definiert ist:

Formel 60: Spielvektor für das Beispiel eines eindeutigen Gately Punkts

Werte der Einerkoalitonen:

$$v(1) = v(2) = v(3) = 0$$

Werte der Zweierkoalitionen:

$$v(1,2) = 0;\ v(1,3) = 6;\ v(2,3) = 4.5$$

Wert der Großen Koalition:

$$v(123) = 9$$

Wird als Imputation $y = (3, 1.5, 4.5)$ herangezogen, ergeben sich die Maße $d = (0.5, 1, 1)$.

Formel 61: Propensities To Disrupt der Spieler für das Beispie1 eines eindeutigen Gately Punkts

Für Spieler 1:

$$d_1(x) = \frac{\sum_{j\neq 1} y_j - v(N\backslash\{1\})}{y_1 - v(\{1\})} = \frac{(1.5 + 4.5) - 4.5}{3 - 0} = \frac{3}{6} = \frac{1}{2}$$

Für Spieler 2:

$$d_2(x) = \frac{\sum_{j\neq 2} y_j - v(N\backslash\{2\})}{y_2 - v(\{2\})} = \frac{(3 + 4.5) - 6}{1.5 - 0} = \frac{1.5}{1.5} = 1$$

Für Spieler 3:

$$d_3(x) = \frac{\sum_{j\neq 3} y_j - v(N\backslash\{3\})}{y_3 - v(\{3\})} = \frac{(3 + 1.5) - 0}{4.5 - 0} = \frac{4.5}{4.5} = 1$$

Mit der Imputation $y = (3, 1.5, 4.5)$ wird jedoch der Gately Punkt nicht erzielt, da die Propensities to Disrupt noch unterschiedlich und noch weiter zu minimieren sind.

Abhilfe schafft hier die Imputation $z = \left({}^{54}/_{22}, {}^{36}/_{22}, {}^{108}/_{22}\right)$, welche die Propensities To Disrupt auf $d_1(x) = d_2(x) = d_3(x) = \frac{5}{6}$ minimiert.

Auf welche Art und Weise der folgende Gately Punkt in Form der Imputation $z = \left({}^{54}/_{22}, {}^{36}/_{22}, {}^{108}/_{22}\right)$ bestimmt wird, soll an dieser Stelle noch nicht näher erläutert werden. Das hier angegebene Beispiel dient vorerst lediglich zur Motivation. Die benötigten Grundlagen werden später in dem Abschnitt 4.1.5.3 zur Herleitung der Nichteindeutigkeit des Gately Punkts gelegt, doch sei in diesem Zusammenhang vorab auf die bereits bekannte Formel 58 und die im Weiteren hergeleitete Formel 64 verwiesen. Anhand dieser beiden Formeln lässt sich die Imputation *z* als richtiger Lösungswert nachvollziehen.

Das unten aufgeführte Codebsp. 8 veranschaulicht abschließend das hier im Abschnitt 4.1.5.2 beschriebene Beispiel und die Einsatzmöglichkeiten der Funktionalitäten von `CoopGame`, für die Bestimmung und Analyse des Gately Punkts.

```
require("testthat")                                    #1
require("CoopGame")                                    #2
                                                       #3
#Define game v                                         #4
v=c(0,0,0,0,6,4.5,9)                                   #5
                                                       #6
#Define imputation y                                  #7
y=c(3,1.5,4.5)                                         #8
                                                       #9
#Output Propensities to Disrupt                        #10
#for imputation y                                      #11
propensityToDisrupt(v,x=y,S=c(1))                      #12
propensityToDisrupt(v,x=y,S=c(2))                      #13
propensityToDisrupt(v,x=y,S=c(3))                      #14
                                                       #15
#Define imputation z                                  #16
z=c(54/22, 36/22, 108/22)                              #17
                                                       #18
#Output Propensities to Disrupt                        #19
#for imputation z                                      #20
propensityToDisrupt(v,x=z,S=c(1))                      #21
propensityToDisrupt(v,x=z,S=c(2))                      #22
propensityToDisrupt(v,x=z,S=c(3))                      #23
                                                       #24
#Output Equal Propensity to Disrupt                    #25
#for game v                                            #26
equalPropensityToDisrupt(v,k=1)                        #27
                                                       #28
#Calculate Gately Point                                #29
lgv<-lexicalGatelyValue(v)                             #30
                                                       #31
#Compare Gateley Point                                 #32
#with imputation z                                     #33
expect_equal(lgv,z)                                    #34
```

Codebsp. 8: Veranschaulichungen zum Gately Punkt

Im Folgenden wird darauf eingegangen, unter welchen Umständen der Gately Punkt eben nicht ein eindeutiges Lösungskonzept ist.

4.1.5.3 Herleitung der Nichteindeutigkeit des Gately Punkts

In diesem Abschnitt wird die implizierte Eindeutigkeit des Gately Punkts widerlegt.

Die im Abschnitt 4.1.5.1 erörterte Formel für die *Propensity To Disrupt* dient als Ausgangspunkt für die Wiederlegung der implizierten Eindeutigkeit.

Formel 62: Definition der Propensity To Disrupt als Ausgangspunkt für die Widerlegung der Eindeutigkeit des Gately Punktes

$$d_i = \frac{\sum_{j \neq i} x_j - v(N \backslash \{i\})}{x_i - v(\{i\})}$$

Durch einfache Umformung ergibt sich hier das folgende Gleichungssystem (siehe Formel 63) für die Spieler $i \in N$ und deren jeweilige *Propensity To Disrupt* von d_i der Form $Ax = b$: $A \in \mathbb{R}^{n*n}, x \in \mathbb{R}^n, b \in \mathbb{R}^n$.

Formel 63: Umformung der Definition der Propensity To Disrupt für die Widerlegung der Eindeutigkeit des Gately Punktes

$$\sum_{j \neq i} x_j - d_i = v(N - \{i\}) - d_i v(\{i\})$$

Da im Fall des Gately Punkts die Propensities To Disrupt für alle Spieler sich in der *Equal Propensity To Disrupt* d^* wiederfinden, lässt sich das Gleichungssystem weiter in das unten aufgeführte System überführen. Die Propensities To Disrupt werden an dieser Stelle überall durch die *Equal Propensity To Disrupt* d^* ersetzt.

Formel 64: Gleichungssystem für die Widerlegung der Eindeutigkeit des Gately Punkts nach Einsetzen der Equal Propensity To Disrupt

$$\begin{array}{cccccccc}
-d^*x_1 & +x_2 & +x_3 & +\cdots+ & x_n & = & v(N-\{1\}) & -d^*v(\{1\}) \\
x_1 & -d^*x_2 & +x_3 & +\cdots+ & x_n & = & v(N-\{2\}) & -d^*v(\{2\}) \\
x_1 & +x_2 & -d^*x_3 & +\cdots+ & x_n & = & v(N-\{3\}) & -d^*v(\{3\}) \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
x_1 & +x_2 & +x_3 & +\cdots+ & -d^*x_n & = & v(N-\{n\}) & -d^*v(\{n\})
\end{array}$$

Die Nichteindeutigkeit bzw. Nichtexistenz einer Lösung ergibt sich aus dem Gleichungssystem, falls $d^* = -1$.

Ist doch das Gleichungssystem (siehe Formel 65), das sich in diesem Fall ergibt, eindeutig unterbestimmt, da die linke Seite mit $x_1 + x_2 + x_3 + \cdots + x_n$ für alle n Gleichungen identisch ist. Hier wird deutlich, dass gar keine punktwertige Lösung mehr existieren kann, weil in diesem Fall $rang(A) = 1 \ll n$ gilt und eine eindeutige Lösung $rang(A) = n$ erfordert.

Formel 65: Gleichungssystem für die Widerlegung der Eindeutigkeit des Gately Punkts nach Einsetzen der Equal Propensity To Disrupt von minus 1

Linke Seite						Rechte Seite	
x_1	$+x_2$	$+x_3$	$+\cdots+$	x_n	$=$	$v(N-\{1\})$	$+v(\{1\})$
x_1	$+x_2$	$+x_3$	$+\cdots+$	x_n	$=$	$v(N-\{2\})$	$+v(\{2\})$
x_1	$+x_2$	$+x_3$	$+\cdots+$	x_n	$=$	$v(N-\{3\})$	$+v(\{3\})$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
x_1	$+x_2$	$+x_3$	$+\cdots+$	x_n	$=$	$v(N-\{n\})$	$+v(\{n\})$

Je nachdem, wie die Werte der n Gleichungen auf der rechten Seite beschaffen sind, lassen sich für das Gleichungssystem mit $rang(A) = 1$ folgende zwei Unterscheidungen bezüglich der Beziehung zu $rang(A|b)$ festhalten.

Sollte für A mit $rang(A)$ hier $rang(A) < rang(A|b)$ gelten, existiert für das zugrundeliegende Spiel keinerlei Lösung, da für diesen Fall (siehe Formel 66) aufgrund der Widersprüche in den n Gleichungen keine Lösung bestimmbar ist.

Formel 66: Zulässiger rechter Spaltenvektor b für das Gleichungssystem A zur Widerlegung der Eindeutigkeit des Gately Punkts

$$rang(A) = 1 < rang(A|b)$$

$$\Rightarrow b \neq \beta * \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} : \forall \beta \in \mathbb{R}$$

Für den zweiten Fall (siehe Formel 67) existieren dagegen mit $rang(A|b) = rang(A)$ unendlich viele Lösungen.

Formel 67: Unzulässiger rechter Spaltenvektor b für Gleichungssystem A zur Widerlegung der Eindeutigkeit des Gately Punkts

$$rang(A) = rang(A|b) = 1$$

$$\Rightarrow b = \beta * \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} : \exists \beta \in \mathbb{R}$$

Aus dem spieltheoretischen Kontext heraus wird für die Betrachtung des Gately Konzepts jedoch nur der Fall $\beta = v(N)$ weiterverfolgt, da nur diesem Fall die für das Konzept benötigte kollektive Rationalität weiter gewahrt bleibt.

Für den zweiten Fall mit $\beta \neq v(N)$, existieren zwar auch unendlich viele Lösungen aber nicht im Sinne des Gately-Konzepts, weil hier entsprechend die kollektive Rationalität verletzt wird.

Formel 68: Fallbetrachtung für Koalitionswert der großen Koalition ungleich der Summer aller Werte der Einerkoalitionen und deren Komplemente

$$v(N - \{i\}) + v(\{i\}) \neq v(N) \quad \exists\, i \in N$$

Der Fall $\beta = v(N)$ tritt hierbei ein, wenn für sämtliche Einerkoalitionen mit den Spielern $i \in N$ der Koalitionswert addiert mit dem Koalitionswert der Komplementkoalition K den Wert der großen Koalition ergibt (siehe Formel 69).

Formel 69: Forderung nach dem Koalitionswert der großen Koalition für jeweils die Summe aller Werte der Einerkoalitionen und deren Komplemente

$$v(N - \{i\}) + v(\{i\}) = v(N) \quad \forall\, i \in N$$

Bei dieser Lösung ergibt sich die bereits bekannte Imputationsmenge, reduziert sich doch die Anzahl der n Gleichungen auf lediglich eine, die eben genau die für die Imputationsmenge geforderte Eigenschaft nach kollektiver Rationalität beschreibt.

Formel 70: Forderung nach kollektiver Rationalität

$$x_1 + x_2 + x_3 + \cdots + x_n = v(N)$$

Die Art und Weise wie ein für das Gately Konzept nicht eindeutiges Spiel definiert werden kann, wird im nächsten Abschnitt thematisiert.

4.1.5.4 Konstruktion eines Spiels mit nichteindeutigem Gately Punkt

Nachdem im Abschnitt 4.1.5.3 die Nichteindeutigkeit des Gately Punktes in der Theorie bewiesen wurde, geht es im Folgenden darum, nach welchem Muster solche nichteindeutigen Spiele konstruiert werden können.

Dies erfolgt erst für den Dreipersonenfall, die Erkenntnisse für die Umsetzung werden später auf den n-Personenfall ($n > 3$) übertragen.

Als wichtiges Kriterium wurde hierfür bereits eine *Equal Propensity To Disrupt* von -1 erarbeitet, da sich hierdurch ein unterbestimmtes Gleichungssystem ergibt.

Um derartige Spiele mit einer *Propensity To Disrupt* von -1 zu definieren, kann erneut die Formel von LITTLECHILD und VAIDYA (1976: 153) herangezogen und deren Wert auf minus 1 spezifiziert werden.

Formel 71: Equal Propensity To Disrupt mit $d^* = -1$ als Ausgangspunkt für die Konstruktion eines Spiels mit nichteindeutigem Gately Punkt

$$d^* = \frac{(n-1)v(N) - \sum_{S:\,|S|=1} v(N-S)}{v(N) - \sum_{S:\,|S|=1} v(S)} = -1$$

Es folgen einige triviale Umformungen.

Formel 72: Umformungen der Formel für die Equal Propensity To Disrupt mit $d^* = -1$ für die Konstruktion eines Spiels mit nichteindeutigem Gately Punkt

$$1.: (n-1)v(N) - \sum_{S:\,|S|=1} v(N-S) = \sum_{S:\,|S|=1} v(S) - v(N)$$

$$2.: (n-1)v(N) + v(N) = \sum_{S:\,|S|=1} v(N-S) + \sum_{S:\,|S|=1} v(S)$$

Durch die Umstellungen der Formel werden einige Anforderungen an den Spielevektor direkt ersichtlich. So muss für ein Spiel mit nichteindeutiger Lösung beim Gately Konzept (neben $d^* = -1$) die Summe für alle Koalitionswerte der Einerkoalitionen und deren Komplemente den Gesamtwert von $n * v(N)$ ergeben.

Formel 73: Resultat der Umformungen der Formel für die Equal Propensity To Disrupt mit $d^* = -1$ für die Konstruktion eines Spiels mit nichteindeutigem Gately Punkt

$$n * v(N) = \sum_{S:\,|S|=1} v(N-S) + \sum_{S:\,|S|=1} v(S)$$

Im Weiteren muss die Summe der Einerkoalitionen echt kleiner sein als der Wert für die große Koalition. Zum einen für die Wahrung der Bedingung für kollektive Rationalität und zum anderen für die strikte individuelle Rationalität.

Formel 74: Bestimmung der oberen Schranke für die Summe der Einerkoalitionswerte zur Konstruktion eines Dreipersonenspiels mit nichteindeutigem Gately Punkt

(1) $Forderung\ nach\ kollektiver\ Rationalität$:

$x_1 + x_2 + x_3 = v(N)$

(2) $Forderung\ nach\ strikter\ individueller\ Rationalität$:

$x_1 > v(\{1\})$

$x_2 > v(\{2\})$

$x_3 > v(\{3\})$

$Aus\ (1)\ und\ (2)\ resultierende\ Schranke\ für\ Einerkoalitionen$

$\Rightarrow v(\{1\}) + v(\{2\}) + v(\{3\}) < v(N)$

Bei Gleichheit mit $v(N)$, ist die *Equal Propensity To Disrupt* (siehe Formel 57) nämlich hier mit $v(N) - \sum_{S:\,|S|=1} v(S)$ aufgrund der daraus resultierenden Division durch 0 – bei der Formel zur Bestimmung der *Equal Propensity To Disrupt* – nicht definiert und die nichtstrikte individuelle Rationalität ist in diesem Fall nicht mehr ausreichend.

Für ein Dreipersonenspiel ergibt sich hierdurch der Spielvektor in der allgemeinen Form nach Formel 75.

Formel 75: Definition eines allgemeinen Dreipersonenspieles mit nichteindeutigem Gately Punkt

$$A = \begin{cases} v(\{1\}) \\ v(\{2\}) \\ v(\{3\}) \\ v(N) - v(\{3\}) \\ v(N) - v(\{2\}) \\ v(N) - v(\{1\}) \\ v(N) \end{cases}$$

$$u.d.N.: v(\{1\}) + v(\{2\}) + v(\{3\}) < v(N)$$

Zur Konstruktion eines Dreipersonenspiels mit den vorher beschriebenen Eigenschaften – der Imputations- als Lösungsmenge und $d^* = -1$ – wird zuerst ein beliebiger reeller Wert größer 0 für die große Koalition gewählt.

Im Anschluss sind die Werte für die Einerkoalitionen unter Berücksichtigung der Nebenbedingung $v(\{1\}) + v(\{2\}) + v(\{3\}) < v(N)$ entsprechend der strikten individuellen und kollektiven Rationalität (siehe Formel 74) festzulegen.

Die Werte der verbleibenden Koalitionen ergeben sich, indem hier die Differenz von dem Wert der großen Koalition mit dem Wert der komplementären Einerkoalition gebildet wird.

Für die Bildung eines entsprechenden Spielevektors für eine beliebige Anzahl von Spielern, wird analog zu der Vorgehensweise für das Dreipersonenspiel verfahren. Der deutliche Unterschied liegt dabei darin, dass für die Koalitionswerte von Koalitionen, bei denen es sich weder um Einerkoalitionen noch deren Komplemente handelt, beliebige Werte zugewiesen werden können.

Der Indexbereich der Elemente mit dem freiwählbaren Wert, startet hierbei bei $n + 1$ und läuft bis $l = 2^n - n - 2$.

Formel 76: Definition eines allgemeinen n-Personenspieles (n>3) mit nichteindeutigem Gately Punkt

$$A = \begin{cases} v(\{1\}) \\ v(\{2\}) \\ v(\{3\}) \\ \vdots \\ v(\{(n-1)\}) \\ v(\{n\}) \\ w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_{l-1} \\ w_l \\ v(N) - v(\{n\}) \\ v(N) - v(\{(n-1)\}) \\ \vdots \\ v(N) - v(\{3\}) \\ v(N) - v(\{2\}) \\ v(N) - v(\{1\}) \\ v(N) \end{cases}$$

$$u.d.N.: v(\{1\}) + v(\{2\}) + v(\{3\}) + \cdots + v(\{n\}) < v(N)$$

Für das Konstruieren eines solchen Spielevektors wurde im Paket `CoopGame` die Funktion `getNondefiniteGameVector4GatelyValue` bereitgestellt. Mit dem Parameter `vN` wird hierbei der gewünschte Wert für die große und mit `vSCs` der Vektor für die Einerkoalitionen festgelegt. Der Parameter `w` spezifiziert im Fall $n > 3$ die Koalitionswerte der übrigen Koalitionen mit Ausnahme für die Komplemente der Einerkoalitionen; diese leiten sich von `vSCs` und `vN` ab. Die restlichen Koalitionen sind für die Berechnung der Gately-Lösung unerheblich.

Das Codebsp. 9 gibt hierfür ein Beispiel. In Zeile 1 wird die Funktion mit 10 für die große Koalition, 0 *bis* 3 für die Einerkoalitionen sowie 6 mal 99 für die unerheblichen Koalitionen aufgerufen. Dies hat den in Zeile 3 ausgegebenen Spielevektor zur Folge, dessen *Equal Propensity To Disrupt* sich auf -1 beläuft (siehe Zeile 5).

```
> A=getNondefiniteGameVector4GatelyValue(10,c(0,1,2,3),w=rep(99,6))   #1
> A                                                                   #2
 [1]  0  1  2  3 99 99 99 99 99 99  7  8  9 10 10                    #3
> equalPropensityToDisrupt(A)                                        #4
[1] -1                                                               #5
```

Codebsp. 9: Verwendung getNondefiniteGameVector4GatelyValue Methode zur Bestimmung eines nichteindeutigen Spiels hinsichtlich des Gately Punktes

An diesem Beispiel wird auch ein Kritikpunkt am Gately Punkt deutlich. So vernachlässigt er Koalitionen, bei denen es sich nicht um die große Koalition und Einerkoaltionen bzw. deren Komplemente handelt, vollständig. In diesem Fall wird wohl kaum eine große Koalition zu-

stande kommen, wenn zur Bildung von beliebigen Zweierkoalitionen ein viel größerer Anreiz besteht.

4.1.5.5 Umsetzung des Gately Punktes in R

Bei der Implementierung des Gately Punktes (siehe Codebsp. 10) wird zuerst überprüft (siehe Zeile 13), ob für den Spielevektor keine strikte individuelle Rationalität (siehe Formel 74) vorliegt. Sollte dies der Fall sein, wird der `else`-Block übersprungen und ein NULL Wert für den Gately Punkt zurückgegeben.

Ansonsten wird im Anschluss die *Equal Propensity To Disrupt* berechnet (siehe Zeile 13) und bei $d^* \neq -1$ folgendes Gleichungssystem mit der internen R Funktionalität `solve` gelöst (siehe Zeile 34).

Formel 77: Gleichungssystems zur Bestimmung des Gately Punkts

$$\begin{array}{ccccccccc}
-d^*x_1 & +x_2 & +x_3 & +\cdots+ & x_n & = & v(N-\{1\}) & -d^*v(\{1\}) \\
x_1 & -d^*x_2 & +x_3 & +\cdots+ & x_n & = & v(N-\{2\}) & -d^*v(\{2\}) \\
x_1 & +x_2 & -d^*x_3 & +\cdots+ & x_n & = & v(N-\{3\}) & -d^*v(\{3\}) \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
x_1 & +x_2 & +x_3 & +\cdots+ & -d^*x_n & = & v(N-\{n\}) & -d^*v(\{n\})
\end{array}$$

Falls aber $d^* = -1$ erfüllt ist, muss für alle Einerkoalitionen i geprüft werden, ob deren Koalitionswert addiert mit dem des Komplements den Wert der großen Koalition (siehe Formel 78) ergibt (siehe Zeile 20) – aufgrund der erforderlichen kollektiven Rationalität (siehe Formel 70: Forderung nach kollektiver Rationalität).

Formel 78: Forderung nach dem Koalitionswert der großen Koalition für jeweils die Summe aller Werte der Einerkoalitionen und deren Komplemente

$$v(N-\{i\}) + v(\{i\}) = v(N): \forall i \in N$$

Sind alle hier geprüften Werte gleich dem der großen Koalition, ist die strikte Imputationsmenge die entsprechende Lösung, als Lösung wird stellvertretend der Schwerpunkt der Imputationsmenge zurückgegegeben. Ansonsten existiert keine Lösung.

```
gatelyValue<-function(A){                                                    #1
        paramCheckResult=getEmptyParamCheckResult()                          #2
        initialParamCheck_gatelyValue(paramCheckResult,A)                    #3
                                                                             #4
        #assignnumberofcoalitions                                            #5
        N<-length(A)                                                         #6
                                                                             #7
        #assignnumberofplayers                                               #8
        n<-getNumberOfPlayers(A)                                             #9
                                                                             #10
        gatelyValue<-NULL                                                    #11
                                                                             #12
        if(!((sum(A[1:n])<A[N]))){                                           #13
            print("Gately Value does not exist as A shows no strict imputation set.")  #14
        }else{                                                               #15
            eptd=equalPropensityToDisrupt(A,k=1)                             #16
            if(eptd==-1){                                                    #17
                additionVectorComplements=sapply(                            #18
                    1:n,                                                     #19
                    function(ix){A[ix]+A[N-ix]}                              #20
                )                                                            #21
                if(all(additionVectorComplements==A[N],TRUE)){               #22
                    msg="Gately Value is not unique as equal with imputation set,"  #23
                    msg=paste(msg,"therefore centroid of the imputation set" )  #24
                    setVertices=imputationsetVertices(A)                     #25
                    centroid=colSums(setVertices)/n                          #26
                    gatelyValue=centroid                                     #27
                    print(msg)                                               #28
                }else{                                                       #29
                    print("No solution existing")                            #30
                }                                                            #31
            }else{                                                           #32
                coeffMat<-createBitMatrix(n,A)[indexLower(n,n-1):(N-1),]     #33
                coeffMat[,1:n][coeffMat[,1:n]==0]=-eptd                      #34
                coeffMat[,"cVal"]=rev(                                       #35
                  sapply(                                                    #36
                    1:n,                                                     #37
                    function(ix){                                            #38
                        coeffMat[((n+1)-ix),"cVal"]-eptd*A[ix]               #39
                    }                                                        #40
                  )                                                          #41
                )                                                            #42
                gatelyValue=tryCatch(                                        #43
                    solve(coeffMat[,1:n],coeffMat[,"cVal"]),                 #44
                    error=function(x){NULL}                                  #45
                )                                                            #46
            }                                                                #47
            if(!isImputation(A,gatelyValue)){                                #48
                gatelyValue=NULL                                             #49
                print("Calculated value is no imputation hence no Gate Value exists.")  #50
            }                                                                #51
        }                                                                    #52
        return(gatelyValue)                                                  #53
}                                                                            #54
```

Codebsp. 10: gatelyValue Funktion

4.2 Nucleolus und Varianten

Der Nucleolus ist prinzipiell als Mengenansatz zu klassifizieren, der eine Menge von Auszahlungsvektoren enthält (HOLLER & ILLING 2006: 300 und SCHMEIDLER 1969).

Erscheint diese Aussage zwar auf den ersten Blick im Kontext der in diesem Kapitel behandelten punktwertigen Lösungskonzepte und dem Nucleolus sowie dessen Derivaten widersprüchlich, lässt sich diese doch durch die Beschränkung auf nur wesentliche Spiele relativieren (PETERS 2015: 159).

Die gewünschte Eindeutigkeit und die damit verbundene Eignung als punktwertiges Lösungskonzepts kann im Weiteren für den Nucleolus und dessen Derivate unter Ausschluss von nichtwesentlichen Spielen erzwungen werden (PETERS 2015: 343&347, PELEG & SUDHÖLTER 2007: 84f).

Eine solche Einschränkung stellt zudem keinen großen Nachteil dar, da aus ökonomischer Sicht die nichtwesentlichen Spiele bei Gewinnverteilungsproblemen von vornherein auf kein großes Interesse stoßen. Der große Zusammenschluss für ein oder mehrere Spieler zahlt sich hier auf keinerlei Weise aus (ZELEWSKI 2009: 44).

Im Weiteren folgt eine Kurzvorstellung des Nucleolus und mit diesem verwandte Ansätze wie des Disruption Nucleolus, Modiclus, Per Capita Nucleolus, Prenucleolus, Proportional Nucleolus und des Simplified Modiclus.

4.2.1 Nucleolus und Prenucleolus

In Allgemeinen handelt es sich beim Nucleolus (SCHMEIDLER 1969) um ein mengenwertiges Lösungskonzept, das die Menge von Auszahlungsvektoren bestimmt, welche *„die Überschüsse der Koalitionen eines Spiels und damit das Potential möglicher Einwände der Spieler gegen einen Auszahlungsvektor […] minimieren“* (HOLLER & ILLING 2006: 300).

PETERS (2015: 159) definiert jedoch zusätzliche Einschränkungen für das Spiel, so dass die Eindeutigkeit immer gewahrt bleibt.

Nach der Definition von PETERS (2015: 159) ist der Nucleolus ein Lösungskonzept, das jedem wesentlichen Spiel mit der damit implizierten nichtleeren Imputationsmenge eine eindeutige Imputation zuweist, die sich zudem innerhalb des Kerns befindet, sofern dieser nicht leer ist.

Der Prenucleolus unterscheidet sich vom Nucleolus nur dahingehend, dass dieser für das Spiel nicht zwingend eine Imputation bestimmt, sondern gegebenenfalls auch lediglich eine Präimputation (PETERS 2015: 347).

In Spielen die balanciert sind, stimmen Nucleolus und Prenucleolus in einer Imputation überein (PETERS 2015: 343).

Im Weiteren richten sich die Ausführungen zum Nucleolus und Prenucleolus für das Verständnis als punktwertige Lösungskonzepte nach der Definition des eindeutigen Nucleolus von PETERS (2015: 159).

4.2.1.1 Definition eines eindeutigen Nucleolus nach Peters

Die Voraussetzungen für einen eindeutigen Nucleolus sind erfüllt, wenn der Auszahlungsraum X, die geforderten Kriterien im Theorem (siehe Formel 79) zur Eindeutigkeit des allgemeinen Nucleolus nach PETERS (2015: 347) und (HOLLER & ILLING 2006: 300&301) wahrt. Demnach sind der Nucleolus bzw. dessen Derivate eindeutig, wenn diese einem nichtleeren, kompakten und konvexen Auszahlungsraum zugrunde liegen.

Formel 79: Definition des Theorems zur Eindeutigkeit des allgemeinen Nucleolus

$$Sei\ X \subseteq \mathbb{R}^n\ nichtleer, kompakt\ und\ konvex,$$
$$dann\ besteht\ der\ allgemeine\ Nucleolus\ hinsichtlich$$
$$des\ Auszahlungsraum\ X\ f\text{ür}\ jedes\ Spiel\ (N, v)$$
$$aus\ einem\ einzigen\ Punkt.$$

PETERS (2015: 347) sieht im Weiteren bei wesentlichen Spielen (N, v) die in Formel 79 beschriebenen Kriterien erfüllt.

Nach PETERS (2015: 159) ist daher ein eindeutiger Nucleolus für wesentliche Spiele (N, v) mit $v(N) \geq \sum_{i \in N} v(\{i\})$ definiert.

In diesem Fall ist die Forderung nach der Wesentlichkeit des Spiels für einen eindeutigen Nucleolus gleichbedeutend mit der nach einer nichtleeren Imputationsmenge, so gilt $v(N) \geq \sum_{i \in N} v(\{i\})$ g. d. w. $I(N, v) \neq \emptyset$.

