Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Victoria Bibaeva

Implementierung und Vergleich verschiedener Lösungsverfahren für praxisnahe Kursplanung

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Victoria Bibaeva

Implementierung und Vergleich verschiedener Lösungsverfahren für praxisnahe Kursplanung

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Bettina Buth
Zweitgutachter : Prof. Dr. Julia Padberg

Eingereicht am 24. Oktober 2014

Victoria Bibaeva

Thema der Arbeit

Implementierung und Vergleich verschiedener Lösungsverfahren für praxisnahe Kursplanung

Stichworte

Kursplanung, Wahlpflichtfachplanung, Operations Research, Modellierung, Losverfahren, Solver

Kurzzusammenfassung

In einigen Hochschulen stellt sich die Zuordnung der Studenten zu den von ihnen gewünschten/ausgewählten Fächern als eine Herausforderung, da es aufgrund der vorhandenen Kapazitäten meistens nicht möglich ist, allen ihre Wünsche zu erfüllen. Dieses (Optimierungs-)Problem lösen die Hochschulen auf verschiedene Art und Weise, um das Ziel zu erreichen, möglichst viele Studenten zufriedenzustellen. In dieser Arbeit werden verschiedene computergestützte Lösungsansätze vorgestellt und miteinander verglichen. Anhand echter sowie zum Testen generierter Daten wird ermittelt, welches Lösungsverfahren unter welchen Bedingungen zu der höchsten Zufriedenheit der Studenten führt.

Victoria Bibaeva

Title of the paper

Implementing and comparing different techniques of solving real world instances of preferential course scheduling problem

Keywords

University timetabling, student-to-course scheduling, preferential course scheduling, operations research, optimization model, random technique, solver

Abstract

In some universities the process of assigning students to courses according to their preferences is a real challenge. Due to the limited availability of institutional resources the planners are often unable to fulfill all the students' wishes. This optimization problem, whose goal is to satisfy the largest possible number of requests, is solved differently by any given university. Several computer-aided solution techniques to the so-called preferential course scheduling problem are implemented and compared in this paper. We solve the instances of the problem with the actual course data as well as with a large amount of generated data and then determine, which technique under which conditions leads to the highest degree of student satisfaction.

Inhaltsverzeichnis

1 Einführung 1

1.1 Motivation 1

1.2 Zielsetzung 2

1.3 Aufbau der Arbeit 2

2 Grundlagen 3

2.1 Problembeschreibung 3

2.2 Mathematisches Modell 5

2.3 Verwandte Probleme 7

2.4 Existenz einer Lösung 9

2.5 Überblick der bekannten Lösungsverfahren 11

2.6 Beschreibung der ausgewählten Lösungsverfahren 13

2.6.1 Losverfahren 14

2.6.2 Branch-and-Bound-Verfahren 16

2.6.3 DROP-Verfahren 19

3 Anforderung an eine (fiktive) Planungsumgebung 21

4 Implementierung 23

4.1 Funktionsweise des Programms 23

4.2 Datenmodell 26

4.2.1 Modell.gmpl 26

4.2.2 Input.csv 27

4.2.3 Input_Modell_Daten.gmpl 27
4.2.4 Output.csv 29
5 Generierung der Testdaten 30
6 Auswertung und Vergleich der Lösungsverfahren 34
6.1 Zufriedenheit 34
6.2 Laufzeit 37
6.3 Skalierbarkeit 39
6.4 Erweiterbarkeit 41
7 Analyse der Einflussfaktoren 43
7.1 Kursbeliebtheit 43
7.2 Freiheitsgrad der Wahl 45
8 Zusammenfassung 47
8.1 Fazit 47
8.2 Ausblick 48
8.3 Offene Probleme 48
9 Symbolverzeichnis 49
10 Abbildungsverzeichnis 50
11 Tabellenverzeichnis 51
12 Literaturverzeichnis 52

1 Einführung

Im Hochschulalltag werden die Führungskräfte regelmäßig damit herausgefordert, einen guten, d.h. für alle Beteiligten akzeptierten Studienplan zu bilden. Meistens handelt es sich um eine Zuordnung zwischen Lehrkräften, Studenten, Zeiten der Lernveranstaltungen und Räumen. Die Wünsche der Studierenden werden nur insofern berücksichtigt, dass sie möglichst keine Veranstaltungen zu demselben Zeitpunkt besuchen müssen. In der Praxis existieren neben den normalen Lehrveranstaltungen auch Wahlveranstaltungen, die die Studenten frei auswählen dürfen, z.B. Ausflüge, Projekte, Wahlpflichtfächer, nicht fachbezogene Kurse o.ä. Bei diesen Veranstaltungen können die Studenten angeben, welchen Kurs sie am liebsten belegen möchten, welchen am zweitliebsten usw. Nachdem alle Studenten ihre Präferenzen angegeben haben, findet die Kurszuordnung statt. Das Ziel dabei ist, dass die meisten Studenten zufrieden sind: am besten sollen die meisten den Kurs mit der ersten Präferenz bekommen. Dadurch entsteht das sogenannte Kursplanungsproblem, ein Sonderfall des bekannteren und erheblich besser erforschten Stundenplanproblems, welcher im Folgenden ausführlich beschrieben wird.

1.1 Motivation

Es gibt Hochschulen, die das Problem auf eine etwas altmodische Art lösen, d.h. es wird per Hand ausgelost, welchen Kurs jeder Student besuchen wird. Die Zufriedenheit der Studenten wird hier nicht mit höchster Priorität beachtet. Das in dieser Arbeit beschriebene Programm entstand, um nachzuweisen, dass moderne Lösungsansätze aus dem Bereich Operations Research und zeitgemäße Nutzung des Computers deutlich zur Zufriedenheit der