Den Zulässigkeitsbereich der Auszahlungsvektoren $x \in I(N, v)$ eines eindeutigen Nucleolus spiegelt nach PETERS (2015: 159) hierbei die Imputationsmenge $I(N, v)$ wider, für deren Elemente die Eigenschaft der individuellen und kollektiven Rationalität erfüllt sind.

Der Prenucleolus x weist dagegen als Zulässigkeitsbereich die Präimputationsmenge $I^*(N, v)$ auf, bei der die Forderung nach der individuellen Rationalität entfällt (PETERS 2015: 347). Die Präimputationsmenge kommt zwar hierbei nicht der Forderung nach Kompaktheit nach, ist aber laut PETERS (2015: 347) dennoch ein bezüglich des Prenucleolus eindeutiger Auszahlungsraum.

Im Folgenden werden die Definitionen eines allgemeinen und mengenwertigen Nucleolus von PETERS (2015: 345ff) aufgegriffen und insofern abgewandelt, dass sie den bereits erörterten Kriterien eines eindeutigen Nucleolus bzw. Prenucleolus nach PETERS (2015: 159) entsprechen.

Innerhalb des jeweiligen Zulässigkeitsbereichs der Auszahlung erfolgt die Bestimmung der Prä- bzw. Imputation, welche die maximalen Überschüsse (engl. excess) minimiert.

Formel 80: Definition des Überschusses (engl. excess)

$$e(S, x, v) = v(S) - x(S)$$

Der Vektor θ mit den Überschüssen für alle Koalitionen $S \subseteq N$ und Auszahlungsvektoren $x \in I(N, v)$ im Fall des Nucleolus bzw. $x \in I^*(N, v)$ für den Prenucleolus lässt sich in eine absteigende lexikalische Ordnung (siehe Abschnitt 0) versetzen.

Formel 81: Defnition der lexikalischen Ordnung der Überschüsse

$$\theta(x) = \big(e(S_1, x, v), e(S_2, x, v), \dots, e(S_{2^n-1}, x, v)\big), so\ dass$$
$$e(S_1, x) \succcurlyeq_{lex} e(S_2, x) \succcurlyeq_{lex} \dots \succcurlyeq_{lex} e(S_{2^n-1}, x)$$

Ein eindeutiger Nucleolus ist auf Grundlage der lexikalischen Ordnung der Überschüsse (siehe Formel 81) wie in Formel 82 definiert und liefert mit x eine eindeutige Imputation, welche die maximalen Überschüsse lexikographisch und auf optimale Weise minimiert.

Formel 82: Definition eines eindeutigen Nucleolus

$$N(N, v, I) = \{x \in I \mid \theta(y) \succcurlyeq_{lex} \theta(x)\ \forall y \in I\}, so\ dass$$
$$card(N(N, v, I)) = 1$$

Entsprechend ist der Prenucleolus x auf der Präimputationsmenge $I^*(N, v)$ als eindeutige Präimputation definiert (siehe Formel 83), welche ebenfalls die maximalen Überschüsse lexikographisch minimiert.

Formel 83 Definition eines eindeutigen Prenucleolus

$$PN(N, v, I^*) = \{x \in I^* \mid \theta(y) \succcurlyeq_{lex} \theta(x)\ \forall y \in I^*\}, so\ dass$$
$$card(PN(N, v, I^*)) = 1$$

In diesem Abschnitt wurde der Nucleolus und Prenucleolus nach PETERS (2015: 159&347) derart definiert, dass die geforderte Eindeutigkeit und die damit verbundene Eignung als punktwertiges Lösungskonzept für den Nucleolus und auch dessen Derivate sichergestellt wurden. Allerdings besteht jedoch durchaus die Möglichkeit, den Nucleolus für andere Auszahlungsräume eindeutig zu gestalten, solange diese nach den bereits erwähnten Kriterien aus Formel 79 nichtleer, kompakt und konvex sind.

4.2.1.2 Berechnung des Nucleolus bzw. Prenucleolus

Der folgende Abschnitt skizziert in groben Zügen die Schritte zur Berechnung des Nucleolus bzw. Prenucleolus in enger Anlehnung an MASCHLER et al. (2013: 815f) und PETERS (2015: 350-353).

Die hier beschriebene Vorgehensweise lässt sich dabei unter Berücksichtigung der jeweiligen Eigenheiten auch auf die Bestimmung der übrigen Nucleolus Derivate übertragen.

Der Nucleolus wird dabei durch eine Folge linearer Programme berechnet und identifiziert hierbei im ersten Schritt alle Auszahlungen X_1 des Auszahlungsraumes X für die Menge der Koalitionen Σ_1, bei denen das Minimum des Überschusses θ_1 erreicht wird (siehe Formel 84).

Formel 84: Identifizierte Auszahlungen aus dem ersten Schritt der (Pre-)Nucleolus Berechnung

$$X_1 = \{x \in X(N,v) : e(S,x,v) \leq \theta_1, \forall S \subseteq N\}$$

Die Menge Σ_1 umfasst hierbei diejenigen Koalitionen, die als maximalen Überschuss θ_1 erreichen (siehe Formel 85).

Formel 85: Identifizierte Koalitionen aus dem ersten Schritt der (Pre-)Nucleolus Berechnung mit θ_1 als maximalem Überschuss

$$\Sigma_1 = \{S \subseteq N : e(S,x,v) = \theta_1, \forall x \in X_1\}$$

Das Lineare Programm im ersten Schritt ist dabei wie in Formel 86 definiert. Dabei gibt die Entscheidungsvariable t der Zielfunktion den zu minimierenden Überschuss wieder. Die Nebenbedingung (1) beschreibt, dass der Überschuss für alle Koalitionen zu minimieren ist und (2) sowie (3) beschränken den Auszahlungsraum auf die Imputationsmenge.

Für den Prenucleolus gilt hier zwar, wie in der Restriktion (2) beschrieben, die Forderung nach kollektiver Rationalität $x(N) = v(N)$, die Einschränkung nach individueller Rationalität aus (3) $x_i \geq v(i)$ für alle Spieler $i \in N$ entfällt aber für alle aufgestellten Linearen Programme hinsichtlich der Präimputationsmenge als Auszahlungsraum. Die Forderung nach Nichtnegativität für den Auszahlungsvektor $x_i \geq 0$ bleibt aber nach SCHMEIDLER (1969: 231) im Sinne einer gültigen Auszahlung weiterbestehen.

Formel 86: Lineares Programm zur Berechnung des Nucleolus im ersten Schritt

$$Minimiere:$$
$$z = F(x_1, \dots, x_n, t) = t$$

$$Unter\ Einhaltung\ der\ Nebenbedingungen:$$

(1) $e(S, x, v) \leq t, \forall S \subset N$

(2) $\sum_{i \in N} x_i = v(N)$

(3) $x_i \geq v(i), \forall i \in N$

Im zweiten Schritt erfolgt analog im Anschluss eine Reduktion unter allen identifizierten Auszahlungen X_1 durch ein aktualisiertes Lineares Programm (siehe Formel 87) auf die Auszahlungen X_2, welche wiederum den zweitgrößten Überschuss so weit wie möglich minimieren.

Hierzu wird für die Menge der Koalitionen Σ_1 (siehe Restriktionen nach 1a), bei denen der maximale Überschuss nicht weiter minimiert werden kann, θ_1 als Schrankenwert festgelegt, um in Folge X_2 sowie Σ_2 bestimmen zu können.

Formel 87: Lineares Programm zur Berechnung des Nucleolus im zweiten Schritt

$Minimiere$:

$z = F(x_1, \dots, x_n, t) = t$

$Unter\ Einhaltung\ der\ Nebenbedingungen$:

$(1a)\ e(S, x, v) = \theta_1, \forall S \in \Sigma_1$

$(1b)\ e(S, x, v) \leq t, \forall S \notin \Sigma_1$

$(2)\ x(N) = v(N)$

$(3)\ x_i \geq v(i), \forall i \in N$

Das beschriebene Vorgehen aus dem zweiten Schritt wiederholt sich sooft, bis mit X_L eine einzigartige Imputation oder Präimputation für den Nucleolus bzw. Prenucleolus bestimmt worden ist. Die Anzahl der Schritte L ist dabei maximal $2^n - 1$ für den Fall, dass bei jeder Iteration nur der maximale Überschuss für jeweils eine Koalition minimiert werden kann.

Es folgt ein kleines eindeutiges Beispiel für den Nucleolus und den Prenucleolus.

4.2.1.3 Beispiel für Nucleolus und Prenucleolus

Als Nucleolus und Prenucleolus-Beispiel dient ein nichtbalanciertes Spiel mit den in Formel 88 beschriebenen Koalitionswerten. Aufgrund der Nichtbalanciertheit werden für den Nucleolus und Prenucleolus unterschiedliche Lösungswerte ermöglicht (PETERS 2015: 343).

Formel 88: Spielevektor für Beispiel zu (Pre-)Nucleolus

Werte der Einerkoalitionen:

$$v(1) = 2;\ v(2) = 6;\ v(3) = 5$$

Werte der Zweierkoalitionen:

$$v(1,2) = 15;\ v(1,3) = 1$$
$$v(2,3) = 18$$

Wert der Großen Koalition:

$$v(1,2,3) = 14$$

Für dieses Beispiel wird in Codebsp. 11 jeweils der Nucleolus (siehe Zeile 4) und Prenucleolus (siehe Zeile 6) berechnet, vorab wird jedoch das definierte Spiel (siehe Zeile 1) auf Balanciertheit geprüft (siehe Zeile 2). Die Prüfung ergibt (siehe Zeile 3), dass es sich hierbei um kein balanciertes Spiel handelt und sich somit auch der Nucleolus (siehe Zeile 5) vom Prenucleolus (siehe Zeile 7) unterscheiden kann, aber nicht zwangsweise muss.

```
> A<-c(2,6,5,15,1,18,14)                          #1
> isBalancedGame(A)                               #2
[1] FALSE                                         #3
> nucleolus(A, enableTermOutLP = FALSE)           #4
[1] 2 7 5                                         #5
> prenucleolus(A, enableTermOutLP = FALSE)        #6
[1] 0 12  2                                       #7
```

Codebsp. 11: Nucleolus und Prenucleolus Beispiel

Im Weiteren werden für die bestimmten Lösungen des Nucleolus und Prenucleolus, die Überschüsse ermittelt und so deren Unterschiede direkt am Beispiel verdeutlicht (siehe Tabelle 13).

Tabelle 13: Überschüsse von Nucleolus und Prenucleolus im Beispiel

Index	**Bit-Zeile für S**	**Spieler $\in S$**	$v(S)$	$e(S, N, v)$ **für** $N(N, v, I) = (2, 7, 5)$	$e(S, PN, v)$ **für** $PN(N, v, I^*) = (0, 12, 2)$	**Differenz exc.**
1	1 0 0	{1}	2	$2 - (2) = 0$	$2 - (0) = 2$	-2
2	0 1 0	{2}	6	$6 - (7) = -1$	$6 - (12) = -6$	5
3	0 0 1	{3}	5	$5 - (5) = 0$	$5 - (2) = 3$	-3
4	1 1 0	{1,2}	15	$15 - (2 + 7) = 6$	$15 - (0 + 12) = 3$	3
5	1 0 1	{1,3}	1	$1 - (2 + 5) = -6$	$1 - (0 + 2) = -1$	-5
6	0 1 1	{2,3}	18	$18 - (7 + 5) = 6$	$18 - (12 + 2) = 4$	3
7	1 1 1	{1,2,3}	14	$14 - (2 + 7 + 5) = 0$	$14 - (0 + 12 + 2) = 0$	0
Summe				5	5	0

Für den Nucleolus ergibt sich auf der Imputationsmenge mit $N(N, v, I) = (2,7,5)$ der Überschussvektor $\theta\big(N(N, v, I)\big)$, siehe Formel 89.

Formel 89: Vektor $\theta(N(N,v,I))$ mit Überschüssen des Nucleolus in lexikographischer Ordnung (Beispiel)

$$\theta\big(N(N,v,I)\big)$$
$$= \big(e(S_6,N,v), e(S_4,N,v), e(S_7,N,v), e(S_3,N,v), e(S_1,N,v), e(S_2,N,v), e(S_5,N,v)\big)$$
$$= (6,6,0,0,0,-1,-6)$$

Für den Prenucleolus ist der Überschussvektor $\theta(PN(N,v,I^*))$ auf der Imputationsmenge wie in Formel 90 definiert.

Formel 90: Vektor $\theta(PN(N,v,I^*))$ mit Überschüssen des Prenucleolus in lexikographischer Ordnung (Beispiel)

$$\theta(PN(N,v,I^*))$$
$$= \big(e(S_6,PN,v), e(S_4,PN,v), e(S_3,PN,v), e(S_1,PN,v), e(S_7,PN,v), e(S_5,PN,v), e(S_2,PN,v)\big)$$
$$= (4,3,3,2,0,-1,-6)$$

Sowohl der Nucleolus, als auch der Prenucleolus minimieren den maximalen Überschuss lexikographisch auf das jeweilige Optimum. Die beiden Lösungen können jedoch stark voneinander abweichen, wie in dem in diesem Abschnitt behandelten Beispiel.

Durch Bildung der Differenz aus dem Überschuss des Nucleolus und des Prenucleolus für jede Koalition S (siehe Tabelle 12, letzte Spalte) wird verdeutlicht, inwieweit die jeweilige Koalition eher mit einer Aufteilung nach dem Verteilungsschlüssel des Nucleolus bzw. des Prenucleolus-Lösungskonzept zufrieden ist. Ein positiver Differenzwert zeugt von einer Präferenz der Koalition weg vom Nucleolus hin zum Prenucleolus, ein negativer vom Verbleib beim Nucleolus und ein Nullwert von Neutralität. Kollektiv betrachtet ergibt sich aber über alle Koalitionen hinweg immer der Nullwert.

Im Weiteren werden die Derivate des Nucleolus kurz erörtert.

4.2.1.4 Beispiel eines nichteindeutigen Nucleolus

Ein nicht eindeutiger Nucleolus ist gegeben (siehe Abschnitt 4.2.1.1), wenn der zugrundeliegende Auszahlungsraum leer oder die Kriterien für Kompaktheit bzw. Konvexität nicht erfüllt sind.

Als Beispiel für die Konstruktion eines nichteindeutigen Nucleolus dient ein symmetrisches Dreipersonenspiel (N,v) mit den in Formel 91 beschriebenen Koalitionswerten.

Formel 91: Spielevektor für ein nicht eindeutiges Beispiel zum Nucleolus

Werte der Einerkoalitionen:

$$v(1) = 2 = v(2) = v(3) = 2$$

Werte der Zweierkoalitionen:

$$v(1,2) = v(1,3) = v(2,3) = 4$$

Wert der Großen Koalition:

$$v(1,2,3) = 24$$

Die Imputationsmenge zu dem hier beschriebenen Spiel ist nicht leer und aufgrund der Symmetrieeigenschaft findet sich der Nucleolus auf Basis der Imputationsmenge als zulässigen Auszahlungsraum X im Mittelpunkt des Imputationsdreiecks durch die Auszahlung $x = (8,8,8)$ wieder.

Für den Nucleolus und das spezifizierte Spiel ergibt sich hierdurch der in Formel 92 beschriebene Überschussvektor $\theta(x)$.

Formel 92: Überschussvektor des Nucleolus für Auszahlungsraum X im nicht eindeutigen Beispiel

$$\theta(N(N,v,X) = \theta\big((8,8,8)\big) = (-12,-12,-12,-6,-6,-6,0)$$

Durch Definition zusätzlicher Einschränkung auf den bisherigen Auszahlungsraum der Imputationsmenge $I(N,v)$ in Form der in Formel 93 spezifizierten oberen Schranken für den Auszahlungsvektor x, geht die Eindeutigkeit des Nucleolus verloren.

Formel 93: Zusätzliche oberer Schranken für den Auszahlungsraum im nicht eindeutigen Beispiel zum Nucleolus

$$\begin{array}{cccc} x_1 & +x_2 & & \leq & 19 \\ x_1 & & +x_3 & \leq & 19 \\ & x_2 & +x_3 & \leq & 19 \end{array}$$

Der neue Auszahlungsraum X′ (siehe Formel 94) ist nicht mehr konvex und auch die Imputation $x = (8,8,8)$ ist als ursprüngliche Lösung hiervon betroffen.

Formel 94: Für den neuen Auszahlungsraum X′ geltende Bedingungen im Beispiel zum nichteindeutigen Nucleolus

$$\begin{array}{cccc} x_1 & & & \geq & 2 \\ & x_2 & & \geq & 2 \\ & & x_3 & \geq & 2 \\ x_1 & +x_2 & +x_3 & = & 24 \\ x_1 & +x_2 & & \leq & 19 \\ x_1 & & +x_3 & \leq & 19 \\ & x_2 & +x_3 & \leq & 19 \end{array}$$

Durch die zusätzlichen Einschränkungen aus Formel 93 sind alle Auszahlungen innerhalb des Dreiecks mit den Eckpunkten $(5, 9.5, 9.5)$, $(9.5, 5, 9.5)$ und $(9.5, 9.5, 5)$ im neuen Auszahlungsraum X' nicht mehr enthalten und somit auch die Imputation $x = (8{,}8{,}8)$.

Der Nucleolus besteht nun für das gegebene Spiel und den neuen Auszahlungsraum X'aus den Imputationen, die auf den Kanten des Dreiecks mit den Eckpunkten $(5, 9.5, 9.5)$, $(9.5, 5, 9.5)$ und $(9.5, 9.5, 5)$ liegen. Diese Imputationen lassen sich durch das in Formel 95 aufgeführte Gleichungssystem beschreiben.

Formel 95: Gleichungssystem für die Lösungsmenge des Nucleolus mit Auszahlungsraum X' im nicht eindeutigen Beispiel

$$
\begin{array}{rcrcl}
5 & \leq & x_1 & \leq & 14 \\
5 & \leq & x_2 & \leq & 14 \\
5 & \leq & x_3 & \leq & 14 \\
x_1 & +x_2 & & = & 19 \\
x_1 & & +x_3 & = & 19 \\
 & x_2 & +x_3 & = & 19 \\
x_1 & +x_2 & +x_3 & = & 24
\end{array}
$$

Für den Nucleolus und das spezifizierte Spiel ergibt sich hinsichtlich des neuen Auszahlungsraum X' der in Formel 96 beschriebene Überschussvektor $\theta(N(N, v, X'))$.

Formel 96: Überschussvektor des Nucleolus für den Auszahlungsraum X' im nichteindeutigen Beispiel

$$\theta(N(N, v, X') = (-15, -15, -12, -6, -3, -3{,}0)$$

Das in diesem Abschnitt behandelte Beispiel sollte helfen, die Nichteindeutigkeit des allgemein spezifizierten Nucleolus zu demonstrieren und zu belegen.

Der Nucleolus aus dem Paket `CoopGame` sowie dessen Derivate sind jedoch auf dem Auszahlungsraum der Imputations- bzw. Präimputationsmenge spezifiziert und somit anders als im Beispiel für wesentliche Spiele garantiert punktwertig.

4.2.2 Proportional Nucleolus

Beim Proportional Nucleolus handelt es sich um ein von YOUNG, OKADA und HASHIMOTO (1982) im Zusammenhang mit Wasserinfrastrukturprojekten vorgestelltes Lösungskonzept zur anteilsmäßigen Aufteilung der hierzu nötigen Investitionskosten.

In der zugehörigen Arbeit von YOUNG et al. (1982: 463) wurden dabei verschiedene anteilsmäßige Kostenaufteilungskonzepte wie unter anderem der Shapley Wert, andere mit dem Kern verwandte Konzepte und besagter Proportional Nucleolus miteinander verglichen.

Die Autoren kamen zwar in dem Paper zu der Schlussfolgerung, dass es nicht die beste Methode als solche gibt (YOUNG et al.1982: 463), der Proportional Nucleolus aber unter allen anderen untersuchten Konzepten sehr positiv heraussticht, da er die meisten in der Arbeit getesteten Kriterien für Fairness erfüllt (YOUNG et al.1982: 464).

4.2.2.1 Definition des Proportional Nucleolus

Beim Proportional Nucleolus handelt es sich um ein Derivat des Nucleolus, das auf eine andere Art den Überschuss für das Anfangsproblem definiert, während der Lösungsalgorithmus unverändert bleibt (LEJANO et al.1999: 170).

Der Proportional Nucleolus verfolgt nach YOUNG et al. (1982: 467) die Intention, den Kern eines Spiels zu modifizieren, indem dieser allen Koalitionen eine Mindeststeuer im Verhältnis zu ihren Kosten auferlegt.

So wird ein Steuersatz t eingeführt und in Abhängigkeit von diesem das anfängliche Lineare Programm wie in Formel 97 formuliert (YOUNG et al. 1982: 467).

Formel 97: Definition des initiale Linearen Programms zur Berechnung des Proportional Nucleolus

$$Minimiere:$$

$$z = F(x_1, \dots, x_n, t) = t$$

$$Unter\ Einhaltung\ der\ Nebenbedingungen:$$

$$(1)\ \sum_{i \in S} x_i \geq (1-t)v(S), \forall S \subset N$$

$$(2)\ \sum_{i \in N} x_i = v(N)$$

$$(3)\ x_i \geq v(i), \forall i \in N$$

Die Nebenbedingung (3) aus Formel 97 wurde dabei selbstständig für die Beschränkung des Auszahlungsraumes auf ausschließlich Imputationen ergänzt.

4.2.2.2 Beispiel für Proportional Nucleolus

Das folgende Beispiel ist direkt aus WANG et al. (2007: 1800) entnommen und untersucht, wie durch Altlasten kontaminierte Industrieflächen hinsichtlich aller beteiligten Stakeholder möglichst kostengünstig wieder saniert und nutzbar gemacht werden können.

In einem hypothetischen Beispiel (WANG et al.2007: 1804) zur besseren Anschauung sind als Stakeholder der Landbesitzer (Spieler 1), der Erschließer (bzw. der Investor als Spieler 2) und die Regierung (Spieler 3) genannt. Für alle drei Gruppen besteht ein finanzielles Interesse, die Flächen wieder einer normalen Nutzung zu überführen. Jedoch ist dies mit erheblichen Kos-

ten verbunden, so dass – um eine gute Verhandlungsgrundlage zu schaffen – der Proportional Nucleolus zu Rate gezogen wird.

Hierfür fließen Kosten und Nutzenschätzungen in die Spielmodellierung (siehe Formel 98) mit ein.

Formel 98: Spielevektor für das Beispiel zum Proportional Nucleolus

Werte der Einerkoalitionen:

$$v(1) = 20;\ v(2) = 200;\ v(3) = 40$$

Werte der Zweierkoalitionen:

$$v(1,2) = 210;\ v(1,3) = 40$$
$$v(2,3) = 200$$

Wert der Großen Koalition:

$$v(1,2,3) = 350$$

Der Proportional Nucleolus für das modellierte Spiel (siehe Formel 98) wird mit der R-Funktion `proportionalNucleolus` aus `CoopGame` berechnet (siehe Codebsp. 12 in Zeile 2). Die Ergebniswerte $(36.992, 291.667, 21.341)$ entsprechen hierbei nach Rundung auf drei Stellen, exakt denen von WANG et al. (2007: 1805)

```
> A<-c(20,200,0,210,40,200,350)                                    #1
> proportionalNucleolus(A, enableTermOutLP = FALSE)                #2
[1]  36.99187 291.66667  21.34146                                  #3
> nucleolus(A, enableTermOutLP = FALSE)                            #4
[1]  63.33333 243.33333  43.33333                                  #5
> shapleyValue(A)$shapleyValue                                     #6
[1]  65 235  50                                                    #7
```

Codebsp. 12: Berechnungen zu Beispiel aus Abschnitt 4.2.2.2

WANG et al. (2007: 1805) geben gleichzeitig auch für andere Lösungskonzepte, die jeweiligen Ergebniswerte in einer Tabelle an (vgl. Tabelle 14).

Tabelle 14: Aufteilung für Beispiel für den Proportional Nucleolus – entnommen aus WANG et al. (2007: 1805)

Lösungskonzept	**Stakeholder**		
	(1) Landbesitzer	**(2) Investor**	**(3) Regierung**
Nucleolus	63.333	243.333	42.333
Proportional Nuc.	36.992	291.667	21.341
Shapley Wert	65	235	50

Auch für die übrigen Lösungskonzepte stimmen die Werte mit denen des Pakets `CoopGame` überein, vgl. Codebsp. 12 mit Tabelle 14.

4.2.3 Per Capita Nucleolus

Der Per Capita Nucleolus (YOUNG 1985) – oder „normalisierter" Nucleolus – ist eine Variante des gewöhnlichen Nucleolus, die den Überschuss $e(S, x, v)$ normalisiert. Die Normalisierung geschieht hierbei auf Grundlage der Anzahl, der an der Koalition S involvierten Spieler $card(S)$ (YOUNG 1985: 68).

4.2.3.1 Definition des Per Capita Nucleolus

Der Überschuss auf der Per Capita Basis ist wie in Formel 99 definiert. Hier findet eine Normierung des Überschusses in Form einer Division durch die Anzahl der an S beteiligten Spieler statt.

Formel 99: Definition des Überschusses auf Per Capita Basis

$$e(S, x, v) = \frac{v(S) - \sum_{i \in S} x_i}{|S|}$$

Auf der Grundlage des Überschusses auf der Per Capita Basis ergibt sich das in Formel 100 aufgeführte initiale Lineare Programm zur Berechnung des Per Capita Nucleolus.

Formel 100: Definition des initialen Linearen Programmes für die Berechnung des Per Capita Nucleolus

$Minimiere$:

$$z = F(x_1, \dots, x_n, t) = t$$

$Unter\ Einhaltung\ der\ Nebenbedingungen$:

$$(1) \sum_{i \in S} x_i + |S| t \geq v(S), \forall S \subset N$$

$$(2) \sum_{i \in N} x_i = v(N)$$

$$(3)\ x_i \geq v(i), \forall i \in N$$

Der Lösungsalgorithmus bleibt auch hier von den Änderungen unberührt.

4.2.3.2 Beispiel für Per Capita Nucleolus

Als Beispiel dient das Kostenaufteilungsspiel aus YOUNG (1985: 66) mit den Kosten $C = (15, 20, 55, 35, 61, 65, 78)$. für die drei Städte 1,2 und 3. Hieraus lassen sich die in Formel 101 beschriebenen Koalitionswerte ableiten.

Formel 101: Spielvektor für das Beispiel zum Per Capita Nucleolus

Werte der Einerkoalitionen:

$$v(1) = v(2) = v(3) = 0$$

Werte der Zweierkoalitionen:

$$v(1,2) = 0;\ v(1,3) = 9$$
$$v(2,3) = 10$$

Wert der Großen Koalition:

$$v(1,2,3) = 12$$

Für dieses Beispiel wird in Codebsp. 13 der Per Capita Nucleolus (siehe Zeile 2) berechnet. Hier zeigt sich, dass der berechnete Lösungswert mit $(2/3, 3/2, 19/2)$ (siehe Zeile 3) dem von YOUNG (1985: 68) entspricht, aber die Funktion `NucleolusCapita` aus dem Paket `GameTheory` von CANO-BERLANGA et al. (2015) mit $(4,4,4)$ (siehe Zeile 4-8) einen falschen Wert liefert.

```
> A<-generateGameVector(cFuncCostSharing,n=3,C=c(15,20,55,35,61,65,78))          #1
> perCapitaNucleolus(A, enableTermOutLP = FALSE)                                 #2
[1] 0.6666667 1.1666667 10.1666667                                               #3
> game<-GameTheory::DefineGame(n=3,V=A)                                          #4
> GameTheoryPCN<-GameTheory::NucleolusCapita(game)$nucleolus[,"x(S)"]            #5
> GameTheoryPCN                                                                  #6
S S S                                                                            #7
4 4 4                                                                            #8
```

Codebsp. 13: Per Capita Nucleolus Berechnung

Der Fehler im Paket `GameTheory` für die Funktion `NucleolusCapita` beruht hierbei auf der falschen Aufstellung des initialen Linearen Programms für den Per Capita Nucleolus.

4.2.4 Modified Nucleolus bzw. Modiclus

Beim von SUDHÖLTER (1996: 734f) eingeführten Modified Nucleolus bzw. Modiclus handelt es sich um eine Abwandlung des Nucleolus, die nicht versucht, den maximalen Wert für die Überschüsse, sondern derer Differenzen lexikographisch zu minimieren.

Diese Variante zielt darauf ab, alle Koalitionen gleichermaßen zu berücksichtigen und betrachtet daher sowohl deren Koalitionswert als auch deren Blockademacht.

4.2.4.1 Definition des Modified Nucleolus bzw. Modiclus

Die Differenz der Überschüsse wird zwischen jeder Koalition S und allen anderen Koalitionen T mit $S, T \subset N \wedge S \backslash T = \emptyset$ wie in Formel 102 spezifiziert. Die folgenden Definitionen sind hierbei SUDHÖLTER (1997:152) entnommen, mit der Änderung, dass für den allgemein definierten Auszahlungsraum die Imputationsmenge $I(N, v)$ festgelegt wurde.

Formel 102: Definition des Überschussdifferenzvektors für alle Koalitionen S und T

$$Sei\ S,T \subset N\ \wedge\ S \not\equiv T\ so\ gilt:$$

$$\widetilde{\Theta}(x,v) = \vartheta\left(\left(e(S,x,v) - e(T,x,v)\right)\right)$$

Auf Grundlage der Definition der Differenz von Überschüssen lässt sich der Modiclus als Imputation x definieren, welche die maximalen Differenzen der Überschüsse lexikographisch minimiert.

Formel 103: Definition des Modified Nucleolus bzw. des Modiclus

$$\Psi(N,v) = \{x \in I(N,v) | \widetilde{\Theta}(x,v) \preccurlyeq_{lex} \widetilde{\Theta}(y,v)\ für\ alle\ y \in I(N,v)\}$$

Hieraus ergibt sich für den Modiclus folgendes intiales Lineares Programm.

Formel 104 Definition des initialen Linearen Programms zur Berechnung des Modiclus

$$Minimiere:$$

$$z = F(x_1, \dots, x_n, t) = t$$

$$Unter\ Einhaltung\ der\ Nebenbedingungen:$$

$$(1)\ \sum_{i \in S} x_i - \sum_{i \in T} x_i + t \geq v(S), \forall S,T \subset N\ \wedge S \not\equiv T$$

$$(2)\ \sum_{i \in N} x_i = v(N)$$

$$(3)\ x_i \geq v(i), \forall i \in N$$

Im Vergleich zum Nucleolus hat sich die Anzahl der Nebenbedingungen erheblich von $2^n - 1$ auf $(2^n - 2) * (2^n - 3) + 1$ erhöht. Der Faktor $(2^n - 2)$ ergibt sich hier aus der Anzahl möglicher Koalitionen S und $(2^n - 3)$ aus der Anzahl für T mögliche Koalitionen. Demnach gibt es neben einer Restriktion für die Effizienz, $(2^n - 2) * (2^n - 3)$ weitere Bedingungen bezüglich der Differenzen aus den Überschüssen.

4.2.4.2 Beispiel für Modiclus

Als Beispiel dient das folgende Handschuhspiel aus Sudhölter (1997:148). In diesem Spiel wird für ein Handschuhpaar 1 € erzielt. Die einzelnen linken und rechten Handschuhe sind jedoch auf die einzelnen Spieler verteilt, von denen jeder nur einen einzigen Handschuh hält, der für sich gesehen ohne den passenden zweiten wertlos ist.

Bei drei Spielern verteilen sich die linken Handschuhe L auf Spieler 2_L und 3_L und Spieler 1_R erhält einen rechten Handschuh.

Spieler 1_R ist in diesem Spiel ein sogenannter Vetospieler, kann schließlich ohne ihn kein Wert gemeinsam erwirtschaftet werden.

Dem trägt der Nucleolus Sorge und bestimmt als Aufteilung $(1,0,0)$. Die Spieler 2_L und 3_L werden hier als machtlos betrachtet, sind sie doch durch einander austauschbar.

Schließen sich aber Spieler 2_L und 3_L gegen Spieler 1_R zusammen, können Sie diesen als Syndikat jedoch blockieren.

Dieser Blockademacht der Spieler 2_L und 3_L trägt der Modiclus Rechnung und sieht 2_L und 3_L im Zusammenschluss durch die Blockadermöglichkeit als gleich mächtig wie Spieler 1_R an.

Die Aufteilung nach dem Modiclus lautet daher $(1/2, 1/4, 1/4)$; dies wird auch durch die Ergebnisse des Pakets `CoopGame` bestätigt (siehe Codebsp. 13).

```
> A<-c(0,0,0,1,1,0,1)                                    #1
> nucleolus(A, enableTermOutLP = FALSE)                  #2
[1] 1 0 0                                                #3
> modiclus(A, enableTermOutLP = FALSE)                   #4
[1] 0.50 0.25 0.25                                       #5
```

Codebsp. 14: Berechnung des Beispiels für den Modiclus

Für eine Vielzahl von Testfällen für den Modiclus, sei darüber hinaus auf PELEG & SUDHÖLTER (2007:128) verwiesen. Auf der Grundlage eines parametrisierten Marktspiels lassen sich unendlich viele Testspiele bestimmen. Das Spiel ist durch die Koalitionsfunktion $v(S) = min\{|S \cap P|, a|S \cap Q|\} for\ all\ S \subseteq N$ und $N = P \cup Q, P = \{1,2\}, Q = \{3,4,5\}, a \geq 0$ bestimmt.

Alle Lösungen zu diesem Spiel ergeben sich aus der Formel $\frac{1}{12}(3v(N), 3v(N), 2v(N), 2v(N), 2v(N))$, die aber in PELEG & SUDHÖLTER (2007:128) mit einem falschen Vorfaktor von $\frac{1}{6}$ aufgeführt ist (schriftlich bestätigt von SUDHÖLTER auf Anfrage).

4.2.5 Simplified Modified Nucleolus bzw. Simplified Modiclus

Beim Simplified Modified bzw. dem Simplified Modiclus handelt es sich um ein Lösungkonzept von TARASHNINA (2011), das in Anlehnung an den Modiclus sowohl den Koalitionswert als auch die Blockademacht berücksichtigt.

Da der Modiclus jedoch mit sehr hohem Rechenaufwand verbunden ist, reagiert dessen Simplified Variante darauf, indem sie nicht mehr alle Überschussdifferenzen aller Koalitionen miteinander vergleicht, sondern jeweils nur für jede Koalition S mit deren Komplementkoalition $N\backslash S$ TARASHNINA (2011: 151).

Außerdem berücksichtigt der Modiclus zwar die Blockademacht einer Koalition, nicht aber, in welchem Verhältnis die konstruktiven und die blockierenden Kräfte insgesamt in der Lösung enthalten sind TARASHNINA (2011: 151). Der Simplified Modiclus entgegnet diesem

Schwachpunkt des Modiclus, indem er konstruktive und blockierende Kräfte gleichermaßen miteinfließen lässt TARASHNINA (2011: 155).

Der Simplified Modiclus ist eine Umsetzung, die die Blockademacht einer Koalition berücksichtigt und gleichzeitig den Aufwand der Berechnung deutlich senkt.

4.2.5.1 Definition Simplified Modiclus

Wurde beim Modiclus die Überschussdifferenz noch zwischen jeder Koalition S und allen anderen Koalitionen T mit $S,T \subset N \ \wedge \ S\backslash T = \emptyset$ berücksichtigt (siehe Formel 102), betrachtet die Simplified Variante nur noch die Überschussdifferenzen, die sich jeweils für eine Koalition S und der zugehörigen Komplementärkoalition ergeben (siehe Formel 105). Die folgenden Definitionen richten sich nach TARASHNINA (2011: 155) und wurden zur besseren Vergleichbarkeit, in den Notationsstil des Modiclus von SUDHÖLTER (1997:152) übertragen.

Formel 105: Definition des Überschussdifferenzvektors $\tilde{P}(x,v)$ für alle Koalition S und deren Komplementkoalition

$$Sei\ S \subseteq N\ so\ gilt:$$
$$\tilde{\mathrm{P}}(x,v) = \varrho\big(e(S,x,v) - e(N\backslash S,x,v)\big) = \varrho(\bar{e}(S,x,v))$$

Auf Grundlage der Definition des Überschussdifferenzvektors aller Koalitionen S und deren Komplementärkoalitionen, lässt sich der Simplified Modiclus $\Upsilon(N,v)$ als Imputation x definieren, welche die maximalen Differenzen der Überschüsse lexikographisch minimiert.

Formel 106: Definition des Simplified Modified Nucleolus bzw. Simplified Modiclus

$$\Upsilon(N,v) = \{x \in I(N,v) | \tilde{\mathrm{P}}(x,v) \preccurlyeq_{lex} \tilde{\mathrm{P}}(y,v)\ f\ddot{u}r\ alle\ y \in I(N,v)\}$$

Für die Aufstellung des Linearen Programm wird die Überschussdifferenz $\bar{e}(S,x,v)$ wie in Formel 107 umgeformt.

Formel 107: Umformung der Überschussdifferenz $\bar{e}(S,x,v)$ als Ausgangspunkt für das initiale Lineare Programm des Simplified Modiclus

$$(1)\ \bar{e}(S,x,v) = e(S,x,v) - e(N\backslash S,x,v)$$
$$(2)\ \bar{e}(S,x,v) = [v(S) - x(S)] - [v(N\backslash S) - x(N\backslash S)]$$
$$(3)\ \bar{e}(S,x,v) = [v(S) - v(N\backslash S)] - [x(S) - x(N\backslash S)]$$
$$\Rightarrow [x(S) - x(N\backslash S)] + \bar{e}(S,x,v) = [v(S) - v(N\backslash S)]$$

Unter Berücksichtigung der Umformung aus Formel 107 kann das initiale Lineare Programm zur Berechnung des Simplified Modiclus aufgestellt werden.

Formel 108 Definition des initialen Lineare Programms für die Berechnung des Simplified Modiclus

$Minimiere$:

$$z = F(x_1, \dots, x_n, t) = t$$

$Unter\ Einhaltung\ der\ Nebenbedingungen$:

$$(1)\ \sum_{i \in S} x_i - \sum_{i \in (N \setminus S)} x_i + t \geq v(S) - v(N \setminus S), \forall S \subseteq N$$

$$(2)\ \sum_{i \in N} x_i = v(N)$$

$$(3)\ x_i \geq v(i), \forall i \in N$$

4.2.5.2 Beispiel für Simplified Modiclus

Erinnert sei nochmals an das Handschuhspiel aus SUDHÖLTER (1997:148), das bereits als Beispiel für den Modiclus herangezogen wurde (siehe Abschnitt 4.2.4.2).

Hier waren auf drei Spieler zwei linke und ein rechter Handschuh verteilt.

Dabei waren die linken Handschuhe L im Besitz von Spieler 2_L und 3_L und der rechte von Spieler 1_R.

Wie bereits bekannt, berücksichtigt der Nucleolus die Blockademacht von dem Syndikat der Spieler 2_L und 3_L nicht und bestimmt als Aufteilung $(1{,}0{,}0)$.

Der Blockademacht der Spieler 2_L und 3_L trägt dagegen der Modiclus Rechnung und sieht 2_L und 3_L im Zusammenschluss durch die Blockademöglichkeit als gleich mächtig wie Spieler 1_R an, was die intuitive Aufteilung $(1/2\,,1/4\,,1/4)$ zur Folge hat.

Ein Mittelweg zwischen der Berücksichtigung der Blockademacht und hohem Berechnungsaufwand auf der anderen Seite schlägt der Simplified Modiclus ein.

Der Simplified Modiclus berücksichtigt nicht alle Überschussdifferenzen und kommt so auf das Ergebnis $(2/3\,,1/6\,,1/6)$.

```
> A<-c(0,0,0,1,1,0,1)                                    #1
> nucleolus(A, enableTermOutLP = FALSE)                  #2
[1] 1 0 0                                                #3
> modiclus(A, enableTermOutLP = FALSE)                   #4
[1] 0.50 0.25 0.25                                       #5
> simplifiedModiclus(A, enableTermOutLP = FALSE)         #6
[1] 0.6666667 0.1666667 0.1666667                        #7
```

Codebsp. 15: Berechnung Simplified Modiclus

4.2.6 Disruption Nucleolus

Der von LITTLECHILD & VAIDYA (1976) vorgestellte Disruption Nucleolus ist ein Lösungskonzept, welches das in Abschnitt 4.1.5 behandelte Maß für Störanfälligkeit (engl. *Propensity To Disrupt*) des Gately Punkts aufgreift, weiter verallgemeinert und auf das Nucleolus Konzept überträgt. Hierdurch ist dem Disruption Nucleolus möglich, sowohl die Vorzüge des Gately Konzepts zu nutzen, als auch sich von dessen Schwächen zu befreien.

Ein Schwachpunkt des Gately Punktes ist es nämlich, dass dieser für die Bestimmung der zu minimierenden *Propensities To Disrupt* eines jeden Spieler nur die Koalitionswerte der Einer- und Komplementkoalitionen sowie den der großen Koalition betrachtet und andere Koalitionswerte hierbei unbeachtet lässt. Auf diesen Makel wurde bereits im Abschnitt 4.1.5.4 bei der Konstruktion eines nichteindeutigen Spielevektors für das Gately Konzept eingegangen. Hier wurde aufgezeigt, dass in einem Vierpersonenspiel die Belegung der Werte der Zweierkoalitionen unerheblich für das spätere Ergebnis ist.

Der Disruption Nucleolus beschränkt sich aus diesem Grund bei der Minimierung der maximalen *Propensities To Disrupt* nicht ausschließlich auf die der Einerkoalitionen, sondern minimiert lexikographisch den Vektor mit den *Propensities To Disrupt* für alle Koalitionen (LITTLECHILD & VAIDYA 1976: 151).

4.2.6.1 Definition des Disruption Nucleolus

Um die von GATELY (1974) eingeführte *Propensity To Disrupt* als Maß für die Störanfälligkeit beliebiger Koalitionen nutzen zu können, führten LITTLECHILD & VAIDYA (1976: 152) eine verallgemeinerte Variante ein (siehe Formel 109), die das Maß auf jede Koalition anwendbar macht.

Formel 109: Definition der Propensity To Disrupt im allgemeinen Fall

$$d(S,x) = \frac{x(N-S) - v(N-S)}{x(S) - v(S)}$$

Auf Grundlage der allgemeinen Form der *Propensity To Disrupt* stellten LITTLECHILD & VAIDYA (1976: 154) ein erstes Programm zur Bestimmung des Disruption Nucleolus auf (siehe Formel 110).

Formel 110: Definition des initialen Programms zur Berechnung des Disruption Nucleolus

$$Minimiere:$$

$$z = F(x_1, \dots, x_n, t) = r$$

$$Unter\ Einhaltung\ der\ Nebenbedingungen:$$

$$(1)\ d(S, x) \leq r, \forall S \subset N$$

$$(2)\ \sum_{i \in N} x_i = v(N)$$

$$(3)\ x_i \geq 0, \forall i \in N$$

Das Lineare Programm aus Formel 110 konnten LITTLECHILD & VAIDYA (1976: 155) durch Umformungen und Substituierungen in das Nichtlineare Programm aus Formel 111 überführen

Formel 111: Definition des initialen Nichtlinearen Programmes zur Berechnung des Disruption Nucleolus

$$Minimiere:$$

$$z = F(x_1, \dots, x_n, t) = r$$

$$Unter\ Einhaltung\ der\ Nebenbedingungen:$$

$$(1)\ \sum_{i \in S} x_i - [v(N) - v(S) - v(N - S)] \frac{1}{1 + r} \geq v(S), \forall S \subset N$$

$$(2)\ \sum_{i \in N} x_i = v(N)$$

$$(3)\ x_i \geq 0, \forall i \in N$$

Durch den Umstand, dass für das Spiel ein strenger Kern gefordert wird, lässt sich das nichtlineare Programm aus Formel 111 durch eine einfache Substitution in ein Lineares Programm (siehe Formel 112) konvertieren. LITTLECHILD & VAIDYA (1976: 155) führten hierzu eine Substituierung von dem nichtlinearen Bestandteil $\frac{-1}{1+r}$ durch t aus. Anstatt der Entscheidungsvariable r wird außerdem t minimiert.

Eine Rückführung von t in r kann dabei durch die Resubstituierung in Form von $r = \frac{-1}{t} - 1$ erfolgen.

Die Entscheidungsvariable t ist hierbei nach unten und oben strikt durch -1 und 0 beschränkt, so dass $-1 < t < 0$ gilt.

Hieraus ergibt sich folgendes Lineare Programm, dessen Minimierung von t sich äquivalent zu der von r verhält.

Formel 112 Definition des initialen Linearen Programmes zur Berechnung des Disruption Nucleolus

$Minimiere$:

$$z = F(x_1, \dots, x_n, t) = t$$

$Unter\ Einhaltung\ der\ Nebenbedingungen$:

$$(1)\ \sum_{i \in S} x_i + [v(N) - v(S) - v(N-S)]t \geq v(S), \forall S \subset N$$

$$(2)\ \sum_{i \in N} x_i = v(N)$$

$$(3)\ x_i \geq 0, \forall i \in N$$

Auf Basis des initialen Linearen Programms aus Formel 112 und des allgemeinen Lösungsalgorithmus des Nucleolus lässt sich der Auszahlungsvektor bestimmen, der die maximalen *Propensities To Disrupt* lexikographisch für alle Koalitionen optimal minimiert.

4.2.6.2 Beispiel für Disruption Nucleolus

Das folgende Beispiel ist aus LITTLECHILD & VAIDYA (1976: 153) entnommen und spezifiziert ein Vierpersonenspiel wie in Formel 113 beschrieben.

Formel 113: Spielevektor für das Beispiel zum Disruption Nucleolus

Werte der Einerkoalitonen:

$$v(1) = v(2) = v(3) = v(4) = 0$$

Werte der Zweierkoalitionen:

$$v(1,2) = 2; v(1,3) = 3; v(1,4) = 4$$
$$v(2,3) = 1;\ v(2,4) = 3$$
$$v(3,4) = 2$$

Werte der Dreierkoalitionen:

$$v(1,2,3) = 8;\ v(1,2,4) = 11;\ v(1,3,4) = 6.5$$
$$v(2,3,4) = 9.5$$

Wert der Großen Koalition:

$$v(1,2,3,4) = 14$$

Die von LITTLECHILD & VAIDYA (1976: 156) angegebene Lösung für den Disruption Nucleolus kann jedoch als falsch angesehen werden, da mit einem Lösungswert von $x^L = (3.19, 5.26, 2.13, 3.42)$ die maximalen *Propensities To Disrupt* $\Delta(x^L, v)$ nicht lexikographisch für alle Koalitionen optimal minimiert werden. Im Folgenden werden die Ergebnisse auf zwei Stellen gerundet angegeben, um der Genauigkeit des Lösungswert von LITTLECHILD & VAIDYA (1976: 156) zu entsprechen.

Formel 114: Lexikographisch geordneter Vektor mit Propensities To Disrupt zum Disruption Nucleolus Beispiel

$$\Delta(x^L, v) = (2.45, 2.45, 2.45, 2.44, 2.35, 1.82, 1.33, 0.75, 0.55, 0.43, 0.41, 0.41, 0.41, 0.41)$$

Für den durch das Paket `CoopGame` berechnete Disruption Nucleolus $x^{CG} = (3.19, 4.75, 2.13, 3.92)$ ergibt sich ein optimalerer Vektor für die *Propensities To Disrupt* $\Delta(x^{CG}, v)$.

Formel 115: Lexikographisch geordneter Vektor mit Propensities To Disrupt zum Disruption Nucleolus Beispiel

$$\Delta(x^{CG}, v) = (2.44, 2.44, 2.44, 1.89, 1.89, 1.73, 1.47, 0.68, 0.58, 0.53, 0.53, 0.41, 0.41, 0.41)$$

Wird die Differenz zwischen der Lösung für den Disruption Nucleolus x^{CG} aus `CoopGame` und dem Lösungswert x^L gebildet, fällt ins Auge, dass bei LITTLECHILD & VAIDYA (1976: 156) Spieler 2 einen um 0.50 höheren und Spieler 4 einen entsprechend niedrigeren Wert erhält.

Tabelle 15 bildet einen Vergleich zwischen den Werten für die *Propensities To Disrupt* entsprechend der Lösung von Littlechild & VAIDYA (1976: 156) x^L und der aus `CoopGame` x^{CG}. Hier werden größere Unterschiede ersichtlich.

Tabelle 15: Vergleich der Propensities To Disrupt entsprechend der Lösung von Littlechild & VAIDYA 1976 ggü. der von `CoopGame`

Index	**Bit-Zeile für S**	**Spieler $\in S$**	$v(S)$	$d(x^L, S)$	$d(x^{CG}, S)$	$\Delta(x^L, v)$	$\Delta(x^{CG}, v)$	$\Delta(x^L, v) - \Delta(x^{CG}, v)$
1	1 0 0 0	{1}	0	0.41	0.41	2.45	2.44	0.01
2	0 1 0 0	{2}	0	0.43	0.58	2.45	2.44	0.01
3	0 0 1 0	{3}	0	0.41	0.41	2.45	2.44	0.01
4	0 0 0 1	{4}	0	0.75	0.53	2.44	1.89	0.55
5	1 1 0 0	{1,2}	2	0.55	0.68	2.35	1.89	0.46
6	1 0 1 0	{**1**, **3**}	3	**2.45**	**2.44**	1.82	1.73	0.09

7	1	0	0	1	$\{\mathbf{1},\mathbf{4}\}$	4	$\mathbf{2.45}$	$\mathbf{1.89}$	1.33	1.47	-0.14
8	0	1	1	0	{2,3}	1	0.41	0.53	0.75	0.68	0.07
9	0	1	0	1	{2,4}	3	0.41	0.41	0.55	0.58	-0.03
10	0	0	1	1	{3,4}	2	1.82	1.47	0.43	0.53	-0.10
11	1	1	1	0	{1,2,3}	8	1.33	1.89	0.41	0.53	-0.12
12	1	1	0	1	$\{\mathbf{1},\mathbf{2},\mathbf{4}\}$	11	$\mathbf{2.45}$	$\mathbf{2.44}$	0.41	0.41	0.00
13	1	0	1	1	{1,3,4}	6.5	$\mathbf{2.35}$	$\mathbf{1.73}$	0.41	0.41	0.00
14	0	1	1	1	$\{\mathbf{2},\mathbf{3},\mathbf{4}\}$	9.5	$\mathbf{2.44}$	$\mathbf{2.44}$	0.41	0.41	0.00
15	1	1	1	1	{1,2,3,4}	14	–	–	–	–	–
Summe							18.66	17.85	18.66	17.85	0.81

Der Angabe von Littlechild & VAIDYA (1976: 156), wonach für die Koalitionen $\{\mathbf{1},\mathbf{3}\},\{\mathbf{1},\mathbf{4}\},\{\mathbf{1},\mathbf{2},\mathbf{4}\},\{\mathbf{1},\mathbf{3},\mathbf{4}\}$ und $\{\mathbf{2},\mathbf{3},\mathbf{4}\}$ die maximale *Propensity To Disrupt* bei $\mathbf{2.45}$ liegt, ließ sich durch das Paket `CoopGame` sowohl für die Auszahlungen x^L als auch x^{CG} widerlegen.

```
#Game example out of LITTLECHILD&VAIDYA (1975: 153)                    #1
A=c(0,0,0,0,2,3,4,1,3,2,8,11,6.5,9.5,14)                              #2
                                                                       #3
#Game solution for Disruption Nucleolus                                #4
#according to LITTLECHILD&VAIDYA (1975: 156)                           #5
expected_x=c(3.19,5.26,2.13,3.42)                                      #6
                                                                       #7
#Calculate Disruption Nucleolus by CoopGame                            #8
cdn=disruptionNucleolus(A, enableTermOutLP = F)                        #9
                                                                       #10
#create 4 player bit matrix for utility reasons                        #11
bmDN=createBitMatrix(4,A)                                              #12
                                                                       #13
#Determine Propensities To Disrupt for all coalitions                  #14
#according to Disruption Nucleolus claimed                             #15
#by CoopGame                                                           #16
ptd_own=sapply(                                                        #17
        1:14,                                                          #18
        function(ix){                                                  #19
                S=getPlayersFromBMRow(bmDN[ix,])                       #20
                propensityToDisrupt(bmDN[,"cVal"],cdn,S=S)             #21
        }                                                              #22
)                                                                      #23
                                                                       #24
#Determine Propensities To Disrupt for all coalitions                  #25
#according to Disruption Nucleolus claimed                             #26
#by LITTLECHILD&VAIDYA (1975: 156)                                     #27
ptd_littlechild=sapply(                                                #28
        1:14,                                                          #29
        function(ix){                                                  #30
                S=getPlayersFromBMRow(bmDN[ix,])                       #31
                propensityToDisrupt(bmDN[,"cVal"],x=expected_x,S=S)    #32
        }                                                              #33
)                                                                      #34
                                                                       #35
#Sort Propensities To Disrupt decreasingly                             #36
#for vector retrieved by solution of CoopGame                          #37
ptd_own_sorted=sort(ptd_own,decreasing = TRUE)                         #38
                                                                       #39
#Sort Propensities To Disrupt decreasingly                             #40
#for vector retrieved by solution of LITTLECHILD&VAIDYA (1975: 156)     #41
ptd_littlechild_sorted=sort(ptd_littlechild,decreasing = TRUE)         #42
                                                                       #43
#Determine Vector showing differences of sorted Propensities To Disrupt #44
#according to LITTLECHILD&VAIDYA (1975: 156) and those of CoopGame      #45
diffs_ptd_sorted=ptd_littlechild_sorted - ptd_own_sorted               #46
                                                                       #47
#Determine maximal Propensity To Disrupt                               #48
#for vector retrieved by solution of LITTLECHILD&VAIDYA (1975: 156)     #49
max_ptd_littlechild=max(ptd_littlechild)                               #50
                                                                       #51
#Determine maximal Propensity To Disrupt                               #52
#for vector retrieved by solution of LITTLECHILD&VAIDYA (1975: 156)     #53
max_ptd_own=max(ptd_own)                                               #54
                                                                       #55
#Compare maximal Propensity To Disrupt                                 #56
#of CoopGame and LITTLECHILD&VAIDYA (1975: 156)                        #57
if(max_ptd_own<max_ptd_littlechild){                                   #58
        print("Maximal Propensity to Disrupt of CoopGame is lower!“)   #59
}                                                                      #60
```

Codebsp. 16: Widerlegung Disruption Nucleolus Beispiel von Littlechild & Vaidya (1976)

4.2.7 Lexical Gately Punkt

Mit der Widerlegung der von GATELY (1974) implizierte Eindeutigkeit für den Gately Punkt, reifte im Verlauf der Arbeit zunehmend die Überzeugung, dass das Konzept als punktwertiges Lösungsansatz in der von GATELY (1974) vorgestellten Form nicht mehr zuverlässig zu gebrauchen war.

Es müssen nämlich bei Verwendung des Konzepts neben einer regulären punktwertigen Lösung, immer zwei Sonderfälle berücksichtigt werden.

Zum einen kann der Fall eintreten, dass überhaupt keine Lösung existiert, zum anderen kann aber auch die gesamte Imputationsmenge den Anforderungskriterien des Gately Punkts entsprechen.

Diese Sonderfälle erschweren daher in hohem Maße die Anwendbarkeit des Lösungsverfahrens. Dies gab im Rahmen der Masterarbeit den Ausschlag auf die Nichteindeutigkeit des Gately Konzepts mit einer eigenen Abwandlung, dem hier vorgestellten Lexical Gately Punkt zu reagieren.

In Anlehnung an den Disruption Nucleolus, der seinerzeit selbst aus dem Gately Punkt heraus entstanden ist (LITTLECHILD & VAIDYA 1976: 1), werden beim Lexical Gately Punkt die maximalen *Propensities To Disrupt* bezüglich der Einerkoalition lexikographisch minimiert.

Das Konzept des Lexical Gately Punkts ist aber noch nicht komplett ausgereift, da es bei Nichteindeutigkeit des Gately Punkts lediglich, je nach verwendetem LP-Solver, einen zufälligen Punkt der Imputationsmenge als eindeutige Lösung liefert. Der vorgestellte Lexical Gately Punkt soll hier aber in erster Linie sensibilisieren und als Diskussionsgrundlage zur Spezifikation weiterer Nebenbedingung dienen, die nur im nichteindeutigen Fall greifen und dabei helfen, einen charakteristischen Punkt der $I(N, v)$ zu bestimmen.

4.2.7.1 Definition Lexical Gately Punkt

Beim Disruption Nucleolus wurden für alle Koalitionen die maximalen *Propensities To Disrupt* lexikographisch minimiert, für den Lexical Gately Punkt lediglich die der Einerkoalitionen. Dementsprechend reicht es zur Bestimmung des Lexical Gately Punktes aus, den Disruption Nucleolus geringfügig bezüglich des initialen Linearen Programmes (siehe Formel 116) zu modifizieren.

Statt wie beim Disruption Nucleolus für alle Koalitionen S die Nebenbedingung zu deren *Propensity To Disrupt* aufzustellen, werden beim Lexical Gately Punkt hierzu nur noch Restriktionen für die Einerkoalitionen spezifiziert.

Formel 116: Definition des intialen Linearen Programms zur Berechnung des Lexical Gately Punkt

$Minimiere$:

$$z = F(x_1, \dots, x_n, t) = t$$

$Unter\ Einhaltung\ der\ Nebenbedingungen$:

$$(1)\ \sum_{i \in S} x_i + [v(N) - v(S) - v(N-S)]t \geq v(S), \forall S \subset N \wedge card(S) = 1$$

$$(2)\ \sum_{i \in N} x_i = v(N)$$

$$(3)\ x_i \geq 0, \forall i \in N$$

4.2.7.2 Beispiel für Lexical Gately Punkt

Für das Lexical Gately Punkt Beispiel wurde ein für das Gately Punkt nichteindeutiger Spielevektor für ein Dreipersonenspiel mit einer *Equal Propensity To Disrupt* von minus 1 konstruiert (siehe Abschnitt 4.1.5.4).

Formel 117 Spielevektor für ein Beispiel zum Lexical Gately Punkt mit einer Equal Propensity To Disrupt von -1

Werte der Einerkoalitonen:

$$v(1) = 3; v(2) = 2; v(3) = 4$$

Werte der Zweierkoalitionen:

$$v(1,2) = 6; v(1,3) = 8$$
$$v(2,3) = 7$$

Werte der Großen Koalition:

$$v(1,2,3) = 10$$

Wie bereits aus Abschnitt 4.1.5.3 bekannt, erfüllt bei einem solch gearteten Spiel die gesamte Imputationsmenge mit den Eckpunkten (3,3,4), (4,2,4) und (3,2,5) die erforderlichen Kriterien des Gately Punkts.

Der Lexical Gately Punkt liefert in diesem Fall den eindeutigen Punkt (4,2,4), welcher einem der Eckpunkte der Imputationsmenge entspricht. Dieser Punkt ist zwar überaus fragwürdig, aber zumindest eindeutig. Auf Grundlage des jetzigen Lexical Gately, gilt es daher weitere Nebenbedingungen zu spezifizieren, welche eine zufriedenstellendere Lösung bereitstellen. So wäre es beispielsweise denkbar, in einem nichteindeutigen Fall den Schwerpunkt der Im-

putationsmenge wie bei der Gately Punkt Implementierung als charakteristischen Lösungswert bestimmen zu lassen (STAUDACHER 2017).

Codebsp. 17 zeigt in Zeile die Berechnung des Beispiels zum Lexical Gately Punkt. Der nichteindeutige Spielvektor lässt sich entsprechend mit `getNondefiniteGameVector4GatelyValue` konstruieren (siehe Zeile 1). Wie der Aufruf von `imputatonsetVertices` nahelegt, entspricht die durch die Funktion `lexicalGatelyValue` bestimmte Lösung (19, 2, 4, 5) einem der Eckpunkte der Imputationsmenge (siehe Zeile 13).

```
> A=getNondefiniteGameVector4GatelyValue(30,c(3,2,4,5),w = rep(99,6))          #1
> lexicalGatelyValue(A)                                                       #2
GLPK Simplex Optimizer, v4.47                                                 #3
5 rows, 5 columns, 8 non-zeros                                                #4
      0: obj =  0.000000000e+000  infeas = 1.600e+001 (1)                     #5
*     1: obj =  0.000000000e+000  infeas = 0.000e+000 (0)                     #6
PROBLEM HAS UNBOUNDED SOLUTION                                                #7
[1] 19  2  4  5                                                              #8
> imputationsetVertices(A)                                                    #9
     [,1] [,2] [,3] [,4]                                                      #10
[1,]   3   18    4    5                                                      #11
[2,]   3    2   20    5                                                      #12
[3,]  19    2    4    5                                                      #13
[4,]   3    2    4   21                                                      #14
```

Codebsp. 17: Berechnung des Beispiels zum Lexical Gately Punkt

5 Implementierung von Nucleolus Varianten

Das folgende Kapitel thematisiert einen wesentlichen Bestandteil der Arbeit: die Konzeption, Implementierung und Nutzung eines eigens erstellten Frameworks für die Berechnung der verschiedenen Nucleolus Derivate auf der Grundlage eines objektorientierten Ansatzes in R mit S4 Klassen. Hierfür sollte im vorhergehenden Kapitel 4 zunächst ein grundlegendes Verständnis über die verschiedenen mit dem Nucleolus verwandten Lösungskonzepte vermittelt werden, um die Beweggründe besser nachvollziehen zu können, die im Rahmen der praktischen Umsetzung getroffen wurden. Ein weiterer wesentlicher Hauptpunkt dieses Kapitels ist dabei die Beschreibung der Ausgangslage, die mit der Bachelorarbeit von GEBELE (2016) und dessen Implementierung des Nucleolus und Prenucleolus bereits vorlag.

Eine Kurzeinführung in Möglichkeiten für Objektorientierung in R dient einmal mehr dazu, die Vorteile und auch Grenzen gegenüber einem rein prozeduralen Ansatz zu verdeutlichen.

Da der Algorithmus zur Berechnung des Nucleolus und dessen Derivaten auf einer Reihe von Linearen Programmen beruht, wird bei den Erläuterungen zur Konzeption des eigentlichen Frameworks auch auf Lineare Programme im Allgemeinen und die genutzte C-Library `GLPK` (THE GNU PROJECT 2017) sowie das R-Packet `glpkAPI` (FRITZEMEIER ET. AL. 2015) eingegangen.

Abschließend befasst sich dieses Kapitel mit den verschiedenen Implementierungen im Einzelnen. Dabei wird aufgezeigt, in welchen Punkten sich die jeweiligen Umsetzungen voneinander unterscheiden.

5.1 Ausgangslage

Mit den Konzepten des Nucleolus und Prenucleolus waren bereits zwei punktwertige Lösungskonzepte im Paket `CoopGame` enthalten. Diese gingen aus der Bachelorarbeit von GEBELE (2016) hervor und sollten im Rahmen dieser Masterarbeit um die Berechnung weiterer Nucleolus Derivate ergänzt werden.

Da die anfängliche Annahme, dass die verschiedenen Varianten sich lediglich bezüglich der initialen Zuweisung der zu minimierenden Strukturvariable unterscheiden, sich bald als überholt erwies, war eine vollständige Reimplementierung unausweichlich.

So wurde unter Berücksichtigung des von MASCHLER et al. (2013: 815f) und PETERS (2015: 350-353) beschriebenen Berechnungsmethodik für den Nucleolus (siehe Abschnitt 4.2.1.2) – sowie der Ergebnisse von GEBELE (2016**)** – ein Rahmenwerk für eine flexiblere und vereinfachte Implementierung von Nucleolus Derivaten geschaffen und die alten Funktionen auf der Basis eines objektorientierten Ansatzes in R mit S4 Klassen weitestgehend überarbeitet.

Die bisherige Implementierung von GEBELE (2016) wurde unter der Zielsetzung reimplementiert, mit einer höheren Modularisierung der einzelnen Berechnungsschritte stärker auf etwaige und auch noch so kleine Unterschiede der verschiedenen Varianten eingehen zu können.

Es folgt ein kurzer Ausblick auf die Objektorientierung in R, die sich wesentlich anders als bei anderen Hochsprachen wie Java oder C# gestaltet und deren Besonderheiten im Folgenden behandelt werden.

5.2 Objektorientierung in R

R hat mit den S3 und S4 Classes sowie den Reference Classes R6 drei objektorientierte Ansätze.

Hierbei haben alle drei Systeme je nach Problemstellung ihre jeweilige Berechtigung. Trotz teils erheblicher Unterschiede ist ihnen allen das Konzept von Klassen und Methoden gemein. Die Klassen weisen zwischeneinander Beziehungen auf und je nachdem zeigen die Instanzen der Klassen unterschiedliches Verhalten.

Mit dem Konzept der Vererbung besteht die Möglichkeit, Verhalten der Elternklassen zu übernehmen aber auch gegebenenfalls zu überschreiben.

Für die praktische Umsetzung wurde das S3 und S4 System näher betrachtet.

5.2.1 Das S3-Konzept

Bei S3 handelt es sich um das erste und einfachste objektorientierte System in R, das durch seinen Minimalismus eine gewisse Eleganz versprüht (WICKHAM 2015).

Ein wesentlicher Kritikpunkt ist jedoch, dass es sich bei S3 nicht wirklich um ein formelles System für die Umsetzung der Objektorientierung in R handelt (BATES 2003). Die Repräsentation und Vererbung bei Klassen ist nicht formell definiert (WICKHAM 2015). Zudem ist das Testen auf S3 Zugehörigkeit von Objekten nur erschwert möglich. Soll beispielsweise geprüft werden, ob es sich bei einem Objekt `x` um ein S3 Objekt in R handelt, kann dies intern nur sehr umständlich mit `is.object(x) & !isS4(x)` geprüft werden. Abhilfe schafft hier die R Bibliothek `pryr` mit der Funktion `otype(x)` (WICKHAM 2015). Des Weiteren ist das S3 Konzept zwar sehr einfach zu verwenden, allerdings ist die Typsicherheit bei der Zuweisung der Komponenten für ein S3 Objekt nicht gewährleistet (BATES 2003). Einer Komponente bzw. einem Slot eines S3 Objekts kann so ohne weiteres ein Objekt oder Wert mit einem falschen Typ zugewiesen werden.

Auch die Auswahl der aufzurufenden Methode, der sogenannte Dispatch, kann nur anhand eines Arguments erfolgen.

Aufgrund der fehlenden Typsicherheit und der anderen oben aufgeführten Kritikpunkte wurde bis auf weiteres auf die Verwendung von S3 Klassen, zu Gunsten der im Anschluss thematisierten S4 Klassen, verzichtet.

5.2.2 Das S4-Konzept

Der große Vorteil des S4 gegenüber dem S3 Systems liegt in der formellen Definition von Repräsentation und Vererbung (WICKHAM 2015).

BATES (2003) bevorzugt daher bei neuen Projekten generell eine Nutzung des S4 Systems. Zum einen wird hier die Typsicherheit für die Komponenten einer Klasse gewährleistet, zum anderen unterstützt S4 auch den Aufruf der entsprechenden Methode nicht nur unter Berücksichtigung eines sondern einer beliebigen Anzahl von Argumenten, sprich Multiple Dispatch (WICKHAM 2015).

Darüber hinaus kann eine Validierungsfunktion mit angegeben werden, die bei Instanziierung von S4 Objekten die Korrektheit überprüft. So können beispielsweise Parameter, die bei der Objekterzeugung mit angegeben wurden, auf ihre Plausibilität hin überprüft werden.

Für einen raschen Einstieg in das S4-Konzept von R hat sich hierbei der Leitfaden von GENOLINI (2008) als sehr hilfreich erwiesen. Elementare Ansätze der Objektorientierung mit dem S4 System konnten – auf diesem aufbauend – schnell erfasst und praktisch umgesetzt werden.

Im Folgenden wird kurz auf die wesentlichen Bestandteile des S4 Systems und deren beispielhafte Verwendung in Anlehnung an GENOLINI (2008) eingegangen.

5.2.2.1 S4 Klassen

In dem S4 System werden bei der formellen Definition von Klassen (siehe Codebsp. 18) mit der Funktion `setClass()` der Klassenname (siehe Zeile 2), eine Liste mit typisierten Klassenattributen bzw. auch Slots genannt (siehe Zeile 3 bis 7), spezifiziert.

Sollen einige Slots bestimmte Default-Werte aufweisen, können diese mit der Funktion `prototype()` (siehe Zeile 8) bei der Klassendefinition spezifiziert werden. Durch Angabe einer Validierungsfunktion (siehe Zeile 9 bis 15) kann die Korrektheit eines erzeugten Objekts sichergestellt werden. Auf Klassenvererbung wird in Abschnitt 5.2.2.5 eingegangen.

```
setClass (                                                          #1
    Class= "ExampleClass",                                          #2
    representation(                                                 #3
        x="numeric",                                                #4
        y="character",                                              #5
        z="logical"                                                 #6
    ),                                                              #7
    prototype (x=1,y="Default Example String",z=TRUE),              #8
    validity=function(object){                                      #9
        if(object@x>=0){                                            #10
                return(TRUE)                                        #11
        }else{                                                      #12
                return(FALSE)                                       #13
        }                                                           #14
    }                                                               #15
)                                                                   #16
```

Codebsp. 18: ExampleClass

Codebsp. 19 zeigt die Objekterzeugung von der Klasse ExampleClass mit der Funktion `new()`, welcher der Name der zu erzeugenden Klasse als Literal übergeben wird (siehe Zeile 1 und 2). Bei dem Codebsp. 19 in Zeile 2 werden darüber hinaus noch Werte für die Slots mit übergeben; in Zeile 1 erfolgt eine Zuweisung anhand der mit `prototype()` in der Klassendefinition (siehe Zeile 8) angegebenen Defaultwerte.

Sollte für eine Klasse `prototype()` in der Klassendefinition (siehe Codebsp. 18 Zeile 8) nicht spezifiziert sein und erfolgt ein Aufruf wie in Codebsp. 19 in Zeile 1, werden die Slots alle mit `NULL` initialisiert.

```
> A1<-new(Class="ExampleClass")                                     #1
> A2 <- new(Class="ExampleClass",x=2,y="Non Default Example String",z=FALSE)   #2
```

Codebsp. 19: Objekterzeugung der ExampleClass

5.2.2.2 Konstruktor

Für die Objekterzeugung einer Klasse empfiehlt sich die Definition einer Wrapperfunktion (siehe Codebsp. 20), die als Art Konstruktor agiert. Entsprechend der übergebenen Argumente kann diese mit der Funktion `new()` ein passendes Objekt erzeugen und anschließend zurückgeben.

Der große Vorteil liegt hier für den Entwickler in der intuitiveren und einfacheren Objekterzeugung. Dieser muss hier nicht mehr den gesamten Klassennamen angeben, sondern kann die Vervollständigungsvorschläge von RStudio nutzen.

```
ExampleClass<-function(x=1,y="Default Example String",z=TRUE){          #1
        retExampleClass<-new(Class="ExampleClass",x=x,y=y,z=z)          #2
        return(retExampleClass)                                         #3
}
```

Codebsp. 20: Wrapperfunktion für Objekterzeugung

Wie in Codebsp. 19 werden in dem unten aufgeführten Codebsp. 21 zwei Objekte der Klasse `ExampleClass` generiert, nur dass in diesem Fall auf die oben spezifizierte Wrapperfunktion für die Objekterzeugung zurückgegriffen wird.

```
> A1<-ExampleClass()                                                    #1
> A2 <- ExampleClass (x=2,y="Non Default Example String",z=FALSE)       #2
```

Codebsp. 21: Objekterzeugung mit Wrapperfunktion

Es besteht die Möglichkeit für die Erzeugung komplexere Klassen, deren Initialisierung über die reine Zuweisung von Werten an die Slots hinausgeht, eine Initialisierungsmethode (siehe Codebsp. 22) zu spezifizieren. Dies ist beispielsweise unumgänglich, wenn es sich bei einigen Klassenkomponenten ebenfalls um Klassen handelt, die es auch vorab zu initialisieren gilt.

Die Definition der `initialize`-Methode – auf Methoden im Allgemeinen wird später in Abschnitt 5.2.2.3 näher eingegangen – erfolgt durch die Funktion `setMethod`. Hier wird in Zeile 2 der festdefinierte Methodenname `initialize` angegeben und zusammen mit Zeile 3 die entsprechende Klasse referenziert. Bei einer Objekterzeugung mit `new()` wird fortan diese Methode zur Initialisierung ausgeführt.

In dem folgenden Codebsp. 22 hängt der an die Klassenkomponente `x` zugewiesen Wert davon ab, ob das Argument `z` auf `TRUE` oder `FALSE` gesetzt ist (siehe Codebsp. 22 Zeile 5 bis 9).

Wichtig im Zusammenhang mit der `initialize`-Methode ist zudem noch, dass nach deren Definition die Validierungsfunktion nicht mehr implizit beim Aufruf von `new()` ausgeführt wird. Soll daher weiterhin eine Prüfung des erzeugten Objekts stattfinden, muss dies nun explizit in der `initialize`-Methode durch Aufruf der `validObject`-Methode (siehe Zeile 12) bewerkstelligt werden.

```
setMethod(                                                    #1
        f="initialize",                                       #2
        signature="ExampleClass",                             #3
        definition=function(.Object,x,y,z){                   #4
                if(z==TRUE){                                  #5
                        .Object@x<-2*x                        #6
                }else{                                        #7
                        .Object@x<-x                          #8
                }                                             #9
                .Object@y<-y                                  #10
                .Object@z<-z                                  #11
                validObject(.Object)                          #12
                return(.Object)                               #13
        }                                                     #14
)                                                             #15
```

Codebsp. 22: `initialize`-Methode für ExampleClass

5.2.2.3 S4 Methoden

Methoden haben gegenüber Funktionen den Vorteil, dass für diese – je nachdem über welches Objekt einer Klasse sie aufgerufen werden – ein unterschiedliches Verhalten durch den Entwickler spezifiziert werden kann.

Bevor jedoch eine Methode zu einer Klasse definiert werden kann, muss vorab für eine Methode ein Generic mit der `setGeneric`-Methode definiert werden – in dem Codebsp. 22 ist dieser intern bereits vordefiniert.

Bei `setGeneric()` wird in Codebsp. 23 zum einen der Name (siehe Zeile 1) und zum anderen die Signatur (siehe Zeile 2 bis 4) festgelegt.

Im Anschluss kann die Methode frei für eine oder mehrere beliebige Klassen über `setMethod`() implementiert werden. Die Funktion benötigt hierfür drei Argumente: den Namen der Methode, die definiert wird (siehe Zeile 8), die Signatur der Klasse (siehe Zeile 9) – für welche die Implementierung erfolgt – und die eigentliche Logik durch Angabe einer Funktion (Zeile 10 bis 16).

In R werden prinzipiell bei Funktionsaufrufen Kopien von den Parametern erstellt und auf der Kopie weitergearbeitet (WICKHAM 2015). Soll daher in der Methode eine Änderung an einem als Parameter übergebenen Objekt vorgenommen werden, muss dies dadurch bewerkstelligt werden, dass das lokale Objekt in der Wirkungsumgebung der Methode das entsprechende Objekt in der Elternumgebung ersetzt. Dies geschieht beispielsweise in dem Codebsp. 23 in Zeile 15.

```
setGeneric("exampleMethod",                                          #1
        function(.Object, newX, newY , newZ){                        #2
                standardGeneric("exampleMethod")                     #3
        }                                                            #4
)                                                                    #5
                                                                     #6
setMethod(                                                           #7
        "exampleMethod",                                             #8
        signature="ExampleClass",                                    #9
        definition=function(.Object,newX,newY,newZ){                 #10
                obj<-.Object                                         #11
                obj@x<- newX                                         #12
                obj@y<- newY                                         #13
                obj@z<- newZ                                         #14
                eval.parent(substitute(.Object<-obj))                #15
        }                                                            #16
)                                                                    #17
```

Codebsp. 23: Generic- und Methodendefinition

5.2.2.4 Attributzugriff

Besteht im Allgemeinen beim S4 System die Möglichkeit, auf die Komponenten einer Klasse direkt über das @-Zeichen als Operator sowohl lesend (siehe Codebsp. 24 Zeile 3) als auch schreibend (siehe Codebsp. 24 Zeile 2) zuzugreifen, gebietet es sich dennoch aus softwaretechnischen Gründen Setter und Getter für den Zugriff auf Klassenattribute zu verwenden.

```
>A1<-ExampleClass()                                                  #1
>A1@x<-999                                                           #2
>A1@x                                                                #3
[1] 999                                                              #4
```

Codebsp. 24: direkter Zugriff auf Klassenkomponente

Im Folgenden wird ein Setter (Codebsp. 25) mit `setReplaceMethod()` definiert, die ihrerseits wiederum nur `setMethod()` aufruft und rein zur intuitiveren Benutzung mit dem `"<-"`-Operator dienlich sein soll; wie in Codebsp. 23 Zeile 15 wird auch hier intern eine Substituierung des Kindobjekts mit dem Elternobjekt vollzogen. Als Besonderheit ist der definierte Generic hervorzuheben (siehe Codebsp. 25 Zeile 1), da hier der Operator`"<-"` bei der Definition mit angegeben wird. Für die Art und Weise der Benutzung des Setters sei auf Codebsp. 27 Zeile 2 verwiesen.

```
setGeneric("setX<-",function(.Object,value){standardGeneric("setX<-")})  #1
setReplaceMethod(                                                    #2
        f="setX",                                                    #3
        signature="ExampleClass",                                    #4
        definition=function(.Object,value){                          #5
                .Object@x<-value                                     #6
                return(.Object)                                      #7
        }                                                            #8
)                                                                    #9
```

Codebsp. 25: Setter für Klassenattribut

Die Definition des Getters bedarf keiner besonderen Erklärung, gleicht sie doch in Gänze der einer ganz gewöhnlichen Methode (siehe Abschnitt 5.2.2.3). Zur besseren Verständlichkeit

wird unterhalb jedoch ein Beispiel zu einer Getter Definition (siehe Codebsp. 26) und deren Benutzung (siehe Codebsp. 27 Zeile 3) aufgeführt.

```
setGeneric("getX",function(.Object){standardGeneric("getX")})                #1
setReplaceMethod(                                                            #2
        f="getX",                                                            #3
        signature="ExampleClass",                                            #4
        definition=function(.Object){                                        #5
                return(.Object@x)                                            #6
        }                                                                    #7
)                                                                            #8
```

Codebsp. 26: Getter für Klassenattribut

Codebsp. 27 zeigt hier das Zusammenspiel von Getter und Setter-Methoden.

```
>A1<-ExampleClass()                                                          #1
>setX(A1)<-999                                                              #2
>getX(A1)                                                                   #3
[1] 999                                                                      #4
```

Codebsp. 27: Zusammenspiel Setter und Getter

5.2.2.5 Vererbung

Vererbung ist ein mächtiges Werkzeug, das konzeptionell unabdingbar in der Objektorientierung ist. Erbt eine Klasse von einer anderen, verfügt diese über alle Klassenkomponenten und Methoden der Elternklassen. Dabei steht es dem Entwickler frei, wo gewünscht, die jeweilige Methode der Elternklassen zu überschreiben, um ein anderes Verhalten bei Methodenaufruf zu spezifizieren. Wird bei der Elternklasse in der Definition bei der Repräsentation `"VIRTUAL"` als Literal mit angegeben, handelt es sich hier um eine virtuelle Klasse. Von dieser kann kein Objekt direkt erzeugt, aber Kindklassen abgeleitet werden.

Codebsp. 28 zeigt eine Kindklasse von `ExampleClass`; somit stehen dieser deren Slots `x, y` und `z` sowie deren Methoden `exampleMethod, setX` und `getX` zur Verfügung. Daneben definiert sie aber auch einen neuen eigenen Slot `l` (siehe Zeile 7).

```
setClass (                                                                   #1
      Class= "ExampleClassSibling",                                          #2
      representation(                                                        #3
      l="numeric"                                                            #4
      ) ,                                                                    #5
      contains="ExampleClass"                                                #6
)                                                                            #7
```

Codebsp. 28: Kindklasse von ExampleClass

Die Kindklasse `ExampleClassSibling` ruft im Codebsp. 29 die Methode `exampleMethod` auf; das Verhalten entspricht hier dem der Elternklasse (siehe Codebsp. 23). Im Grunde kommt es hier lediglich zu einer Neubelegung der Slots mit den spezifizierten Parametern der Methode.

```
> A_Child1<-new("ExampleClassSibling")                                    #1
> exampleMethod(A_Child1,newX = 1,newY = "uses parents‘ method",newZ =FALSE)  #2
```

Codebsp. 29: Aufruf von exampleMethod vor Überschreiben

Im Weiteren wird in Codebsp. 30 die Elternmethode überschrieben. Hierzu wird Methode erneut definiert, nur dass in diesem Fall die Signatur der Klasse auf die Kindklasse `ExampleClassSibling` (siehe Zeile 3) verweist und anders als in Codebsp. 23 auch kein Generic erneut definiert werden muss.

Bei der neuen Methode ändert sich deren Verhalten. Für den Slot `x` wird der doppelte Parameterwert zugewiesen und auch der String, welcher für die Komponente `y` bestimmt ist, wird "Sibling" vorangestellt.

```
setMethod(                                                    #1
        "exampleMethod",                                      #2
        signature="ExampleClassSibling",                      #3
        definition=function(.Object,newX,newY,newZ){          #4
                obj<-.Object                                  #5
                obj@x<- newX*2                                #6
                obj@y<- paste("Sibling",newY)                 #7
                obj@z<- newZ                                  #8
                eval.parent(substitute(.Object<-obj))         #9
        }                                                     #10
)                                                             #11
```

Codebsp. 30: Überschreiben der Elternmethode exampleMethod

Der Aufruf von `exampleMethod` (siehe Codebsp. 31 Zeile 2) veranschaulicht das durch die Überschreibung veränderte Methodenverhalten. Dem Slot `x` wurde der doppelte Parameterwert zugewiesen (siehe Zeile 8 und 9) und beim `y` wurde dem übergebenen String "Sibling" vorangestellt (siehe Zeile 11 und 12).

```
> A_Child1<-new("ExampleClassSibling")                                    #1
> exampleMethod(A_Child1,newX = 1,newY = "overwrites method",newZ =FALSE)  #2
> A_Child1                                                                #3
An object of class "ExampleClassSibling"                                  #4
Slot "l":                                                                 #5
numeric(0)                                                                #6
                                                                          #7
Slot "x":                                                                 #8
[1] 2                                                                     #9
                                                                          #10
Slot "y":                                                                 #11
[1] "Sibling overwrites method"                                           #12
                                                                          #13
Slot "z":                                                                 #14
[1] FALSE                                                                 #15
```

Codebsp. 31: Aufruf von exampleMethod nach Überschreiben

Die beispielhaften Erläuterungen sollten vor allem die Besonderheiten der Objektorientierung mit S4 verdeutlichen und dem Leser zu einem besseren Verständnis für die folgende Vorstellung des Nucleolus Framework verhelfen.

5.3 Konzeption eines Nucleolus Frameworks

Im folgenden Abschnitt wird die Konzeption des Frameworks für die Berechnung der verschiedenen Nucleolus Derivate auf Grundlage eines objektorientierten Ansatzes in R mit S4 Klassen vorgestellt. Die Gründe für eine Auswahl des S4 Ansatzes wurden bereits im Abschnitt 5.2.2 erörtert.

Bei der Konzeption eines Frameworks der Berechnung des Nucleolus und dessen Derivaten kamen neben einer gewünschten sauberen Trennung von Verantwortlichkeiten weitere Hauptüberlegungen zum Tragen.

Zum einen galt es, möglichst flexibel auf die Unterschiede der verschiedenen Derivate reagieren zu können, zum anderen sollte redundanter Code weitestgehend vermieden werden.

Daher war es notwendig eine Modularisierung und Anpassung des bereits vorhandenen Programmcodes von GEBELE (2016) durchzuführen, um so stärker in den einzelnen Berechnungsschritten auf mögliche Unterschiede der verschiedenen Varianten eingehen zu können und redundanten Code zu vermeiden.

Für eine erste Trennung der Verantwortlichkeiten wurden zwei Klassen umgesetzt: die Klasse `LPCoopGameUtils` und `NucleolusBase.`

Mit der Klasse `LPCoopGameUtils` bestand die Absicht, Aufgaben und gehaltene Daten im Zusammenhang mit dem Linearen Programm von der Logik zur Berechnung des Nucleolus oder eines seiner Derivate abzugrenzen. Indes trägt die Klasse `NucleolusBase` die Zuständigkeit für die Implementierung des Berechnungs-Algorithmus.

Auf die beiden Klassen wird in den folgenden Abschnitten eingegangen. Vorab erfolgt jedoch eine kurze Einführung in die für die Lösung der Linearen Programme verwendeten GLPK Library.

5.3.1 Die GLPK Library und deren Verwendung in R

Der folgende Abschnitt gibt einen Kurzüberblick über die GLPK Library und das R-Packet `glpkAPI`, welches als Interface auf `GLPK` in `CoopGame` dient. Weiterhin soll ein kleines abschließendes Anwendungsbeispiel anhand der in `CoopGame` implementierten Prüfung auf Balanciertheit (siehe Abschnitt 3.3) dem Leser in einem einfachen Fall helfen, das Zusammenspiel beider Komponenten zu vermitteln.

5.3.1.1 GLPK und das R-Paket glpkAPI

Bei `GLPK` handelt es sich dabei um eine weit verbreitete Bibliothek, geschrieben in ANSI C, zur Lösung Linearer Programme. `GLPK` wurde dabei von Andrew O. Makhorin an der Russischen Universität für Luft- und Raumfahrt in Moskau entwickelt und im Jahr 2000 als Teil des GNU Projekts unter GPLv3 veröffentlicht (THE GNU PROJECT 2017).

Dabei bietet der eigenständige LP/MIP-Solver von `GLPK` die Möglichkeit für ein Lineares Problem die primale und duale Lösung anhand des in Abschnitt 2.2 vorgestellten Simplex und Innere-Punkte Algorithmus zu lösen, wie auch der Branch-And-Cut Methode, die an dieser Stelle lediglich erwähnt sei (THE GNU PROJECT 2017).

Die Definition der Linearen Programme erfolgt hierbei über AMPL – A Mathematical Programming Language (The GNU Project 2017). Bei AMPL handelt es sich um eine algebraische Modellierungssprache, die von Robert Fourer, David Gay und Brian Kernighan an den Bell Labratories entwickelt worden ist (AMPL OPTIMIZATION INC. 2013). Große Beliebtheit genießt AMPL aufgrund seiner Notation, die bei arithmetischen Ausdrücken starke Ähnlichkeiten zur gebräuchlichen algebraischen Darstellungsform aufweist. So erweist sie sich vor allem für Mathematiker ohne großen Einarbeitungsaufwand als intuitiv verständlich (FOURER et al.2009: 17).

Durch das von Claus Jonathan Fritzemeier, Gabriel Gelius-Dietrich und Louis Luangkesorn an der Universität Düsseldorf und auf CRAN veröffentlichte R Packet `glpkAPI` (FRITZEMEIER ET. AL. 2015) bietet sich die Möglichkeit in `CoopGame` über dessen Interface auf `GLPK` zur Berechnung Linearer Programme zuzugreifen.

5.3.1.2 Zusammenspiel von GLPK und glpkAPI am Beispiel der Balanciertheit

Um das Zusammenspiel von `GLPK` und `glpkAPI` an dieser Stelle zu veranschaulichen, wird das Beispiel aus Abschnitt 3.3.2 zu einem balancierten Spiel und dem Spielvektor $A = (1,2,2,4,3,4,6)$ wieder aufgegriffen.

Wie bereits in Abschnitt 3.3 erläutert, ist ein Spiel mit transferierbaren Nutzen dann balanciert, wenn der Kern nicht leer ist.

Hierzu muss die Lösung der Zielfunktion des untenstehenden Linearen Programmes (hier für den Dreispielerfall**)** für die Koalitionsfunktion v: $2^N \rightarrow \mathbb{R}$ nach Maximierung dem Wert für die große Koalition N mit v_N entsprechen.

Formel 118: Lineares Programm für die Prüfung auf Balanciertheit im Dreipersonenfall

$Maximiere$:

$$z = F(\lambda_1, \dots, \lambda_n) = \ v_1\ \lambda_1 + v_2\lambda_2 + v_3\lambda_3 + v_{1,2}\lambda_4 + v_{1,3}\lambda_5 + v_{2,3}\lambda_6 + v_{1,2,3}\lambda_7$$

$Unter\ Einhaltung\ der\ Nebenbedingungen$:

$$\begin{array}{cccccccccc} \lambda_1 & +0 & +0 & +\lambda_4 & +\lambda_5 & +0 & +\lambda_7 & = & 1 \\ 0 & +\lambda_2 & +0 & +\lambda_4 & +0 & +\lambda_6 & +\lambda_7 & = & 1 \\ 0 & +0 & +\lambda_3 & +0 & +\lambda_5 & +\lambda_6 & +\lambda_7 & = & 1 \end{array}$$

Unter Berücksichtigung des Dreipersonenspieles mit dem Spielvektor $A = (1,2,2,4,3,4,6)$ ergibt sich das bereits bekannte Lineare Programm mit der Zielfunktion aus Formel 24 und den Gleichheitsnebenbedingungen aus Formel 23.

Formel 119: Lineares Programm für die Prüfung auf Balanciertheit im Dreipersonenfall anhand eines konkreten Beispiels

$Maximiere$:

$$z = F(\alpha_1, \dots, \alpha_n) = \ \lambda_1 + 2\lambda_2 + 2\lambda_3 + 4\lambda_4 + 3\lambda_5 + 4\lambda_6 + 6\lambda_7$$

$Unter\ Einhaltung\ der\ Nebenbedingungen$:

$$\begin{array}{cccccccccc} \lambda_1 & +0 & +0 & +\lambda_4 & +\lambda_5 & +0 & +\lambda_7 & = & 1 \\ 0 & +\lambda_2 & +0 & +\lambda_4 & +0 & +\lambda_6 & +\lambda_7 & = & 1 \\ 0 & +0 & +\lambda_3 & +0 & +\lambda_5 & +\lambda_6 & +\lambda_7 & = & 1 \end{array}$$

Jenes Lineare Programm (siehe Formel 119) gilt es an dieser Stelle unter zur Hilfenahme der `GLPK`-Library über dessen R Interface `glpkAPI` zu lösen. Hierzu wurden für die Berechnung des Linearen Programmes Code-Fragmente aus der allgemeinen Funktion `isBalancedGame` entnommen und in verständlicher Art und Weise in Codebsp. 32 auf das konkrete Beispiel zugeschnitten.

In den Zeilen 5 bis 7 (siehe Nebenbedingungen in Formel 119) werden die Koeffizienten der Strukturvariblen der Koeffizientenmatrix zeilenweise festgelegt und eine entsprechende Matrix erstellt (siehe Zeile 11). Die Koeffizienten jeder Zeile entsprechen hier jeweils den charakteristischen Vektoren der Spieler.

Für den späteren Ladeprozess der Koeffizientenmatrix in das LP-Solver Objekt, werden in den Zeilen 15 und 16 die Zeilen- bzw. Spaltenindizes für deren Einträge definiert.

In den Zeilen 21 bis 23 folgt die Festlegung der Schranken für die Nebenbedingungen (unterer und oberer Schrankenwert sowie -typ). Hier wird durch den Schrankentyp `GLP_FX` (Variable ist fest definiert) eine Gleichheitsbedingung für jede der drei Nebenbedingungen angegeben, mit 1 als festem Wert.

Die Spezifikation der Zielfunktion wird den Zeilen 26 bis 29 durchgeführt. Im Beispiel wird jeweils für die Entscheidungsvariablen der Schrankentyp `GLP_LO` (Variable ist nach unten beschränkt) zusammen mit dem Schrankenwert 0 angegeben, hierdurch wird die Nichtnegativitätsbedingung gewahrt. Ansonsten werden die Koeffizienten entsprechend Formel 119 über die Einträge des Spielevektors festgelegt.

Die weiteren Wesentlichen Schritte im Anschluss sind:

1. Initialisierung einer LP-Solver Instanz (siehe Zeile 32)
2. Spezifizierung des Lineare Problem als Minimierungsproblem (siehe Zeile 35)
3. Spezifizierung der Anzahl der Nebenbedingungen (siehe Zeile 41)
4. Übermittelung der Nebenbedingungs-Eigenschaften
   an LP-Solver Instanz (siehe Zeile 49)
5. Spezifizierung der Spaltenanzahl (siehe Zeile 52)
6. Übermittelung der Spalten sowie Zielfunktion-Eigenschaften
   an LP-Solver Instanz (siehe Zeile 64)
7. Laden der Koeffizientenmatrix (siehe Zeile 67)
8. Lösen des Linearen Programms (siehe Zeile 70)

Anhand eines solchen Einsatz von `glpkAPI` und `R` wird als Lösung im Codebsp. 32 der Wert 6 für das dort spezifizierte Lineare Programm ermittelt (siehe Zeile 76). Der Wert 6 entspricht hierbei dem der großen Koalition, somit ist das Spiel nach dem Bondareva-Shapley Theorem aus Abschnitt 3.3.1 balanciert. Die weiteren im Beispiel aufgeführten Schritte sind nicht essentiell und dienen hauptsächlich der schöneren Darstellung.

```
#Game Vector A                                                    #1
A=c(1,2,2,4,3,4,6)                                                #2
                                                                  #3
#Coefficients for every restriction                               #4
r1=c(1,0,0,1,1,0,1)                                               #5
r2=c(0,1,0,1,0,1,1)                                               #6
r3=c(0,0,1,0,1,1,1)                                               #7
                                                                  #8
#Create coefficent matrix gathering                               #9
#all restrictions' coefficients                                   #10
LPCoeffMatrix<-matrix(c(r1,r2,r3), nrow=3, byrow = TRUE)          #11
                                                                  #12
#Create row and colum indices for later loading of                #13
#coefficient matrix                                               #14
indicesRows<-c(1,1,1,1,1,1,1,2,2,2,2,2,2,2,3,3,3,3,3,3,3)        #15
indicesCols<-c(1,2,3,4,5,6,7,1,2,3,4,5,6,7,1,2,3,4,5,6,7)        #16
                                                                  #17
#Define restrictions' lower and upper bound as well               #18
#as their boundary type                                           #19
rlb<-c(1.0,1.0,1.0)                                               #20
rub<-c(1.0,1.0,1.0)                                               #21
```

```
rtype<-c(GLP_FX,GLP_FX,GLP_FX)                                                  #22
                                                                                #23
#Define decision variables' lower and upper bound                             #24
#as well as their boundary type                                                 #25
clb<-c(0,0,0,0,0,0,0)                                                          #26
cub<-c(Inf,Inf,Inf,Inf,Inf,Inf)                                                #27
ctype<-c(GLP_LO,GLP_LO,GLP_LO,GLP_LO,GLP_LO,GLP_LO,GLP_LO)                     #28
obj<-A                                                                          #29
                                                                                #30
#Init instance of GLPK's Linear Programme                                       #31
prob=initProbGLPK()                                                             #32
                                                                                #33
#Set optimization direction                                                     #34
setObjDirGLPK(prob,GLP_MAX);                                                    #35
                                                                                #36
#Set name for Linear Programme                                                  #37
setProbNameGLPK(prob,"Checking Balancedness for Game");                         #38
                                                                                #39
#Set number of rows for Linear Programme                                        #40
addRowsGLPK(prob,3)                                                             #41
                                                                                #42
#Set row names for Linear Programme                                             #43
setRowNameGLPK(prob,1,"r1");                                                    #44
setRowNameGLPK(prob,2,"r2");                                                    #45
setRowNameGLPK(prob,3,"r3");                                                    #46
                                                                                #47
#Set rows' boundaries and boundary types                                        #48
setRowsBndsGLPK(prob,1:3,rlb,rub,rtype)                                         #49
                                                                                #50
#Set number of colums for Linear Programme                                      #51
addColsGLPK(prob,7)                                                             #52
                                                                                #53
#Set column names for Linear Programme                                          #54
setColNameGLPK(prob,1,"lambda_1")                                               #55
setColNameGLPK(prob,2,"lambda_2")                                               #55
setColNameGLPK(prob,3,"lambda_3")                                               #56
setColNameGLPK(prob,4,"lambda_4")                                               #57
setColNameGLPK(prob,5,"lambda_5")                                               #58
setColNameGLPK(prob,6,"lambda_6")                                               #59
setColNameGLPK(prob,7,"lambda_7")                                               #60
                                                                                #61
#Set columns' boundaries, objective function's coefficients                     #62
#and boundary types                                                             #63
setColsBndsObjCoefsGLPK(prob,1:7,clb,cub,obj,ctype)                             #64
                                                                                #65
#Load coefficient matrix to instance of the Linear Programme                    #66
loadMatrixGLPK(prob,21,indicesRows,indicesCols,as.vector(t(LPCoeffMatrix)))     #67
                                                                                #68
#Solve the Linear Programme                                                     #69
solveSimplexGLPK(prob)                                                          #70
                                                                                #71
#Retrieve the optimal values for the deciscion variables                        #72
valuesOfDecisionVariables=getColsPrimGLPK(prob);                                #73
                                                                                #74
#Retrieve the optimal solution for the objective function                       #75
objVal=getObjValGLPK(prob);                                                     #76
                                                                                #77
#Write corresponding Linear Programme                                           #78
writeLPGLPK(prob,"LP_in_AMPL_Notation.txt")                                     #79
```

Codebsp. 32: Zusammenspiel von GLPK und glpkAPI zur Prüfung eines Spiels auf Balanciertheit

5.3.2 Die Klasse LCoopGameUtils

Die Klasse `LPCoopGameUtils` behandelt Aufgaben im Hinblick auf das Lineare Programm und `GLPK` bzw. `glpkAPI`.

Um das Lineare Programm mit der Klasse `LPCoopGameUtils` (siehe Codebsp. 33) abbilden zu können, enthält diese drei weitere eigen definierte Klassen `LPBndsObjCoefs` (siehe Zeile 4), `LPRows` (siehe Zeile 5) und `LPMatrix` (siehe Zeile 6) als Klassenkomponenten. Mit dem Slot `LPObjDir` (siehe Zeile 8) wird festgehalten, ob es sich um ein Minimierungs- oder Maximierungsproblem handelt. Des Weiteren beinhaltet der Slot `LP` einen Pointer auf eine Instanz der Strukturklasse `glpkPtr`, welche ihrerseits Verweise auf die von `GLPK` benötigten `C` Strukturen bereitstellt.

```
setClass(                                                    #1
        "LPCoopGameUtils",                                   #2
        representation(                                      #3
                LPBndsObjCoefs="LPBndsObjCoefs",             #4
                LPRows="LPRows",                             #5
                LPMatrix="LPMatrix",                         #6
                LP="glpkPtr",                                #7
                LPObjDir="numeric"                           #8
        ),                                                   #9
        prototype=prototype(LPObjDir=GLP_MIN)                #10
)                                                            #11
```

Codebsp. 33: Klassendefinition LPCoopGameUtils

Die Klasse `LPRows` (siehe Codebsp. 34) definiert hierbei für die Nebenbedingungen, den Schrankentypen über den Slot `rtype` (siehe Zeile 6) und mit `rlb` die Werte für die unteren (siehe Zeile 4) und mit `rub` für die oberen Schranken (siehe Zeile 5).

```
setClass(                                                    #1
        "LPRows",                                            #2
        representation(                                      #3
                rlb="numeric",                               #4
                rub="numeric",                               #5
                rtype="numeric"                              #6
        ),                                                   #7
        validity=function(object){                           #8
                retVal=TRUE                                  #9
                lengthRlb=length(object@rlb)                 #10
                lengthRub=length(object@rub)                 #11
                lengthRtype=length(object@rtype)             #12
                if(!all(c(lengthRub,lengthRlb)==lengthRtype)){ #13
                        retVal=FALSE                         #14
                }elseif(log2(lengthRub+1)%%1!=0){            #15
                        retVal=FALSE                         #16
                }                                            #17
                return(retVal)                               #18
        }                                                    #19
)                                                            #20
```

Codebsp. 34: Klassendefinition LPRows

Die Koeffizienten der Nebenbedingungen werden durch die Klasse `LPMatrix` verwaltet.

```
setClass(                                                    #1
        "LPMatrix",                                          #2
        representation(matrix="matrix")                      #3
)                                                            #4
```
Codebsp. 35: Klassendefinition LPMatrix

Die Klasse `LPBndsObjCoefs` (siehe Codebsp. 36) trägt die Informationen über die Entscheidungsvariablen der Zielfunktion, mit `obj` über deren Koeffizienten (siehe Zeile 7), mit `cub` über deren oberen (siehe Zeile 5) und mit `clb` über deren unteren Schranken (siehe Zeile 4) sowie mit `ctype` über die Schrankentypen (siehe Zeile 6).

```
setClass(                                                    #1
        "LPBndsObjCoefs",                                    #2
        representation(                                      #3
                clb="numeric",                               #4
                cub="numeric",                               #5
                ctype="numeric",                             #6
                obj="numeric"                                #7
        ),                                                   #8
        validity=function(object){                           #9
                lengthClb=length(object@clb)                 #10
                lengthCub=length(object@cub)                 #11
                lengthCtype=length(object@ctype)             #12
                lengthObj=length(object@obj)                 #13
                if(!all(c(lengthClb,lengthCub,lengthCtype)==lengthObj)){  #14
                        return(FALSE)                        #15
                }                                            #16
        }                                                    #17
)                                                            #18
```
Codebsp. 36: Klassendefinition LPBndsObjCoefs

Neben den Gettern und Settern, auf die an dieser Stelle nicht näher eingegangen wird, stellt die Klasse `LPCoopGameUtils` noch weitere Methoden zur Verwaltung eines Linearen Programmes zur Verfügung.

So ändert die Methode `changeLPCoopGameUtilsObjDirToMin` die Richtung für die Optimierung auf ein Minimierungs- und `changeLPCoopGameUtilsObjDirToMax` entsprechend auf ein Maximierungsproblem.

Mit `solveLP()` wird die Lösung des in den Klassenkomponenten spezifizierten Linearen Programmes berechnet. Derzeitig handelt es sich bei dem zur Lösung verwendeten Algorithmus, der durch das Paket `glpkAPI` aufgerufen wird, um einen Vertreter aus der Klasse der Simplex Algorithmus Verfahren. Es ist aber ohne weiteres möglich bei Bedarf eine Klasse von `LPCoopGameUtils` abzuleiten und deren `solveLP`-Methode zu überschreiben, so dass ein anderer Lösungsalgorithmus für das Lineare Programm Verwendung findet.

Nach der Lösung des Linearen Programms, kann mit `isFeasible` geprüft werden, ob eine zum Linearen Programm zulässige Lösung gefunden worden ist. Ist dies der Fall stehen mit `getLPDualSolution`, `getLPPrimalSolution`, `getLPObjVal` Methoden bereit, um Informationen zu dem berechneten Linearen Programm abzufragen.

Die primale Lösung liefert `getLPPrimalSolution` und die korrespondierende duale Lösung `getLPDualSolution`; mit `getLPObjVal` kann hingegen der errechnete Zielfunktionswert abgerufen werden.

Werden Änderungen an den Klassenattributen vorgenommen, kann mit `updateLPCoopGameUtils` das Lineare Programm aktualisiert werden.

Mit der Klasse `LPCoopGameUtils`, `LPBndsObjCoefs`, `LPRows` und `LPMatrix` wurden alle benötigten Funktionalitäten zur Verwaltung des Linearen Programms bereitgestellt. Ein ausführliches Klassendiagramm ist im Anschluss (siehe Abbildung 1) aufgeführt.

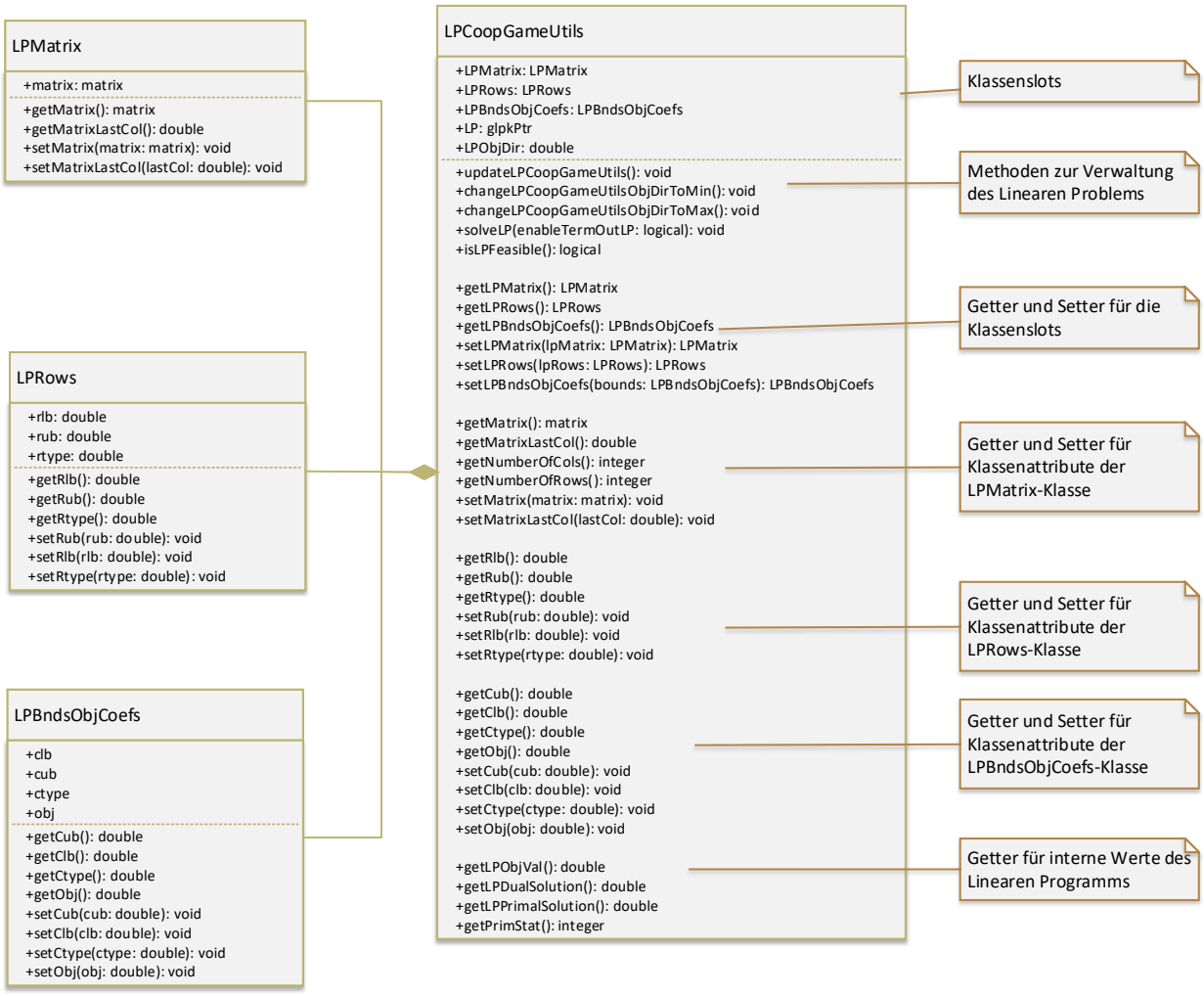


Abbildung 1: Klassendiagramm LPCoopGameUtils

5.3.3 Die Klasse NucleolusBase

Im vorigen Abschnitt wurde die Klasse `LPCoopGameUtils` beschrieben, welche die Basis für die einfache Handhabung aller Verantwortlichkeiten im Zusammenhang der Linearen Programme schafft. Die virtuelle Klasse `NucleolusBase` hält dagegen die Zuständigkeit für die Implementierung des Algorithmus zur Berechnung der jeweiligen Nucleolus Derivate. Da

es sich hier um eine virtuelle Klasse handelt, kann von dieser keine Instanz erzeugt werden – nur für deren Kindsklassen. Den Berechnungen der verschiedenen Derivate ist gemein, dass bei ihnen über eine Folge von Linearen Programmen, der optimale Lösungspunkt bestimmt wird. Unterschiede hingegen können sich bei der Berechnung der Derivate sowohl in der anfänglichen Initiierung bzw. späteren Aktualisierung der Koeffizientenmatrix, verschiedenen Abbruchkriterien, Schrankentypen sowie in Schrankenwerten für das Lineare Programm zeigen.

Die virtuelle Klasse `NucleolusBase` orientiert sich dabei an der Umsetzung von GEBELE (2016) und realisiert die Berechnung des Nucleolus als Ausgangspunkt für die weiteren Derivate.

In welchen Punkten und wie stark die unterschiedlichen Derivate sich hierbei voneinander abgrenzen kann dabei sehr verschieden ausgeprägt sein.

Im Folgenden wird darauf eingegangen, wie die Klasse `NucleolusBase` im Hinblick auf eine möglichst flexible und einfache spätere Implementierung der Nucleolus Derivate aufgebaut ist und welche Überlegungen hier speziell zum Tragen kamen.

5.3.3.1 Aufbau der Klasse NucleolusBase

Die virtuelle Klasse `NucleolusBase` (siehe Codebsp. 37) enthält als Klassenattribute den Spielevektor A (siehe Zeile 4) und ein Objekt der Klasse `LPCoopGameUtils` (siehe Zeile 5).

```
setClass(                                                    #1
        "NucleolusBase",                                     #2
        representation(                                      #3
                A="numeric",                                 #4
                LPCoopGameUtils="LPCoopGameUtils",           #5
                "VIRTUAL"                                    #6
        )                                                    #7
)                                                            #8
```

Codebsp. 37: Klassendefinition NucleolusBase

Daneben enthält die Klasse (siehe unten Abbildung 2) Methoden, um indirekt über sein Objekt vom Typ `LPCoopGameUtils` mit `initLPBndsObjCoefs`, `initLPMatrix`, `initLPRows` sowie `determineExcessCoefficients` in die Initialisierung und mittels `updateLPBndsObjCoefs`, `updateLPMatrix`, `updateLPRows`, `getLPDualSolutionPos` sowie `getLPRowsBoundsFunc` in die Aktualisierung eingreifen zu können.

Die Methode `checkAbort` steuert dagegen, unter welchen Umständen bei der Berechnung die zuständige Methode `calculateNucleolus` vorzeitig abgebrochen werden soll, falls entweder die Berechnung abgeschlossen oder keine Lösung greifbar ist.

Alle Derivate erben später hierbei von der Klasse `NucleolusBase` (siehe Abschnitt 5.4) und bei Unterschieden wird die zuständige Methode einfach überschrieben (siehe Abschnitt 5.2.2.5).

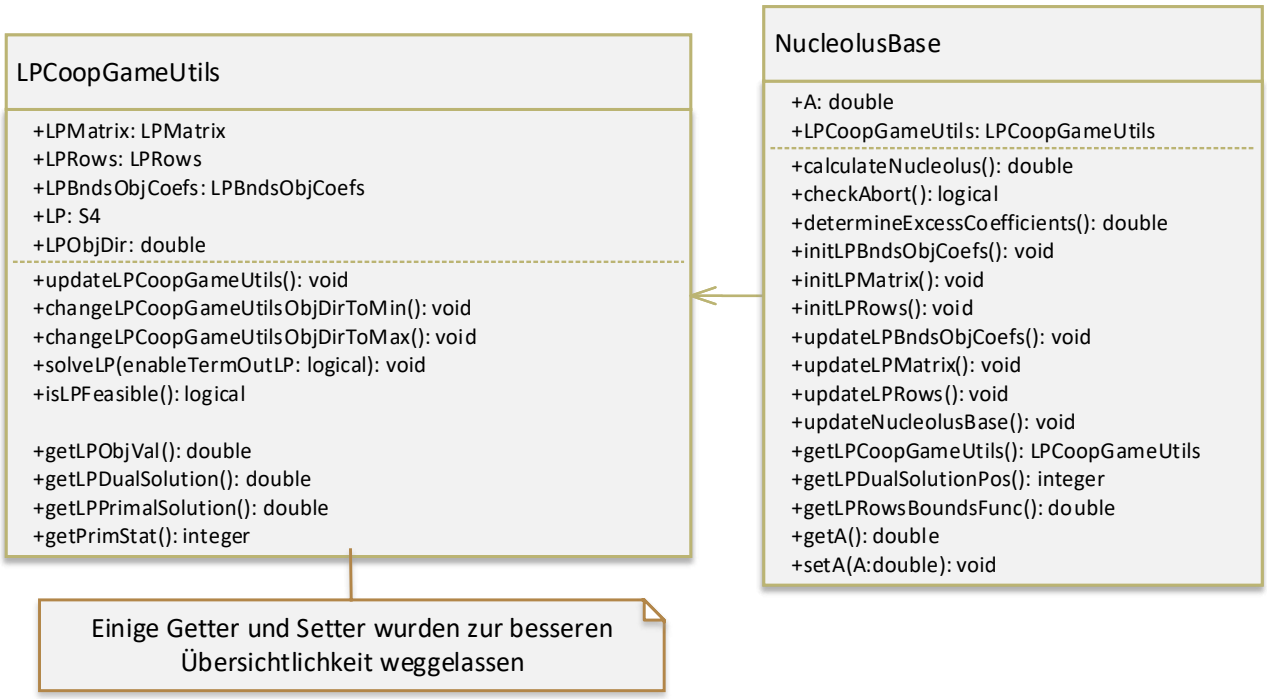


Abbildung 2: Klassendiagramm NucleolusBase mit Abhängigkeit zu LP`CoopGame`Utils

Im Weiteren findet nun der Lösungsablauf mit der Objektinitialisierung in der `initialize` und der Logik innerhalb der Methode calculateNucleolus besondere Beachtung, da hier zuerst alle wichtigen Voraussetzungen geschaffen und später die wesentlichen Schritte zur Berechnung des Nucleolus bzw. dessen Derivat angestoßen werden.

5.3.3.2 Berechnung der Lösung der Nucleolus- Derivate in allgemeiner Form

Der allgemeine Lösungsablauf lässt sich hierbei folgendermaßen beschreiben:

1. Schritt: Anfängliche Initialisierung des Linearen Programms

Im ersten Schritt erfolgt die Initialisierung des Linearen Programms in der `initialize`-Methode (siehe Codebsp. 38) bei der Objekterzeugung eines Kindobjekts von `Nucleolus-Base`.

Neben der Zuweisung des Spielvektors zum zugehörigen Slot (siehe Zeile 5) und der Erzeugung eines Objekts des Typ `LPCoopGameUtils` wird sowohl die Koeffizientenmatrix (siehe Zeile 7) als auch die Schrankentypen und Schrankenwerte der einzelnen Restriktionen (siehe Zeile 8) sowie der Entscheidungsvariablen mit dem entsprechenden Vektor für die Zielfunktions-Koeffizienten (siehe Zeile 9) initialisiert.

Ein abschließender Aufruf der `updateLPCoopGameUtils` (siehe Zeile 10) sorgt dafür, dass anhand der vorausgegangenen Initialisierung im Anschluss der LP-Solver von `GLPK` entsprechend mit den in der Klasse `LPCoopGameUtils` spezifizierten Daten initialisiert wird.

```
setMethod(                                                          #1
        f="initialize",                                             #2
        signature="NucleolusBase",                                  #3
        definition=function(.Object,A){                             #4
                .Object@A=A                                         #5
                .Object@LPCoopGameUtils<-LPCoopGameUtils()          #6
                initLPMatrix(.Object)                               #7
                initLPRows(.Object)                                 #8
                initLPBndsObjCoefs(.Object)                         #9
                updateLPCoopGameUtils(.Object@LPCoopGameUtils)      #10
                return(.Object)                                     #11
        }                                                           #12
)                                                                   #13
```

Codebsp. 38: initialize-Methode von NucleolusBase

Für den Entwickler besteht hier die Möglichkeit für benötigte Anpassungen bezüglich der kompletten Koeffizientenmatrix, die Methode `initLPMatrix` zu überschreiben. Unterscheidet sich das zu implementierende Derivat lediglich in der initialen Zuweisung der zu minimierenden Strukturvariable, reicht es aus, `determineExcessCoefficients` zu reimplementieren.

Für abweichende initiale Belegungen der Schrankentypen und Schrankenwerte der einzelnen Restriktionen (siehe Zeile 8) sowie der Entscheidungsvariablen und Zielfunktions-Koeffizienten gilt es hingegen, jeweils die Methoden `initLPRows` bzw. `initLPBndsObjCoefs` neu zu definieren.

2. Schritt: Abfolge Linearer Programme zur Bestimmung der eindeutigen Lösung

Die Bestimmung der eindeutigen Lösung erfolgt nach Aufstellung des anfänglichen Linearen Programms (siehe 1. Schritt) in einer Abfolge Linearer Programme, gesteuert durch die Methode `calculateNucleolus` (siehe Codebsp. 39).

```
setMethod(                                                          #1
        "calculateNucleolus",                                       #2
        signature="NucleolusBase",                                  #3
        definition=function(.Object,enableTermOutLP=TRUE){          #4
                checkGamePreconditions(.Object)                     #5
                N=getNumberOfRows(.Object@LPCoopGameUtils)          #6
                oldObjValue=NULL                                    #7
                primal=NULL                                         #8
                for(i in 1:N){                                      #9
                        solveLP(.Object@LPCoopGameUtils,enableTermOutLP)    #10
                        if(checkAbort(.Object,oldObjValue)){        #11
                                break                               #12
                        }                                           #13
                        oldObjValue=getLPObjVal(.Object@LPCoopGameUtils)    #14
                        primal=getLPPrimalSolution(.Object@LPCoopGameUtils) #15
                        updateNucleolusBase(.Object)                #16
                        if(all(getMatrixLastCol(.Object@LPCoopGameUtils)==0)){  #17
                                break                               #18
                        }                                           #19
                }                                                   #20
                return((primal[-length(primal)]))                   #21
        }                                                           #22
)                                                                   #23
```

Codebsp. 39: calculateNucleolus Methode zur Berechnung der Nucleolus Lösung

Zu Beginn prüft hier die Methode `checkGamePreconditions`, ob das definierte Spiel die für eine eindeutige Lösung geforderte Eigenschaft der Wesentlichkeit erfüllt.

Dabei werden in einer Schleife (siehe Zeile 9 bis 20) mit höchstens *N* Iterationen (hier Anzahl der Restriktionen, entspricht beim Nucleolus der Anzahl der Spielvektoreinträge) immer wieder folgende Teilschritte durchlaufen.

Teilschritte:

a) Lösen des aktuellen Linearen Programms (siehe Zeile 10)
b) Prüfen auf vorzeitigen Abbruch (siehe Zeile 11)
c) Aktualisieren des Linearen Programms (siehe Zeile 16)
d) Prüfen auf regulären Abbruch,
   ist für alle Restriktionen das jeweilige Optimum gefunden worden (siehe Zeile 17)?
   - Falls JA: Abbruch
   - Falls NEIN: Gehe zurück zu Teilschritt a

Während des Durchlaufens der Teilschritte a bis d, kam es entweder in der `checkAbort`-Methode (siehe Zeile 11) zum vorzeitigen Komplettabbruch, bei dem die Methode mit einem Fehlerhinweis stoppt, oder es wird im Anschluss die bestimmte punktwertige Lösung zurückgegeben (Zeile 21).

Für den Entwickler besteht auch in Schritt 2 die Möglichkeit einzugreifen und auf Besonderheiten beim jeweiligen Derivat einzugehen.

Zum einen kann der Entwickler unterschiedliche Abbruchkriterien durch Überschreiben von `checkAbort` definieren, zum anderen kann er auch die Aktualisierung des Linearen Programms durch die Methode `updateNucleolusBase` (siehe Codebsp. 40) beeinflussen.

```
setMethod(                                                          #1
        "updateNucleolusBase",                                      #2
        signature="NucleolusBase",                                  #3
        definition=function(.Object){                               #4
                obj<-.Object                                        #5
                updateLPMatrix(obj)                                 #6
                updateLPRows(obj)                                   #7
                updateLPBndsObjCoefs(obj)                           #8
                updateLPCoopGameUtils(obj@LPCoopGameUtils)          #9
                eval.parent(substitute(.Object<-obj))               #10
        }                                                           #11
)                                                                   #12
```

Codebsp. 40: updateNucleolusBase Methode

Die Methode `updateNucleolusBase` umfasst seinerseits zahlreiche weitere Aufrufe von Methoden wie `updateLPMatrix`, `updateLPRows` und `updateLPBndsObjCoefs`, die ihrerseits je nach Bedarf überschreibbar sind. Für `updateLPMatrix` (siehe Zeile 6) besteht aber kaum Notwendigkeit diese zu überschreiben. So setzt sie standardmäßig lediglich die Einträge der letzten Spalte für die zu minimierenden Strukturvariable aus der Zielfunktion in

der Koeffizientenmatrix auf den Wert 0. Dieses Verhalten entsprach allen bisherigen Implementierungen.

Deutlich Unterschiede bei den Derivaten offenbaren sich dagegen bezüglich der Aktualisierung der Schrankentypen sowie Schrankenwerte der einzelnen Restriktionen. Dem wurde durch Reimplementierungen der updateLPRows bei großen und bei kleineren Abweichungen der Methode getLPRowsBoundsFunc Rechnung getragen.

Bei der `getLPRowsBoundsFunc` Methode handelt es sich hierbei um eine Methode, die zur Bestimmung des jeweiligen Schrankenwertes – für die Restriktion mit dem gefundenen optimalen Wert – aufgerufen wird.

Rein der Vollständigkeit sowie der besseren Trennung halber und um allen etwaigen Unterschieden gewappnet zu sein, wird auch die `updateLPBndsObjCoefs` Methode aufgeführt. Diese Methode verfügt zwar über keinerlei Logik, ist aber bei nötigen Aktualisierungen der Schrankenwerte und -typen sowie der Koeffizienten für die Zielfunktion zu überschreiben.

Im Folgenden wird auf die jeweiligen Implementierungen eingegangen.

5.4 Implementierung der Nucleolus-Derivate

Unter Nutzung des im Abschnitt 5.3 eingehend vorgestellten Frameworks wurde der Nucleolus sowie der Prenucleolus reimplementiert und auch neue Derivate wie der Per Capita Nucleolus, der Proportional Nucleolus, der Disruption Nucleolus, der Modiclus und einer dessen Varianten des Simplified Modiclus realisiert. Durch den modularen Aufbau und die hohe Granularität war dies zum Großteil ohne größere Anpassung möglich.

Im Weiteren wird auf die jeweilige Implementierung des Nucleolus und dessen Varianten eingegangen, wobei die Verwirklichung des Nucleolus besonders Beachtung findet; dient sie doch als Grundlage aller weiteren Derivate.

5.4.1 Implementierung des Nucleolus

Das von GEBELE (2016) implementierte Konzept des Nucleolus diente als Orientierungshilfe für die in der Klasse `NucleolusBase` vorhandene Logik zur Berechnung der Derivate.

Für die Realisierung des Nucleolus reichte es daher aus, lediglich für die Klasse `Nucleolus` ohne jegliche Anpassung die Klasse `NucleolusBase` anzugeben, da die Logik hier immer mit dem des Nucleolus Konzept übereinstimmt, sofern keine Überschreibung einer Methode erfolgt.

Erfolgte in Abschnitt 5.3.3 eher eine abstrakte Betrachtung der Klasse `NucleolusBase`, wird daher an dieser Stelle noch einmal kurz auf die implementierte Logik in `NucleolusBase` im besonderen Hinblick auf die Realisierung des Nucleolus Konzepts eingegangen.

5.4.1.1 Anfängliche Initialisierung des Linearen Programms

Für die Initialisierung der Koeffizientenmatrix zum Linearen Programm wird die `initLP-Matrix` Methode (siehe Codebsp. 41) spezifiziert.

Bei einer Anzahl von n Spielern wird eine Matrix mit $(2^n - 1)$ Zeilen- und $(n + 1)$ Spalteneinträgen erstellt.

Die Anzahl der Zeilen ergibt sich hierbei aus der Zahl möglicher Koalition entsprechend der Mächtigkeit der Potenzmenge für die Spieleranzahl von n mit $card\big(Pot(\{1, \dots, n\})\big) = 2^n$, wobei die Nullkoalition mit einer leeren Spielermenge ausgeschlossen wird und sich so der Wert von $2^n - 1$ erschließt.

Die ersten n Spalten enthalten für jeden Spieler den charakteristischen Vektor, der darüber Aufschluss gibt, ob der betreffende Spieler in der jeweiligen Koalition beteiligt ist (siehe Abschnitt 2.1.5).

Die Spalte für die zu minimierenden Strukturvariable aus der Zielfunktion – mit dem Index $(n + 1)$ – erhält die durch den Aufruf der Methode `determineExcessCoefficients` bestimmten Werte zugewiesen.

```
setMethod(                                                                  #1
        "initLPMatrix",                                                     #2
        signature="NucleolusBase",                                          #3
        definition=function(.Object){                                       #4
                lpCoopGameUtils<-.Object@LPCoopGameUtils                    #5
                excess<-determineExcessCoefficients(.Object)                #6
                coeffMat<-createBitMatrix(n=getNumberOfPlayers(.Object@A),excess)  #7
                setMatrix(lpCoopGameUtils)<-coeffMat                        #8
                eval.parent(substitute(.Object@LPCoopGameUtils<-lpCoopGameUtils))  #9
        }                                                                   #10
)                                                                           #11
```

Codebsp. 41: initLPMatrix Methode

Im Fall des Nucleolus wird hier bei der `determineExcessCoefficients Methode` (siehe Codebsp. 42) lediglich die Stukturvariable des Excess mit '1' als Koeffizienten für alle Einträge bis auf die große Koalition mit '0' zugewiesen.

```
setMethod(                                                                  #1
        "determineExcessCoefficients",                                      #2
        signature="NucleolusBase",                                          #3
        definition=function(.Object){                                       #4
                N=length(.Object@A)                                         #5
                return(c(rep(1,N-1),0))                                     #6
        }                                                                   #7
)                                                                           #8
```

Codebsp. 42: determineExcessCoefficients Methode

In der `initLPRows` Methode (siehe Codebsp. 43) werden als untere Schranke die Werte des Spielvektors (siehe Zeile 7) gesetzt. Dagegen bleiben die Restriktionen nach oben hin unbeschränkt (siehe Zeile 8), mit Ausnahme für den Eintrag der großen Koalition. Für diesen wird auch hier der entsprechende Koalitionswert als obere Schranke (siehe Zeile 8) gesetzt. Für die Schrankentypen wird für alle Restriktionen mit 'GLP_LO' angegeben, dass diese, bis auf die Restriktion für die große Koalition, nach unten beschränkt sind (siehe Zeile 9). In der Restriktion für die große Koalition wird mit 'GLP_FX' für den Schrankentyp eine Gleichheitsnebenbedingung gesetzt (siehe Zeile 9).

```
setMethod(                                                                  #1
        "initLPRows",                                                       #2
        signature="NucleolusBase",                                          #3
        definition=function(.Object){                                       #4
                lpCoopGameUtils<-.Object@LPCoopGameUtils                    #5
                N<-length(.Object@A)                                        #6
                setRlb(lpCoopGameUtils)<-.Object@A                          #7
                setRub(lpCoopGameUtils)<-c(rep(Inf,N-1),.Object@A[N])       #8
                setRtype(lpCoopGameUtils)<-c(rep(GLP_LO,N-1),GLP_FX)        #9
                eval.parent(substitute(.Object@LPCoopGameUtils<-lpCoopGameUtils))  #10
        }                                                                   #11
)                                                                           #12
```

Codebsp. 43: initLPRows Methode

Die Methode `initLPBndsObjCoefs` ist für den Nucleolus so definiert, dass für die unteren Schranken der Entscheidungsvariablen die Werte der Einerkoalitionen $v(\{i\})$ festgelegt werden (siehe Zeile 7). Für die obere Schranke wird festgelegt, dass diese ins Positive auch unbeschränkt sind (siehe Zeile 8). Eine Ausnahme stellt die zu minimierende Entscheidungsvariable für den Überschuss dar; bei ihr handelt es sich um eine freie Variable, die gänzlich unbeschränkt ist.

Bei der Zielfunktion soll nur die Entscheidungsvariable für den Überschuss minimiert werden (siehe Zeile 10).

```
setMethod(                                                                  #1
        "initLPBndsObjCoefs",                                               #2
        signature="NucleolusBase",                                          #3
        definition=function(.Object){                                       #4
                lpCoopGameUtils<-.Object@LPCoopGameUtils                    #5
                n=getNumberOfPlayers(.Object@A)                             #6
                setClb(lpCoopGameUtils)<-c(.Object@A[1:n],-Inf) #changed    #7
                setCub(lpCoopGameUtils)<-rep(Inf,n+1)                       #8
                setCtype(lpCoopGameUtils)<-c(rep(GLP_DB,n),GLP_FR)#changed  #9
                setObj(lpCoopGameUtils)<-c(rep(0,n),1)                      #10
                eval.parent(substitute(.Object@LPCoopGameUtils<-lpCoopGameUtils))  #11
        }                                                                   #12
)                                                                           #13
```

Codebsp. 44: initLPBndsObjCoefs Methode

Die Zeilen 7 und 9 weisen hierbei Unterschiede zu den ursprünglichen Schrankenwerten des Nucleolus von GEBELE (2015) auf. Die Forderung einer Auszahlung nach individueller Rationalität wird umgesetzt, als Lösung liefert der Nucleolus somit nach Änderung eine Imputation. Zuvor bestimmte der Nucleolus als Lösung teilweise auch lediglich eine Präimputation.

Als Beispiel kann hier das unwesentliche Spiel mit dem Spielevektor $A = c(2,2,1)$ angeführt werden. Die alte Nucleolus Implementierung zeichnete hier die Präimputation $c(0.5,0.5)$ als Lösung aus. Die neue Implementierung lässt dagegen solche Spiele auf Grund der damit verbundenen leeren Imputationsmenge von vornherein nicht mehr zu.

5.4.1.2 Berechnung der Nucleolus Lösung

Bei dem unten aufgeführten Codebsp. 45 führt es zum sofortigen Abbruch der Nucleolous Berechnung, wenn keine Lösung für das Lineare Programm greifbar ist (siehe Zeile 11-13). Für diesen Fall liefert die Berechnung auch keinen Lösungswert.

Sofern der alte Zielfunktions- auch mit dem neuen Zielfunktionswert übereinstimmt, gibt die Methode `TRUE` zurück. Die Rückgabe von `TRUE` führt in `calculateNucleolus` anschließend zum Verlassen der Schleife und die zuletzt berechnete primale Lösung des Linearen Programms entspricht der Lösung für den Nucleolus und wird zurückgegeben.

```
setMethod(                                                          #1
        "checkAbort",                                               #2
        signature="NucleolusBase",                                  #3
        definition=function(.Object,oldObjValue){                   #4
                retVal=FALSE                                        #5
                lpCoopGameUtils<-.Object@LPCoopGameUtils            #6
                if((!is.null(oldObjValue))){                        #7
                        if(oldObjValue==getLPObjVal(lpCoopGameUtils)){  #8
                                retVal=TRUE                         #9
                        }                                           #10
                }elseif(!isLPFeasible(lpCoopGameUtils)){            #11
                        stop("No Solution exists for this game.")   #12
                }                                                   #13
                return(retVal)                                      #14
        }                                                           #15
)                                                                   #16
```

Codebsp. 45: checkAbort Methode

Zur Bestimmung, welche Einträge zu aktualisieren sind, dient die `getLPDualSolutionPos` Methode (siehe Codebsp. 46). Sie wird hierbei sowohl in der `updateLPMatrix` (siehe Codebsp. 47) als auch in der `updateLPRows` Methode (siehe Codebsp. 48) verwendet und liefert die Indizes der Einträge zurück, die für die duale Lösung einen positiven Wert aufzeigen (siehe Zeile 5).

```
setMethod(                                                          #1
        "getLPDualSolutionPos",                                     #2
        signature="NucleolusBase",                                  #3
        definition=function(.Object){                               #4
                return(which(getLPDualSolution(.Object@LPCoopGameUtils)>0))  #5
        }                                                           #6
)                                                                   #7
```

Codebsp. 46: getLPDualSolutionPos

In `updateLPMatrix` (siehe Codebsp. 47) wird für die Nebenbedingungen, für welche jeweils eine optimale Lösung vorliegt (siehe Zeile 7), die zu minimierende Strukturvariable für den Überschuss auf den Wert '0' gesetzt.

```
setMethod(                                                                    #1
        "updateLPMatrix",                                                     #2
        signature="NucleolusBase",                                            #3
        definition=function(.Object){                                         #4
                lpCoopGameUtils<-.Object@LPCoopGameUtils                      #5
                excessCoefs=getMatrixLastCol(lpCoopGameUtils)                 #6
                pos=getLPDualSolutionPos(.Object)                             #7
                excessCoefs[pos]<-0                                           #8
                setMatrixLastCol(lpCoopGameUtils)<-excessCoefs                #9
                eval.parent(substitute(.Object@LPCoopGameUtils<-lpCoopGameUtils)) #10
        }                                                                     #11
)                                                                             #12
```

Codebsp. 47: updateLPMatrix Methode

Neben der Anpassung der Koeffizienmatrix werden auch die Schrankenwerte und -typen für die Nebenbedingungen durch `updateLPRows` (siehe Codebsp. 48) aktualisiert, für die durch `getLPDualsSolutionPos` (siehe Zeile 9) identifizierten Einträge.

Die identifizierten Nebenbedingungen werden als Gleichheitsbedingungen typisiert (siehe Zeile 14).

Für die Festlegung des Schrankenwertes wird für jede betreffende Nebenbedingung die `getLPRowsBoundsFunc` Methode aufgerufen (siehe Zeile 12) und der entsprechende Wert bestimmt.

```
setMethod(                                                                    #1
        "updateLPRows",                                                       #2
        signature="NucleolusBase",                                            #3
        definition=function(.Object){                                         #4
                lpCoopGameUtils<-.Object@LPCoopGameUtils                      #5
                rlb=getRlb(lpCoopGameUtils)                                   #6
                rub=getRub(lpCoopGameUtils)                                   #7
                rtype=getRtype(lpCoopGameUtils)                               #8
                pos=getLPDualSolutionPos(.Object)                             #9
                coeffMat=getMatrix(lpCoopGameUtils)[pos,,drop=FALSE]          #10
                rlb[pos]<-rub[pos]<-apply(                                    #11
                    coeffMat,1,FUN=getLPRowsBoundsFunc,.Object=.Object,pos=pos #12
                )                                                             #13
                rtype[pos]<-GLP_FX                                            #14
                setRlb(lpCoopGameUtils)<-rlb                                  #15
                setRub(lpCoopGameUtils)<-rub                                  #16
                setRtype(lpCoopGameUtils)<-rtype                              #17
                eval.parent(substitute(.Object@LPCoopGameUtils<-lpCoopGameUtils)) #18
        }                                                                     #19
)                                                                             #20
```

Codebsp. 48: updateLPRows Methode

Im Fall des Nucleolus bestimmt die `getLPRowsBoundsFunc` Methode den unteren und oberen Schrankenwert für die Nebenbedingungen.

```
setMethod(                                                              #1
        "getLPRowsBoundsFunc",                                          #2
        signature="NucleolusBase",                                      #3
        definition=function(.Object,x,pos){                             #4
                primal=getLPPrimalSolution(.Object@LPCoopGameUtils)     #5
                n=length(primal)                                        #6
                primal=primal[-n]                                       #7
                return(sum(x[-n]  * primal[-n]))                        #8
        }                                                               #9
)                                                                       #10
```

Codebsp. 49: getLPRowsBoundsFunc Methode

Im Folgenden werden die Implementierungen der anderen mit dem Nucleolus verwandten Lösungskonzepte thematisiert, wobei hier nur eine Hervorhebung der jeweiligen Unterschiede erfolgt.

5.4.2 Implementierung des Prenucleolus

Die Reimplementierung des Prenucleolus (siehe Abschnitt 4.2.1) erfolgte unter Berücksichtigung des Orginalcodes von GEBELE (2016).

Der Prenucleolus berücksichtigt – anders als der Nucleolus – nicht die Imputations- sondern die Präimputationsmenge als zulässigen Auszahlungsraum. Dies stellt den einzigen Unterschied zwischen beiden Konzepten dar.

Daher musste lediglich die Methode `initLPBndsObjCoefs` (siehe Codebsp. 50) überschreiben werden, um die Schrankenwerte für die Entscheidungsvariablen entsprechend der Präimputationsmenge als Auszahlungsraum zu setzen.

```
setMethod(                                                              #1
        "initLPBndsObjCoefs",                                           #2
        signature="PreNucleolus",                                       #3
        definition=function(.Object){                                   #4
                lpCoopGameUtils<-.Object@LPCoopGameUtils                #5
                n=getNumberOfPlayers(.Object@A)                         #6
                setClb(lpCoopGameUtils)<-c(rep(0,n),-Inf) #changed      #7
                setCub(lpCoopGameUtils)<-rep(Inf,n+1)                   #8
                setCtype(lpCoopGameUtils)<-c(rep(GLP_DB, n),GLP_FR) #changed  #9
                setObj(lpCoopGameUtils)<-c(rep(0,n),1)                  #10
                eval.parent(substitute(.Object@LPCoopGameUtils<-lpCoopGameUtils))  #11
        }                                                               #12
)                                                                       #13
```

Codebsp. 50: initLPBndsObjCoefs Methode überschrieben durch Klasse PreNucleolus

Waren die Restriktionen in der Nucleolus Implementierung noch durch die geforderte Eigenschaft der individuellen Rationalität für Imputationen nach unten hin beschränkt, fällt die Forderung mit dem größer gewordenen Geltungsbereich für die Präimputationsmenge weg (siehe Zeile 7). Die Zeilen 7 und 8 weisen hierbei Unterschiede zu den ursprünglichen Schrankenwerten von GEBELE (2015) für den Prenucleolus auf. Hier wird die Forderung einer Auszah-

lung nach Nichtnegativität umgesetzt, als Lösung liefert der Prenucleolus somit nach Änderung eine Präimputation, welche die Nichtnegativitätsbedingung für die Auszahlung wahrt.

5.4.3 Implementierung des Per Capita Nucleolus

Der Per Capita Nucleolus (siehe Abschnitt 4.2.3) normiert den Nucleolus durch Division des Überschusses mit der in der jeweiligen Koalition beteiligten Anzahl von Spielern.

Hier reicht es aus, `determineExcessCoefficients` für die Klasse PerCapitaNucleolus zu überschreiben, so dass gleichbedeutend, die Spaltenwerte der zu minimierenden Strukturvariable die entsprechende Anzahl der in der jeweiligen Koalition beteiligten Spieler als Koeffizienten zugewiesen bekommt (siehe Zeile 8).

```
setMethod(                                                          #1
        "determineExcessCoefficients",                              #2
        signature="PerCapitaNucleolus",                             #3
        definition=function(.Object){                               #4
                N=length(.Object@A)                                 #5
                n=log2(N+1)                                         #6
                coeffMat<-createBitMatrix(n)                        #7
                cardS=sapply(1:N,function(ix){sum(coeffMat[ix,1:n])}) #8
                return(c(cardS[-N],0))                              #9
        }                                                           #10
)                                                                   #11
```

Codebsp. 51: determineExcessCoefficients Methode überschrieben durch Klasse PerCapitaNucleolus

Ansonsten sind hier keinerlei Änderung vorzunehmen.

5.4.4 Implementierung des Proportional Nucleolus

Für die Implementierung des Proportional Nucleolus ist es ausreichend, die `determineExcessCoefficients` Methode zu überschreiben. Nach Umformung der ersten Nebenbedingung des initialen Linearen Programms (siehe Formel 97) ergeben sich für die Spaltenwerte der zu minimierende Strukturvariable die des Spielvektors. Eine Ausnahme stellt die Nebenbedingung für die große Koalition dar (wegen $S \subset N$ in erster Nebenbedingung der Formel 97), hier erhält die zu minimierende Strukturvariable den Wert 0 zugewiesen (siehe Forderung nach Effizienz in zweiter Nebenbedingung der Formel 97).

```
setMethod(                                                          #1
        "determineExcessCoefficients",                              #2
        signature="ProportionalNucleolus",                          #3
        definition=function(.Object){                               #4
                N=length(.Object@A)                                 #5
                return(c(.Object@A[-N],0))                          #6
        }                                                           #7
)                                                                   #8
```

Codebsp. 52: determineExcessCoefficients Methode überschrieben durch Klasse ProportionalNucleolus

5.4.5 Implementierung des Modiclus

Als sehr umfangreich erweisen sich die Unterschiede zwischen der Implementierung des Modiclus und der des Nucleolus.

Der Modiclus minimiert lexikographisch im Gegensatz zum Nucleolus nicht den maximalen Wert für die Überschüsse sondern deren Differenzen; so reicht ein lediglliches Überschreiben der `determineExcessCoefficients` nicht mehr aus. Zusätzlich erhöht sich die Anzahl der betrachteten Nebenbedingungen um ein Vielfaches (siehe Abschnitt 4.2.4.1). Die gesamte Koeffizientenmatrix und die zugehörige rechte Spalte des initialen Linearen Programms (siehe Formel 104) muss angepasst werden. Hierzu werden die beiden Methoden `initLPMatrix` sowie `initLPRows` durch die Klasse `Modiclus` überschrieben.

Codebsp. 53 zeigt die überschriebene Methode `initLPMatrix`, die der Initialisierung der Koeffizientenmatrix dient.

```
setMethod(                                                                  #1
        "initLPMatrix",                                                     #2
        signature="Modiclus",                                               #3
        definition=function(.Object){                                       #4
                lpCoopGameUtils<-.Object@LPCoopGameUtils                    #5
                n<-getNumberOfPlayers(.Object@A)                            #6
                N<-length(.Object@A)                                        #7
                tempBM<-as.data.frame(createBitMatrix(n))                   #8
                lpMatrix<-matrix(ncol=(n+1),nrow=0)                         #9
                for(i in 1:(nrow(tempBM)-1)){                               #10
                        currEntry=tempBM[i,]                                #11
                        corrEntries=tempBM[c(-i,-N),]                       #12
                        matTemp=matrix(                                     #13
                                unlist(                                     #14
                                        apply(corrEntries,1,FUN=function(x){ #15
                                                currEntry-x                 #16
                                        })                                  #17
                                ),                                          #18
                                ncol=(n+1),                                 #19
                                byrow=TRUE                                  #20
                        )                                                   #21
                        lpMatrix=rbind(lpMatrix,matTemp)                    #22
                }                                                           #23
                lpMatrix[,(n+1)]=1                                          #24
                lpMatrix=rbind(lpMatrix,c(rep(1,n),0))                      #25
                colnames(lpMatrix)[(n+1)]<-"cVal"                           #26
                setMatrix(lpCoopGameUtils)<-lpMatrix                        #27
                eval.parent(substitute(.Object@LPCoopGameUtils<-lpCoopGameUtils)) #28
        }                                                                   #29
)                                                                           #30
```

Codebsp. 53: initLPMatrix Methode überschrieben durch Klasse Modiclus

Die Initialisierung der rechten Seite des Linearen Programms wird durch das Überschreiben der `initLPRows` Methode bewerkstelligt (siehe Codebsp. 54).

```
setMethod(                                                                  #1
        "initLPRows",                                                       #2
        signature="Modiclus",                                               #3
        definition=function(.Object){                                       #4
                lpCoopGameUtils<-.Object@LPCoopGameUtils                    #5
                N<-length(.Object@A)                                        #6
                n<-getNumberOfPlayers(.Object@A)                            #7
                A<-.Object@A                                                #8
                rlb<-c()                                                    #9
                N=length(A)                                                 #10
                for(i in 1:(N-1)){                                          #11
                        valuesOfT=A[c(-i,-N)]                               #12
                        rlbValues=sapply(valuesOfT,function(x){A[i]-x})     #13
                        rlb<-c(rlb,rlbValues)                               #14
                }                                                           #15
                rlb<-c(rlb,A[N])                                            #16
                setRlb(lpCoopGameUtils)<-rlb                                #17
                setRub(lpCoopGameUtils)<-c(rep(Inf,length(rlb)-1),.Object@A[N])  #18
                setRtype(lpCoopGameUtils)<-c(rep(GLP_LO,length(rlb)-1),GLP_FX)   #19
                eval.parent(substitute(.Object@LPCoopGameUtils<-lpCoopGameUtils))  #20
        }                                                                   #21
)                                                                           #22
```

Codebsp. 54: initLPRows Methode überschrieben durch Klasse Modiclus

5.4.6 Implementierung des Simplified Modiclus

Genau wie der Modiclus minimiert der Simplified Modiclus lexikographisch die maximalen Differenzen der Überschüsse. Im Gegensatz zum Modiclus werden aber nicht alle Differenzen der Überschüsse betrachtet, sondern nur der zu den Komplementkoalitionen. Die Anzahl der Nebenbedingungen stimmt somit mit der des Nucleolus überein.

Auch hier werden die beiden Methoden `initLPMatrix` sowie `initLPRows` überschrieben.

Codebsp. 55 zeigt die überschriebene Methode `initLPMatrix`, die der Initialisierung der Koeffizientenmatrix dient.

```
setMethod(                                                                  #1
        "initLPMatrix",                                                     #2
        signature="SimplifiedModiclus",                                     #3
        definition=function(.Object){                                       #4
        lpCoopGameUtils<-.Object@LPCoopGameUtils                            #5
        n=getNumberOfPlayers(.Object@A)                                     #6
        N=length(.Object@A)                                                 #7
        lpMatrix=createBitMatrix(n)                                         #8
        lpMatrix[lpMatrix==0]=-1                                            #9
        lpMatrix[,(n+1)]=1                                                  #10
        lpMatrix[N,(n+1)]=0                                                 #11
        setMatrix(lpCoopGameUtils)<-lpMatrix                                #12
        eval.parent(substitute(.Object@LPCoopGameUtils<-lpCoopGameUtils))   #13
        }                                                                   #14
)                                                                           #15
```

Codebsp. 55: initLPMatrix Methode überschrieben durch Klasse SimplifiedModiclus

Die Initialisierung der rechten Seite des Linearen Programms wird durch das Überschreiben der `initLPRows` Methode erreicht (siehe Codebsp. 56).

```
setMethod(                                                                  #1
        "initLPRows",                                                       #2
        signature="SimplifiedModiclus",                                     #3
        definition=function(.Object){                                       #4
                lpCoopGameUtils<-.Object@LPCoopGameUtils                    #5
                N<-length(.Object@A)                                        #6
                n<-getNumberOfPlayers(.Object@A)                            #7
                rlb<-sapply(1:(N-1),FUN=function(ix){A[ix]-A[N-ix]})        #8
                rlb<-c(rlb,A[N])                                            #9
                setRlb(lpCoopGameUtils)<-rlb                                #10
                setRub(lpCoopGameUtils)<-c(rep(Inf,length(rlb)-1),.Object@A[N])  #11
                setRtype(lpCoopGameUtils)<-c(rep(GLP_LO,length(rlb)-1),GLP_FX)   #12
                eval.parent(substitute(.Object@LPCoopGameUtils<-lpCoopGameUtils))  #13
        }                                                                   #14
)                                                                           #15
```

Codebsp. 56: initLPRows Methode überschrieben durch Klasse SimplifiedModiclus

5.4.7 Implementierung des DisruptionNucleolus

Der Disruption Nucleolus ist das Lösungskonzept, welches die maximalen *Propensities To Disrupt* für alle Koalitionen lexikographisch minimiert. Da das initiale Programm zur Minimierung der maximalen *Propensity To Disrupt* (siehe Formel 111) nichtlinear ist, wird zur Bestimmung das äquivalente Lineare Programm (siehe Formel 112) herangezogen und die Methode `determineExcessCoefficients` überschrieben. Die Zuweisung der Koeffizienten für die zu minimierende Strukturvariable entspricht hierbei dem Ausdruck $[v(N) - v(S) - v(N - S)]$ aus der ersten Bedingung von Formel 112.

```
setMethod(                                                                  #1
        "determineExcessCoefficients",                                      #2
        signature="DisruptionNucleolus",                                    #3
        definition=function(.Object){                                       #4
                A=.Object@A                                                 #5
                N=length(A)                                                 #6
                tfac=sapply(1:((N-1)/2),function(ix){A[N]-A[ix]-A[N-ix]})   #7
                tfac=c(tfac,rev(tfac),0)                                    #8
                return(tfac)                                                #9
        }                                                                   #10
)                                                                           #11
```

Codebsp. 57: determineExcessCoefficients Methode überschrieben durch Klasse DisruptionNucleolus

Da die zu minimierende Entscheidungsvariable t aus dem äquivalenten Linearen Programm zusätzlich auf den Bereich $-1 < t < 0$ beschränkt ist, muss zusätzlich die Methode `initLPBndsObjCoefs` überschrieben werden. In Codebsp. 58 wird t entsprechend beschränkt.

```
setMethod(                                                                  #1
        "initLPBndsObjCoefs",                                               #2
        signature="DisruptionNucleolus",                                    #3
        definition=function(.Object){                                       #4
                lpCoopGameUtils<-.Object@LPCoopGameUtils                    #5
                n=getNumberOfPlayers(.Object@A)                             #6
                setClb(lpCoopGameUtils)<-c(rep(0,n),-1+1e-16)               #7
                setCub(lpCoopGameUtils)<-c(rep(Inf,n),-1e-16)               #8
                setCtype(lpCoopGameUtils)<-c(rep(GLP_LO,n),GLP_DB)          #9
                setObj(lpCoopGameUtils)<-c(rep(0,n),1)                      #10
                eval.parent(substitute(.Object@LPCoopGameUtils<-lpCoopGameUtils))  #11
        }                                                                   #12
)                                                                           #13
```

Codebsp. 58: initLPBndsObjCoefs Methode überschrieben durch Klasse DisruptionNucleolus

5.4.8 Implementierung des Lexical Gately Punkt

Wie der Disruption Nucleolus minimiert auch der Lexical Gately Punkt lexikographisch die maximalen *Propensities To Disrupt*, nur nicht unter Rücksichtnahme aller Koalitionen. Aus diesem Grund wird für die Klasse LexicalGatelyValue auch DisruptionNucleolus als Elternklasse angegeben.

Der Lexical Gately Punkt beschränkt sich als eindeutige Variante des Gately Punkt nur auf die Einerkoalitionen, daher muss lediglich die Methode `determineExcessCoefficients` überschrieben werden (siehe Codebsp. 59).

```
setMethod(                                                                  #1
        "determineExcessCoefficients",                                      #2
        signature="LexicalGatelyValue",                                     #3
        definition=function(.Object){                                       #4
                A=.Object@A                                                 #5
                n=getNumberOfPlayers(A)                                     #6
                N=length(A)                                                 #7
                tfac=sapply(1:n,function(ix){A[N]-A[ix]-A[N-ix]})           #8
                tfac=c(tfac,0)                                              #9
                return(tfac)                                                #10
        }                                                                   #11
)                                                                           #12
```

Codebsp. 59: determineExcessCoefficients Methode überschrieben durch Klasse LexicalGatelyValue

6 Vorbereitung von CoopGame auf CRAN

In diesem Abschnitt wird ein kurzer Eindruck über die Schritte und Überlegungen vermittelt, die durchgeführt wurden, um einen geeigneten Rahmen für eine zeitnahe Veröffentlichung von `CoopGame` (sowie im Anschluss von `CommunicationGames` und `PartitionGames`) auf CRAN zu ermöglichen.

Hierzu werden einführend die Programmiersprache R und die Veröffentlichungsplattform CRAN vorgestellt, sowie auf die Entstehungsgeschichte der drei oben genannten Pakete mit einer Würdigung der am Zustandekommen beteiligten Personen eingegangen

Die Vorstellung des Pakets AuxiliaryTools stellt den Übergang hinsichtlich der für dieVeröffentlichung vorgenommenen Schritte dar. Außer in der Dokumentation, erleichtert das Paket auch Standardaufgaben zur Herstellung der Benutzerfreundlichkeit, wie bei den Parameter Checks.

Abschließend werden die in `CoopGame` eingepflegten Komponententests beleuchtet.

6.1 Programmiersprache R

R war zwar lange Zeit eine relativ unbekannte Sprache ihr Bekanntheitsgrad hat sich aber in den letzten Jahren deutlich gesteigert. Das liegt nicht zuletzt daran, dass sie gegenüber anderen Programmiersprachen wie Java oder C# in der Datenverarbeitung gewisse Vorzüge aufweist.

Bei R handelt es sich um eine freie und kostenlose Umgebung sowie Programmiersprache, die von Beginn an auf statistische Berechnungen und Datenanalyse ausgelegt wurde (THE R FOUNDATION 2017).

Des Weiteren bietet R eine zusätzliche Kommando-Schnittstelle, die es auf einfachste Weise erlaubt, Code interaktiver und flexibler zu verwenden als bei vielen vergleichbare Sprachen. R kann sowohl interaktiv, also im Einzelbefehlsmodus, als auch als Skriptsprache und im Batch-Modus verwendet werden. Neben schnellem Prototyping und dem funktionsorientierten Aufbau, stellt R dem Nutzer viele bereits integrierte Funktionalitäten bereit, um Daten auf einfachste Weise handhaben zu können, so dass es sich auch ideal für Data Mining eignet. So wurde R in den Jahren 2014 bis 2016 als Sprache mit dem höchsten Nutzungsanteil im Bereich Data Science nach Befragung von über 2900 Daten Analytikern sowie Wissenschaftlern im KDNuggets Ranking ermittelt (SMITH 2017).

6.2 Plattform CRAN

Der CRAN steht für *„The **C**omprehensive **R** **A**rchive **N**etwork“* und ist ein Netzwerk aus auf der ganzen Welt verteilten FTP und Web Servern, die identischen und aktuellen R-Code sowie Dokumentationen öffentlich bereitstellen.

Jeder Entwickler ist dazu angehalten, seinen Code auf dieser Plattform frei nach GNU GPL zu veröffentlichen. So sind derzeitig (Stand 11.07.2017) 11021 R Packages auf CRAN frei verfügbar.

Dabei gilt es dennoch gewisse Qualitätsstandards bei Veröffentlichung eines eigenen Pakets zu wahren. Die CRAN REPOSITORY MAINTAINERS (2017) fordern bestimmte Qualitätskriterien in Form einer CRAN Repository Policy ein, deren Einhaltung vor Paketveröffentlichung automatisiert überprüft wird. So wird sichergestellt, dass der Code auf CRAN einem gewissen Qualitätsstandard genügt.

6.3 Das Paket CoopGame

Das Paket `CoopGame` ist aus einem Projekt zur kooperativen Spieltheorie im Master Angewandte Informatik unter der Leitung von Prof. Dr. Staudacher und den Teilnehmern Anna Merkle sowie Kuebra und Fatma Tokay im Sommersemester 2015 heraus entstanden.

Orientierten sich in den Anfängen die umgesetzten Konzepte teilweise noch relativ stark bezüglich Logik und Struktur an denen von `TUGlab` (MIGUEL & RODRIGUEZ 2006**)**, konnten diese aber im Zuge des Folgeprojekts im Wintersemester 2015/2016 (Teilnehmer: Alexandra Tiukkel, Michael März, Johannes Anwander) weitestgehend abgelöst und um weitere ergänzt werden. Ungefähr zeitgleich fanden die Konzepte des Nucleolus sowie des Prenucleolus als Ergebnis der Bachelorarbeit von GEBELE (2015) Eingang in das Paket. Diese Bachelorarbeit schuf darüber hinaus den Grundstein für das im Rahmen dieser Masterarbeit entstandene Framework zur Berechnung weiterer Nucleolus Derivate. Das Rahmenwerk orientierte sich hierbei an den Umsetzungen von GEBELE (2015) und löste diese später ab.

Mit der Bachelorarbeit von Franz Müller im Wintersemester 2016/2017 konnten auch die bisherigen mengenbasierten Ansätze – mit Anlehnung an den Umsetzungen aus `TUGlab` – vollständig abgelöst werden, es liegen somit keine Überschneidungen mit `TUGlab` in `CoopGame` mehr vor.

Aus den Masterprojekten und den Abschlussarbeiten gingen unter anderem folgende Konzepte für `CoopGame` hervor:

Koalitionsfunktionen für Spiele: Abstimmungsspiel (`cFuncQuota`), Kostenspiel (`cFuncCostSharing`), Handschuhspiel (`cFuncGlove`) und vieler weitere

Spieleeigenschaften: Additivität (`isAdditiveGame`), Zulässigkeit (`isAdmissibleGame`), Balanciertheit (`isBalanced`), Konvexität (`isBalanced`), Wesentlichkeit (`isEssentialGame`), Superadditivität (`isSuperadditiveGame`), Schwache Additivität (`isWeaklyAdditiveGame`)

Eigenschaften von Aufteilungen: Imputation (`isImputation`), Kern (`belongToCore`)

Bestimmung Ergebnisraum bzw. mengenwertige Lösungskonzepte: Imputationsmenge (`imputationsetVertices`), Selektop (`selectope3Vertices`), Weberset (`websersetVertices`), Epsilon Kern (`epsilonCoreVertices`), Kern (`coreVertices`)

Punktwertiger Lösungskonzepte (nicht aufgeführt deren Machtindizes): Shapley Wert (`shapleyValue`), $\tau - Wert$ (`tauValue`), Public Help Wert (`publicHelpValue`), Public Good Wert (`publicGoodValue`), Johnston Wert (`johnstonValue`), Deegan-Packel Wert (`deeganPackelValue`), Gately Punkt (`gatelyValue`)

Nucleolus Derivate: Nucleolus (`nucleolus`), Prenucleolus (`prenucleolus`), Disruption Nucleolus (`disruptionNucleolus`), Per Capita Nucleolus (`perCapitaNucleolus`), Modiclus (`modiclus`), Simplified Modiclus (`simplifiedModiclus`), Proportional Nucleolus (`proportionalNucleolus`)

6.4 Weitere R-Pakete

Neben `CoopGame` gingen noch drei weitere Pakete aus den Projekten als auch aus Abschlussarbeiten hervor. Bei diesen drei Paketen handelt es sich um `CommunicationGames`, `PartitionGames` sowie AuxiliaryTools.

Die beiden Pakete `PartitionGames` und `CommunicationGames` dienten hierbei zur besseren Abgrenzung zwischen der reinen kooperativen Spieltheorie und partitiven sowie netzwerkbasierten kooperativen Ansätzen und sollen nach `CoopGame` auf CRAN veröffentlicht werden.

Mit dem dritten Paket AuxiliaryTools wurde dagegen ein Paket geschaffen, das nicht für die Veröffentlichung auf CRAN vorgesehen ist, aber Funktionalitäten für eine zentrale Pflege von `CoopGame` und die anderen Pakete bereitstellt.

6.4.1 CommunicationGames

Bei der Umsetzung von kooperativen Konzepten im Rahmen des Projekts im Wintersemester 2015/2016 (Teilnehmer: Alexandra Tiukkel, Michael März, Johannes Anwander) fiel die Entscheidung, dass es sinnvoll ist, kooperative Spiele mit zugrundeliegender Netzwerkstruktur in Form eines eigenen Subpakets von den übrigen Konzepten in `CoopGame` stärker abzuheben. Aus diesem Grund fiel die Entscheidung zu Gunsten der Erstellung des neuen Pakets `CommunicationGames`. Daniel Gebele trug im Zuge seiner Bachelorarbeit (GEBELE 2015) zu neuen Konzepten mit Netzwerkstruktur bei, die auf dem Nucleolus und Prenucleolus aufbauten.

Das Projektteam aus dem Wintersemester 2016/2017 – mit den Mitgliedern Patrick Köpfler, Samuel Claeys, Bastian Pätzold, Arthur Kowalski und Mario Brunner – trug ebenfalls seinen Teil dazu bei, das Paket um neue netzwerkbasierter Konzepte zu erweitern.

In dem Einmannprojekt im Sommersemester 2017 überprüfte Lucas Greising vorwiegend die vorhandenen Konzepte und erweiterte aber auch das Paket um neue Ansätze.

6.4.2 PartitionGames

Im Laufe der Bachelorarbeit von Nicole Cyl im Wintersemester 2016/2017 wurde die Entscheidung gefällt, auch der Untergruppe partitiver Ansätze in der kooperativen Spieltheorie ein eigenes Paket namens `PartitionGames` zu widmen. Partitive Ansätze aus den Vorprojekten wurden entsprechend in dieses verschoben.

Im Weiteren wurden auch neue Konzepte durch das Projektteam aus dem Wintersemester 2016/2017 – mit den Mitgliedern Patrick Köpfler, Samuel Claeys, Bastian Pätzold, Arthur Kowalski und Mario Brunner – beigesteuert.

6.4.3 AuxiliaryTools

Die Fehlercodes sowie die Fehlermeldungen dienen dem Nutzer als Anhaltspunkt, welche Parameter bei Funktionsaufruf eventuell falsch verwendet wurden und tragen somit direkt zur besseren Benutzerfreundlichkeit mit bei.

Die Zielsetzung des Pakets `AuxiliaryTools` war es, die Entwicklung der Pakete `CoopGame`, `Communication` Games und `PartitionGames` zu vereinfachen. In erster Linie sollte es helfen, Zeit einzusparen, die vor Bestehen des Pakets benötigt wurde, dieselben Fehlercodes mit Beschreibungen sowohl im Programmcode als auch in der Dokumentation sowie einer zusätzlichen behelfsmäßigen Excel-Datei zu pflegen.

Durch die Einführung des Hilfspakets `AuxiliaryTools` erfolgte fortan die Pflege zentral nur noch an einer Stelle. Wie die Fehlercodes und -nachrichten dadurch innerhalb der Dokumentation und in den Parameterchecks genutzt werden können, wird in den entsprechenden Abschnitten 6.5 und 6.6 erörtert.

Im Folgenden wird anhand eines Szenarios aufgezeigt, wie mit `AuxiliaryTools` die Voraussetzungen bezüglich der Fehlercodes für Dokumentation und Benutzung im Programmcode geschaffen wurden.

Zu Beginn kann einmalig über die Funktion `exportEmptyErrFile` eine leere CSV-Datei `errorcodes.csv` zur Verwaltung der Fehlercodes im Verzeichnis für externe Ressourcen `./inst/extdata` angelegt werden.

Der Nummernbereich für die Fehler wurde in diesem Beispiel auf 1000 bis 2000 gesetzt.

```
> setwd("./CoopGame")                                                    #1
> exportEmptyErrFile("errorcodes.csv", errCodesStart = 1000, errCodesEnd = 2000)   #2
```

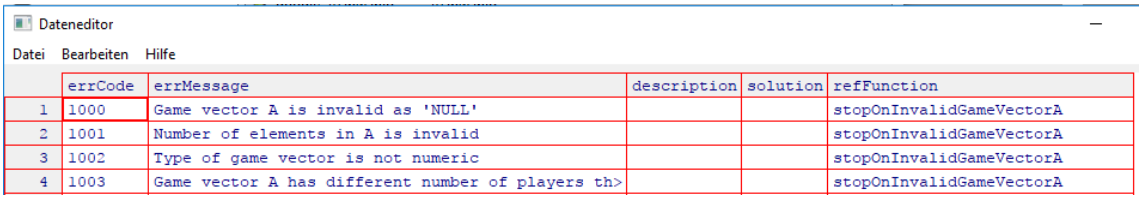
Codebsp. 60: Anlegen leerer CSV-Datei für Verwaltung der Fehlercodes

Über den Aufruf von `maintainCSVDataWithFix` bzw. über ein externes Tabellenkalkulationsprogramm wie Excel kann anschließend mit der Pflege der Fehlercodes begonnen werden.

```
> setwd("./CoopGame")                                                    #1
> maintainCSVDataWithFix()                                               #2
```

Codebsp. 61: Pflege von Fehlercodes

Für jeden Fehlercode (siehe Abbildung 3) kann hier ein Feld für die Fehlermeldung (`errMessage`), die Fehlerbeschreibung (`description`), die Fehlerbehebung (`solution`) und einer Referenz auf die Parameter Checkmethode (`refFunction`) gepflegt werden.



	errCode	errMessage	description	solution	refFunction
1	1000	Game vector A is invalid as 'NULL'			stopOnInvalidGameVectorA
2	1001	Number of elements in A is invalid			stopOnInvalidGameVectorA
3	1002	Type of game vector is not numeric			stopOnInvalidGameVectorA
4	1003	Game vector A has different number of players th>			stopOnInvalidGameVectorA

Abbildung 3: Pflege von Fehlercodes über interne R fix Funktion

Sind alle Fehlermeldung eingepflegt, können über den Aufruf von `exportErrSysdata` die Fehlercode-Einträge und weitere Hilfsfunktionen in die Datei `./R/sysdata.rda` exportiert werden. Alle Objekte dieser Datei stehen nach dem Laden des Pakets zur ausschließlichen internen Verwendung innerhalb des Pakets zur Verfügung.

```
> setwd("./CoopGame")                                                    #1
> exportErrSysdata()                                                     #2
```

Codebsp. 62: Export Objekte in sysdata.rda

Die Objekte werden hierbei als Listenobjekt mit dem Namen `SYSDATA_OBJECTS` exportiert. Über dieses Objekt lässt sich auf die Fehlercode Einträge sowohl im Code als auch bei der automatischen Generierung der Dokumentation durch Roxygen zugreifen.

Codebsp. 63 zeigt zum einen den direkten Zugriff auf einen Fehlercode Eintrag (síehe Zeile 1), zum anderen den Abruf eines Eintrags mit der Hilfsfunktion `getSysdataErrorEntry` über dessen Identifikationsnummer (siehe Zeile 4).

```
Browse[2]> SYSDATA_OBJECTS$errorcodes[1,c("errCode","errMessage")]                #1
    errCode                              errMessage                               #2
1       1000 Game vector A is invalid as 'NULL'                                   #3
Browse[2]> SYSDATA_OBJECTS$getSysdataErrorEntry(1000)[c("errCode","errMessage")]#4
    errCode                              errMessage                               #5
1       1000 Game vector A is invalid as 'NULL'                                   #6
```

Codebsp. 63: Exemplarischer Zugriff auf Fehlercode Eintrag

Für die weiteren Einsatzmöglichkeiten des Pakets `AuxiliaryTools` sei auf dessen Paketdokumentation verwiesen.

6.5 Dokumentation mit Roxygen

Die Dokumentation von `CoopGame` (sowie der übrigen Pakete) wurde mittels dem R Paket `roxygen2` umgesetzt. Dies bot den großen Vorteil, dass Funktionen, Klassen und Methoden direkt im Programmcode von den Entwicklern dokumentiert werden können. Eine separate Entwicklerdokumentation, welche sich nur mit erhöhtem Aufwand – bei der Vielzahl von zwischenzeitlichen Änderungen – aktuell halten lässt, wird somit überflüssig. Die ganze Dokumentation wird hierbei automatisiert anhand der Roxygen Tags generiert. Als Referenz sei hier an erster Stelle auf WICKHAM (2015b) verwiesen.

Die Verwendung von Templates bei der Dokumentation schuf außerdem einen einheitlichen Rahmen.

Am Beispiel des Template `ParameterCheck` (siehe Codebsp. 65) und der Dokumentation zum Parameter-Check `stopOnInvalidGameVectorA` (siehe Codebsp. 64) soll der Einsatz von `roxygen2` zur Paketdokumentation vorgeführt werden.

In der Dokumentation des Parameter-Checks wird in Zeile 1 das Template `ParameterCheck` eingebunden. In Zeile 2 erfolgt zusätzlich die Übergabe der Template Variable `PARAMETER-CHECK_NAME` mit dem Wert `stopOnInvalidGameVectorA` an das Template `Parameter-Check`. Die anderen Roxygen-Tags sollen hier nicht weiter erörtert werden, hier sei erneut auf WICKHAM (2015b) verwiesen.

```
#' @template Templates/ParameterCheck                                    #1
#' @templateVar PARAMETERCHECK_NAME stopOnInvalidGameVectorA             #2
#' @description stopOnInvalidGameVectorA checks if game vector A         #3
#'  is specified as parameter correctly.                                  #4
#'  Validation result gets stored to object paramCheckResult              #5
#'  in case an error occured and causes stop otherwise.                   #6
#' @template author/JA                                                   #7
#' @export stopOnInvalidGameVectorA                                      #8
#' @examples                                                             #9
#'#Create empty object for check result:                                 #10
#'  paramCheckResult=getEmptyParamCheckResult()                          #11
#'#Define invalid game vector A (not numeric):                          #12
#'  A=c("0","0","0","60","60","60","72")                                 #13
#'#Check if game vector A is valid:                                      #14
#'  stopOnInvalidGameVectorA(paramCheckResult,A)                         #15
```

Codebsp. 64: Roxygen Dokumentation für stopOnInvalidGameVectorA Funktion

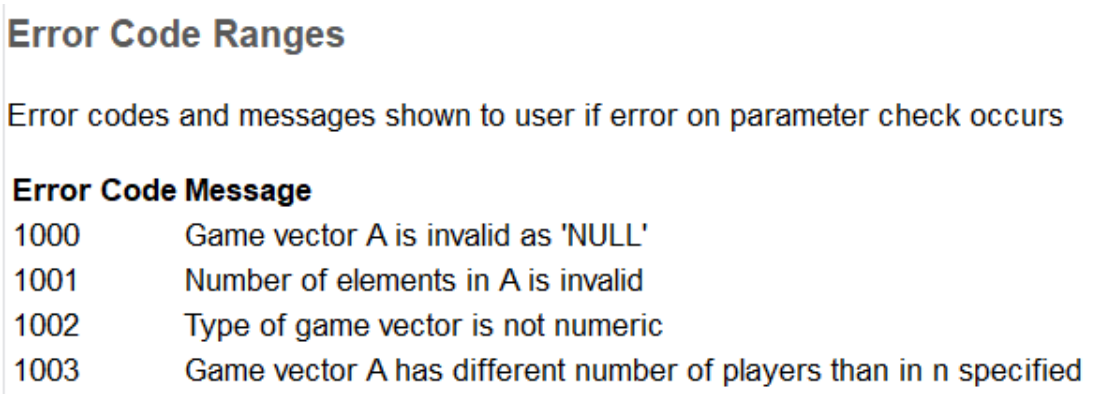
In dem eingebundenen Template `ParameterCheck` kann Code zur Ausführung durch `roxygen2` mitangegeben werden, der bei der Dokumentationserzeugung ausgeführt wird. Dies erfolgt in der Form `#' <% EXECUTE_CODE() %>`, wie beispielsweise in Zeile 23. Hier werden aus den Fehlercode-Einträgen des Pakets diejenigen selektiert, bei denen das Feld `refFunction` mit der Template Variable `PARAMETERCHECK_NAME` übereinstimmt (siehe hierzu Abschnitt 6.4.3). Innerhalb der Zeilen 24 bis 33 wird die tabellarische Ausgabe der betreffenden Fehlercode-Einträge vorbereitet und als String an die Variable OUTPUT_ERRORCODES übergeben.

In Zeile 44 erfolgt die Ausgabe des Inhalts der Variable OUTPUT_ERRORCODES.

```
#'<%#############################################################################################################################%>
#' <%# Rd-Files Roxygen-Template for Parameter Check Functions                                                                   %>
#' <%# Version: 1.0                                                                                                             %>
#' <%# Date: 20170107                                                                                                          %>
#' <%# Author: Johannes Anwander                                                                                               %>
<%#############################################################################################################################%>
#' <%#                                                                                                                         %>
#' <%# USAGE:                                                                                                                  %>
#' <%#  Include template by:                    #' @template Templates/ParameterChecks                                        %>
#' <%#  Include tempate variables by scheme:    #' @templateVar <VAR_NAME> <VAR_VALUE>                                        %>
#' <%#  Add description in source file:         #' @description <AUTHOR>                                                      %>
#' <%#  Add export in source file:              #' @export <PARAMETERCHECK_NAME>                                              %>
#' <%#  Add examples in source:                 #' @examples <EXAMPLES>                                                       %>
#' <%# Template Variables of this template:                                                                                   %>
#' <%#  PARAMETERCHECK_NAME - Name of the corresponding parameter check function                                             %>
#' <%#                                                                                                                         %>
#' <%#                                                                                                                         %>
#'<%#############################################################################################################################%>
#' <%#                                                                                                                         %>
#' <%#PREPARATION STEPS                                                                                                        %>
#' <%#                                                                                                                         %>
#' <%#SECTION for retrieving error codes for specific parameter check function and their number – START                        %>
#' <% errorObjects=subset(SYSDATA_OBJECTS$errorcodes,refFunction==PARAMETERCHECK_NAME & errMessage!="")                        %>
#' <% numberErrorObjects=nrow(errorObjects)                                                                                    %>
#' <%#SECTION for retrieving error codes for specific parameter check function and their number – END                          %>
#' <%#                                                                                                                         %>
#' <%#SECTION for generating variable 'OUTPUT_ERRORCODES' – START                                                              %>
#' <%#'OUTPUT_ERRORCODES' contains output string for table with the attributes                                                 %>
#' <% OUTPUT_ERRORCODES=""                                                                                                     %>
#' <% OUTPUT_ERRORCODES=paste(OUTPUT_ERRORCODES,"\\tabular{lll}{",sep="")                                                      %>
#' <% OUTPUT_ERRORCODES=paste(OUTPUT_ERRORCODES,"\\strong{Error Code} \\tab \\strong{Message} \\cr ",sep="")                   %>
#' <% for(i in 1:numberErrorObjects){ OUTPUT_ERRORCODES=paste(OUTPUT_ERRORCODES,errorObjects[i,]$errCode," \\tab ",errorObjects[i,]$errMessage," \\cr ",sep="")} %>
#' <% OUTPUT_ERRORCODES=paste(OUTPUT_ERRORCODES,"}",sep="")                                                                    %>
#' <%#SECTION for generating variable 'OUTPUT_ERRORCODES' – END                                                                %>
#' <%#                                                                                                                         %>
#' <%#############################################################################################################################%>
#'                                                                                                                              #37
#' <%#TEMPLATE OUTPUT %>                                                                                                       #37
#'                                                                                                                              #38
#' @title Parameter Function <%=PARAMETERCHECK_NAME%>                                                                          #39
#' @name <%=PARAMETERCHECK_NAME%>                                                                                              #40
#' @family ParameterChecks_CoopGame                                                                                            #41
#' @section Error Code Ranges:                                                                                                 #42
#' Error codes and messages shown to user if error on parameter check occurs                                                   #43
#'  <%= cat(OUTPUT_ERRORCODES) %>                                                                                               #44
```

Codebsp. 65: Roxygen Template ParameterCheck

Die durch `PARAMETERCHECK_NAME` und `refFunction` referenzierten Fehlercode-Einträge stehen somit in der Dokumentation, wie in Abbildung 4 dargestellt, dem Benutzer zur Verfügung.



Error Code Ranges

Error codes and messages shown to user if error on parameter check occurs

Error Code	Message
1000	Game vector A is invalid as 'NULL'
1001	Number of elements in A is invalid
1002	Type of game vector is not numeric
1003	Game vector A has different number of players than in n specified

Abbildung 4: Abschnitt mit Fehlercode Beschreibung der stopOnInvalidGameVectorA Funktion

Die Hilfe zur Funktion kann in R Studio über den Konsolenbefehl `?stopOnInvalidGame-VectorA` aufgerufen werden.

6.6 Parameter Checks

Um die Nutzerfreundlichkeit und die Übersichtlichkeit des Programmcodes zu steigern, werden in `CoopGame` (sowie in den anderen Paketen) vor Ausführung der eigentlichen Funktionslogik, die übergebenen Parameter überprüft.

Werden bei der Validierung Unstimmigkeiten festgestellt, kommt es zum Abbruch der Funktion. Da sich die übergebenen Parameter sehr ähneln, wird in fast jeder Funktion der Spielevektor A mitübergeben. Für die Parameter Checks wurde so ein Mechanismus geschaffen, dass dem Nutzer bei Fehlern immer dieselben Meldungen angezeigt werden.

Das Einpflegen der Fehlercode Einträge mit den Funktionalitäten des Pakets `Auxiliary-Tools` wurde bereits in Abschnitt 6.4.3 erörtert.

Es folgt hier eine Vorstellung der Parametercheckfunktion `stopOnInvalidGameVectorA` (siehe Codebsp. 67), über die das Konzept zur Validierung der Eingabeparameter näher vermittelt werden soll.

Für die Validierung des Spielevektors A wird `stopOnInvalidGameVectorA` mit dem Spielvektor A, der Spieleranzahl n und einem leeren Fehlerobjekt `paramCheckResult` an der zu prüfenden Stelle aufgerufen.

Das Fehlerobjekt wird hierbei vorab über `getEmptyParamCheckResult` erzeugt. Hier handelt es sich um ein Listenobjekt mit einem Element für den Fehlercode und einem weiteren für die Fehlermeldung. Der Parameter n ist nicht obligatorisch, er wird nur angegeben, falls eine ermittelte Spieleranzahl mit der vom Spielvektor implizierten Anzahl verglichen werden soll.

Im Codebsp. 66 werden zwei Testfälle für einen ungültigen Spielevektor aufgeführt.

```
> paramCheckResult=getEmptyParamCheckResult()                    #1
> A<-NULL                                                        #2
> stopOnInvalidGameVectorA(paramCheckResult,A)                   #3
Error in stopOnParamCheckError(paramCheckResult) :               #4
  Error Code 1000: Game vector A is invalid as 'NULL'            #5
                                                                 #6
> A<-1:8                                                         #7
> stopOnInvalidGameVectorA(paramCheckResult,A)                   #8
Error in stopOnParamCheckError(paramCheckResult) :               #9
  Error Code 1001: Number of elements in A is invalid            #10
```

Codebsp. 66: Parametertests für Spielevektor A

Für die Ausgabe der richtigen Fehlercodes werden die von AuxiliaryTools exportierten Funktionen und Fehlereinträge genutzt. Im Codebsp. 67 erfolgt in Zeile 6 eine Befüllung des Fehlerobjekts mit dem durch die ID 1000 spezifizierten Eintrag, *Game vector A is invalid as 'NULL'*.

```
stopOnInvalidGameVectorA=function(paramCheckResult,A,n=NULL){                #1
    checkResult=getEmptyParamCheckResult()                                   #2
    numberOfPlayersA=log2(length(A)+1)                                       #3
    #CheckifAisnull                                                          #4
    if(is.null(A)){                                                          #5
        SYSDATA_OBJECTS$fillParamCheckResult(checkResult,1000)               #6
        #Check if number of players deduced from A is valid integer          #7
    }elseif(numberOfPlayersA%%1!=0){                                         #8
        SYSDATA_OBJECTS$fillParamCheckResult(checkResult,1001)               #9
        #Check if game vector A isnumeric                                    #10
    }elseif(!is.numeric(A)){                                                 #11
        SYSDATA_OBJECTS$fillParamCheckResult(checkResult,1002)               #12
        #Check if n is specified if game vector A has n players              #13
    }elseif(!is.null(n)){                                                    #14
        if(numberOfPlayersA!=n){                                             #15
            SYSDATA_OBJECTS$fillParamCheckResult(checkResult,1003)           #16
        }                                                                    #17
    }                                                                        #18
    eval.parent(substitute(paramCheckResult<-checkResult))                   #19
    stopOnParamCheckError(paramCheckResult)                                  #20
}                                                                            #21
```

Codebsp. 67: Parametercheckfunktion stopOnInvalidGameVectorA

Das vorgestellte Beispiel sollte einen kurzen Einblick in die Validierungsmöglichkeiten von `CoopGame` vermitteln.

6.7 Unit-Tests

Um die korrekte Funktionalität aller implementierten Bestandteile und die Richtigkeit der Konzepte sicherzustellen, sind Komponententests für `CoopGame` unerlässlich. Getestet wird nach Möglichkeit, ob die Funktionen bei spezifiziertem Input auch den gewünschten Output liefern und die Beispiele aus der Literatur von `CoopGame` richtig gelöst werden. Da sich unter Umständen eine Änderung auf viele andere Bereiche auswirkt, ist es mit dem einmaligen Testen nach Ende der Entwicklung nicht getan. Die Tests müssen fest im Paket `CoopGame` hinterlegt sein, so dass nach jeder Änderung bzw. Ergänzung die Funktionalität immer wieder aufs Neue geprüft werden kann.

Zudem halten auch die CRAN REPOSITORY MAINTAINERS (2017) in ihrer CRAN Repository Policy dazu an, für die vom Nutzer aufrufbaren Funktionalitäten, Tests zu hinterlegen.

Insofern ist für `CoopGame` die Verwendung einer Test-Suite unabdinglich, um den Qualitätsstandards von CRAN zu genügen. Die Wahl fiel hier bezüglich der Umsetzung auf das von WICKHAM (2015b) empfohlene Paket `testthat`.

Im Weiteren folgt die Beschreibung eines Testfalls mit dem Nucleolus Beispiel aus Abschnitt 4.2.1.3, der mittels testthat realisiert worden ist. Die Ausführungen orientieren sich hier unter anderem an den Empfehlungen von WICKHAM (2015b).

Zu anfangs ist die Testumgebung mit dem Paket devtools über Aufruf von use_testthat zu initialisieren. Dies hat unter anderem zur Folge, dass die Ordnerstruktur `./tests/testthat` innerhalb des Pakets aufgebaut wird. In diesem erzeugten Ordner sind fortan alle Tests abzulegen.

Für `CoopGame` wurde die Konvention getroffen, dass alle Testdateien mit dem Präfix test gefolgt von der Testnummer und eines Suffixes in Form des Namens der zu testenden Funktionalität benannt sind. Entsprechend wird die Datei für die Nucleolus Tests als `test_67_Nucleolus.R` bezeichnet.

In Codebsp. 68 wird das Nucleolus-Beispiel mit dem Spielvektor $A = (2,6,5,15,1,18,14)$ getestet. Als Ergebnis wird der Lösungswert $(2,7,5)$ erwartet (siehe Zeile 7).

```
test_that("Check 67.4 - testing calculation of Nucleolus with A=(2,6,5,15,1,18,14)",{   #1
        if(boolSkip){                                                                    #2
           skip("Test was skipped")                                                      #3
        }                                                                                #4
        A<-c(2,6,5,15,1,18,14)                                                           #5
        expected_x=c(2,7,5)                                                              #6
        expect_equal(nucleolus(A,enableTermOutLP=F),expected_x)                          #7
})                                                                                       #8
```

Codebsp. 68: Testfall für Nucleolus Beispiel

Nach diesem Schema sind übrigen Tests aufgebaut, wobei das Paket `testthat` neben `expect_equal` (siehe Zeile 8) auch eine Vielzahl unterschiedlicher Testfunktionen bereitstellt.

7 Zusammenfassung und Ausblick

Ziel der angestrebten Masterarbeit war in erster Linie die Weiterentwicklung des R-Paketes `CoopGame` zur kooperativen Spieltheorie, für die eine zeitnahe Veröffentlichung auf CRAN folgen sollte. Hierfür wurde der geeignete Rahmen geschaffen. Besonderes Augenmerk lag dabei auf der Abstimmung aller bisherigen Entwicklungen, die in das Paket miteinflossen. Stellenweise erforderte dies, notwendige Modifikationen der bisherigen Strukturen für eine in sich stimmige Gesamtintegration durchzuführen, um auch den durch CRAN gestellten Qualitätsansprüchen Sorge zu tragen.

Des Weiteren wurden auch eigene Untersuchungen zur kooperativen Spieltheorie betrieben, die sich primär auf punktwertige Lösungskonzepte konzentrierten.

So wurden zahlreiche Nucleolus Derivate mit einem objektorientierten Ansatz umgesetzt. Für den Gately Punkt, der dabei ebenfalls in gewisser Weise als eine Form von Nucleolus Derivat anzusehen ist, konnte hierbei sogar die von GATELY (1974) implizierte Eindeutigkeit widerlegt werden. Das gab den Anstoß zu einer eigenen eindeutigen Umdeutung des Gately Punkts, welche die gewünschte Eindeutigkeit wiederherstellt.

Als Resultat dieser Masterarbeit zeichnet sich das Paket `CoopGame` nun durch eine breitgefächerte Auswahl von Konzepten aus der kooperativen Spieltheorie aus. Die implementierten Konzepte reichen hierbei von Machtindizes, Prüfung auf Spieleeigenschaften, Bestimmung punktwertiger bzw. mengenwertiger Lösungen bis hin zu Koalitionsfunktionen und anderen Hilfskonzepte zur Steigerung der Benutzerfreundlichkeit.

Mit AuxiliaryTools wurde zudem ein Tool geschaffen, mit dem sich vor Veröffentlichung eine einheitliche Dokumentation von `CoopGame` (sowie für die daraus abgeleiteten Pakete) realisieren lässt.

Nach erfolgter Integration der Ergebnisse aus der Bachelorarbeit von Franz Müller zur (Re-)Implementierung mengenbasierter Ansätze mit Hilfe des `rcdd` Pakets steht der Visualisierung der Konzepte – und somit letztendlich auch der zeitnahen Paketveröffentlichung von `CoopGame` sowie den weiteren Paketen im Anschluss – nichts mehr im Wege.

Inhaltsverzeichnis der beigelegten CD

Die der Masterarbeit beigelegte CD umfasst eine digitale Fassung der eingereichten Arbeit als PDF:

20170717_MAI_ANWANDER_Masterarbeit.pdf

Des Weiteren sind die R Pakete enthalten, bei denen im Rahmen der Masterarbeit mitgewirkt und eigene Beiträge eingebracht wurden:

- **CoopGame**
- **CommunicationGames**
- **PartitionGames**
- **AuxiliaryTools**

Zum anderen ist der CD eine kleine Ansammlung ausgewählter Skriptdateien beigefügt, wie die Skripte zur:

- Widerlegung der Korrektheit des Beispiels von Littlechild & Vaidya (1976) zum Disruption Nucleolus in Abschnitt 4.2.6.2 (Codebsp. 16):
  Codebsp16_Widerlegung_Disruption_Nucleolus_Beispiel_von_Littlechild_und_Vaidya_1976
- Veranschaulichung des Zusammenspiels von GLPK und glpkAPI zur Prüfung eines Spiels auf Balanciertheit in Abschnitt 5.3.1.2 (Codebsp. 32):
  Codebsp32_Zusammenspiel_von_GLPK_und_glpkAPI_zur_Pruefung_eines_Spiels_auf_Balanciertheit.R
- Prüfung auf gleiche Überschüsse bei den Lösungspunkten eines nichteindeutigen Nucleolus in Abschnitt 4.2.1.4 (Code war in Masterarbeit nicht aufgeführt):
  Skript_Pruefung_auf_gleiche_Ueberschuesse_bei_nichteindeutigem_Nucleolus_Abschnitt_4_2_1_4.R

Die Ordnerstruktur ist wie folgt:

```
CD
├── BEISPIEL_SKRIPTE
│   ├──Codebsp16_Widerlegung_Disruption_Nucleolus_Beispiel_von_Littlechild_und_Vaidya_1976.R
│   ├──Codebsp32_Zusammenspiel_von_GLPK_und_glpkAPI_zur_Pruefung_eines_Spiels_auf_Balanciertheit.R
│   └──Skript_Pruefung_auf_gleiche_Ueberschuesse_bei_nichteindeutigem_Nucleolus_Abschnitt_4_2_1_4.R
├── PAKETE
└── SCHRIFTLICHE_ARBEIT
```

Literaturverzeichnis

- **AMPL Optimization Inc. (2013):** ABOUT US. Online verfügbar unter http://ampl.com/about-us/.
- **Bate, D. (2003):** Converting Packages to S4. In: *R News* Volume 3/1, S. 6–8. Online verfügbar unter https://cran.r-project.org/doc/Rnews/Rnews_2003-1.pdf.
- **Bertini, C.; Gambarelli, G.; Stach, I. (2008):** A Public Help Index. In: Matthew Braham und Frank Steffen (Hg.): Power, Freedom, and Voting. Unter Mitarbeit von Manfred J. Holler. Berlin, Heidelberg: Springer-Verlag Berlin Heidelberg, S. 83–98.
- **Bilbao, J. M. (2000):** Cooperative Games on Combinatorial Structures. Boston, MA: Springer (Theory and Decision Library, Series C, 26). Online verfügbar unter http://dx.doi.org/10.1007/978-1-4615-4393-0.
- **Bondareva, O. N. (1963):** Some applications of linear programming methods to the theory of cooperative games. In: *Problemy kibernetiki* 10, S. 119–139. Online verfügbar unter http://www.princeton.edu/~erp/ERParchives/archivepdfs/WP0.pdf#page=82.
- **Borgwardt, K. H. (2001):** Optimierung Operations Research Spieltheorie. Mathematische Grundlagen. Basel, s.l.: Birkhäuser Basel. Online verfügbar unter http://dx.doi.org/10.1007/978-3-0348-8252-1.
- **Borgwardt, K. H. (2010):** Aufgabensammlung und Klausurentrainer zur Optimierung. Für die Bachelorausbildung in mathematischen Studiengängen. 1. Aufl. Wiesbaden: Vieweg+Teubner Verlag / GWV Fachverlage GmbH Wiesbaden. Online verfügbar unter http://dx.doi.org/10.1007/978-3-8348-9354-3.
- **Brams, X. Y.; Schotter, X. Y.; Schwödlauer, X. Y. (2013)**: Applied Game Theory. Proceedings of a Conference at the Institute for Advanced Studies, Vienna, June 13–16, 1978: Physica-Verlag HD. Online verfügbar unter https://books.google.de/books?id=H9yoCAAAQBAJ.
- **Branzei, R.; Dimitrov, D.; Tijs, S. (2008)**: Models in Cooperative Game Theory. Second Edition. Berlin, Heidelberg: Springer-Verlag Berlin Heidelberg. Online verfügbar unter http://dx.doi.org/10.1007/978-3-540-77954-4.
- **Cano-Berlanga, S.; Gimenez-Gomez, J. M.; Vilella, C. (2015):** Enjoying cooperative games. The R package GameTheory. In: *Working Paper No. 06; CREIP; Spain*.
- **CRAN Repository Maintainers (2017)**: CRAN Repository Policy. Online verfügbar unter https://cran.r-project.org/web/packages/policies.html#Source-packages, zuletzt aktualisiert am 04.07.2017, zuletzt geprüft am 11.07.2017.

- **Dantzig, G. (1987):** ORIGINS OF THE SIMPLEX METHOD. Technical Report. Stanford University. Online verfügbar unter http://www.dtic.mil/dtic/tr/fulltext/u2/a182708.pdf.
- **Driessen, T.; Tijs, S. H. (1983):** The t-value, the nucleolus and the core for a subclass of games. In: *Other publications TiSEM* (73fdfe73-c88c-4a9f-8ee7-cd3d26003ea7).
- **Driessen, T.; Tijs, S. H. (1985):** The τ-value, The core and semiconvex games. In: *International Journal of Game Theory* 14 (4), S. 229–247. DOI: 10.1007/BF01769310.
- **Eichhorn, W.; Gleißner, W. (2016):** Mathematics and Methodology for Economics. Applications, Problems and Solutions. 1st ed. 2016. s.l.: Springer-Verlag (Springer Texts in Business and Economics). Online verfügbar unter http://ebooks.ciando.com/book/index.cfm/bok_id/2004175.
- **Fourer, R.; Gay, D. M.; Kernighan, B. W. (2009):** AMPL. A modeling language for mathematical programming. 2. ed., 5. [print.]. Belmont, Ca.: Brooks/Cole. Online verfügbar unter http://ampl.com/resources/the-ampl-book/chapter-downloads/.
- **Fritzemeier, C. J.; Celius-Dietrich, G.; Luankesorn, L. (2015)**: glpkAPI. R Interface to C API of GLPK. Depends R(>=2.6.0); Requires GLPK(>=4.42). Version 1.3.0: CRAN. Online verfügbar unter https://cran.r-project.org/web/packages/glpkAPI/index.html.
- **Fromen, B. (2004):** Faire Aufteilung in Unternehmensnetzwerken. Lösungsvorschläge auf der Basis der kooperativen Spieltheorie. Wiesbaden: Deutscher Universitätsverlag (Information - Organisation - Produktion). Online verfügbar unter http://dx.doi.org/10.1007/978-3-322-81803-4.
- **Gately, D. (1974)**: Sharing the Gains from Regional Cooperation. A Game Theoretic Application to Planning Investment in Electric Power. In: *International Economic Review* 15 (1), S. 195. DOI: 10.2307/2526099.
- **Gebele, D. (2016)**: Nucleolus und Pränucleolus als Lösungskonzepte fur Kooperationsspiele und Untersuchungen in R. Bachelorarbeit in der Wirtschaftsinformatik an der Hochschule Kempten.
- **Genolini, C. (2008)**: A (Not So) Short Introduction to S4. Object Oriented Programming in R. Online verfügbar unter ftp://cran.r-project.org/pub/R/doc/contrib/Genolini-S4tutorialV0-5en.pdf.
- **Guajardo, M.; Jörnsten, K. (2014):** Common Mistakes in Computing the Nucleolus. In: *Discussion paper (Norges handelshøyskole, Institutt for foretaksøkonomi)*.

- **HOLLER, M. J.; ILLING, G. (2006)**: Einführung in die Spieltheorie. 6., überarbeitete Auflage. Berlin, Heidelberg: Springer Berlin Heidelberg (Springer-Lehrbuch). Online verfügbar unter http://dx.doi.org/10.1007/3-540-29948-3.
- **JENE, S. (2015)**: Die faire Verteilung von Effizienzgewinnen in Kooperationen. Eine kritische Analyse der Eignung des [Tau]-Werts und des [Chi]-Werts. Zugl.: Duisburg, Essen, Univ., Diss., 2014. Wiesbaden: Springer Gabler (Research).
- **LEJANO, RAUL P.; DAVOS, CLIMIS A. (1999)**: Cooperative Solutions for Sustainable Resource Management. In: *Environmental Management* 24 (2), S. 167–175. DOI: 10.1007/s002679900224.
- **LITTLECHILD, S. C.; VAIDYA, K. G. (1976):** The propensity to disrupt and the disruption nucleolus of a characteristic function game. In: *International Journal of Game Theory* 5 (2), S. 151–161. DOI: 10.1007/BF01753316.
- **MASCHLER, M.; ZAMIR, S.; SOLAN, E. (2013)**: Game Theory: Cambridge University Press. Online verfügbar unter https://books.google.de/books?id=lqwzqgvhwXsC.
- **MIGUEL, A. M.; RODRIGUEZ, E. S. (2006)**: TUGlab. Online verfügbar unter http://mmiras.webs.uvigo.es/TUGlab/ zuletzt aktualisiert am 30.04.2012, zuletzt geprüft am 14.07.2017.
- **NARAHARI, Y. (2014):** Game theory and mechanism design. Singapore, Hackensack, N.J: World Scientific Pub. Co (IISc lecture notes series, v. 4).
- **NARAHARI, Y.(2012)**: Cooperative Game Theory Other Solution Concepts. Online verfügbar unter http://lcm.csa.iisc.ernet.in/gametheory/ln/web-cp6-othertopics.pdf.
- **NEBEL, B. (2009):** Spieltheorie. Skript. Universität Freiburg. Online verfügbar unter http://gki.informatik.uni-freiburg.de/teaching/ss09/gametheory/spieltheorie.pdf.
- **PATIL, USHA; RASHMI, M. (2016)**: SOLVING LINEAR PROBLEMS USING SIMPLEX METHOD. In: *International Journal of Recent Trends in Engineering and Research*, S. 388. Online verfügbar unter http://www.ijrter.com/papers/volume-2/issue-5/solving-linear-problems-using-simplex-method.pdf.
- **PELEG, B.; SUDHÖLTER, P. (2007):** Introduction to the Theory of Cooperative Games. Second Edition. Berlin, Heidelberg: Springer-Verlag Berlin Heidelberg (Theory and Decision Library, Series C, 34). Online verfügbar unter http://site.ebrary.com/lib/alltitles/docDetail.action?docID=10189328.
- **PETERS, H. (HG.) (2015)**: Game theory. A multi-leveled approach. 2nd ed. 2015. Berlin, Heidelberg, s.l.: Springer Berlin Heidelberg (Springer Texts in Business and Economics).
- **SCHMEIDLER, D. (1969)**: The Nucleolus of a Characteristic Function Game. In HART, SERGIU (HG.) (1995): Game and economic theory. Selected contributions in honor of

Robert J. Aumann. Hg. Unter Mitarbeit von Robert J. Aumann. Ann Arbor: Univ. of Michigan Press. Online verfügbar unter https://books.google.de/books?hl=de&lr=&id=O-zDTe2CjIgC&oi=fnd&pg=PA231&dq=The+nucleolus+of+a+characteristicfunction+game.+&ots=opEDClfhXU&sig=m0OCXtIbx9G0ZxyU0hszGbEBe9s#v=onepage&q=The%20nucleolus%20of%20a%20characteristicfunction%20game.&f=false.

- **SHAPLEY, L. S. (1953):** A Value for n-Person Games. In: Harold William Kuhn und Albert William Tucker (Hg.): Contributions to the Theory of Games (AM-28), Volume II. Princeton: Princeton University Press.
- **SHAPLEY, L. S. (1967):** On balanced sets and cores. In: *Naval Research Logistics (NRL)* 14 (4), S. 453–460. DOI: 10.1002/nav.3800140404.
- **SLIKKER, M.; NOUWELAND, A. (2001):** Social and Economic Networks in Cooperative Game Theory. Boston, MA: Springer (Theory and Decision Library, Series C, 27). Online verfügbar unter https://books.google.de/books?id=X57wBwAAQBAJ&lpg=PR9&ots=gii0Zgough&dq=slikker%202001.
- **SMITH, D. (2017)**: Python and R top 2017 KDnuggets rankings. Online verfügbar unter http://blog.revolutionanalytics.com/2017/06/python-and-r-top-2017-kdnuggets-rankings.html, zuletzt geprüft am 10.07.2017.
- **STAUDACHER, J. (2017):** Erkenntnisse aus Diskussionen bzw. aus dem Schriftverkehr zwischen Prof. Dr. Staudacher und Masterand Johannes Anwander im Kontext der Masterarbeit *„Untersuchungen zur kooperativen Spieltheorie und Erweiterung des R-Pakets CoopGame“*.
- **STRAFFIN, P. D. (1993)**: Game theory and strategy. 8. printing. Washington, DC: Mathematical Association of America (New mathematical library, 36).
- **SUDHÖLTER, P. (1996):** The Modified Nucleolus as Canonical Representation of Weighted Majority Games. In: *Mathematics of Operations Research* 21 (3), S. 734–756.
- **SUDHÖLTER, P. (1997):** The Modified Nucleolus. Properties and Axiomatizations. In: *Int. Journal of Game Theory* (26), S. 147–182.
- **SUDHÖLTER, P.; ROSENMÜLLER, J. (2000):** Cartels via the Modiclus. In: *Institute of Mathematical Economics*.
- **TARASHNINA, S. (2011):** The simplified modified nucleolus of a cooperative TU-game. In: *TOP* 19 (1), S. 150–166. DOI: 10.1007/s11750-009-0118-z.

- **THE GNU PROJECT (2017)**: GLPK. GNU Linear Programming Kit. Version 4.61: Free Software Foundation. Online verfügbar unter https://www.gnu.org/software/glpk/.
- **THE R FOUNDATION (2017)**: R: What is R? Online verfügbar unter https://www.r-project.org/about.html, zuletzt aktualisiert am 19.06.2017, zuletzt geprüft am 10.07.2017.
- **TIJS, STEF H. (1981)**. Bounds for the core of a game and the t-value. In O. Moeschlin, & D. Pallaschke (Eds.), Game Theory and Mathematical Economics (pp. 123-132). Amsterdam: North-Holland Publishing Company. Online verfügbar unter https://pure.uvt.nl/portal/files/666664/ST16____.PDF.
- **TIJS, STEF H. (1987)**: An axiomatization of the τ-value. In: *Mathematical Social Sciences* 13 (2), S. 177–181. DOI: 10.1016/0165-4896(87)90054-0. Online verfügbar unter http://ac.els-cdn.com/0165489687900540/1-s2.0-0165489687900540-main.pdf?_tid=0af81786-576a-11e7-9a2e-00000aab0f02&acdnat=1498150042_9456e00982b25730928ef5304469d665.
- **WANG, L.; FANG, L.; HIPEL, K. (2007):** A Game-Theoretic Approach to Brownfield Redevelopment. Negotiation on Cost and Benefit Allocation. Online verfügbar unter https://pdfs.semanticscholar.org/9213/3329883fc28c4631b871c6849d2bfec6aefc.pdf
- **WICKHAM, H. (2015)**: OO field guide ..Advanced R. Online verfügbar unter http://adv-r.had.co.nz/OO-essentials.html, zuletzt aktualisiert am 24.01.2017, zuletzt geprüft am 16.07.2017.
- **WICKHAM, H. (2015b):** R packages. First edition. Sebastopol, CA: O'Reilly Media. Online verfügbar unter http://site.ebrary.com/lib/alltitles/docDetail.action?docID=11040183
- **WIESE, H. (2005):** Kooperative Spieltheorie. München: De Gruyter Oldenbourg. Online verfügbar unter http://www.degruyter.com/search?f_0=isbnissn&q_0=9783486837469&searchTitles=true
- **YOUNG, H. P. (1985):** Monotonic Solutions of Cooperative Games. In: *Int. Journal of Game Theory* (14), S. 65–72.
- **YOUNG, H. P.; OKADA, N.; HASHIMOTO, T. (1982):** Cost allocation in water resources development. In: *Water Resources Research* 18 (3), S. 463–475.
- **ZELEWSKI, S. (2009):** Faire Verteilung von Effizienzgewinnen in Supply Webs - ein spieltheoretischer Ansatz auf der Basis des τ-Werts: Logos-Verlag. Online verfügbar unter https://books.google.de/books?id=gfYXqTb6f1UC.

Erklärung und Ermächtigung

Erklärung

Ich versichere, dass ich diese Bachelorarbeit selbstständig angefertigt, nicht anderweitig für Prüfungszwecke vorgelegt, alle benutzten Quellen und Hilfsmittel angegeben, sowie wörtliche und sinngemäße Zitate gekennzeichnet habe.

Kempten, den _____ _____

Unterschrift

Ermächtigung

Hiermit ermächtige ich die Hochschule Kempten zur Veröffentlichung der Kurzzusammenfassung (Abstract) meiner Arbeit, z. Bsp. Auf gedruckten Medien oder auf einer Internetseite.

Kempten, den _____ _____

Unterschrift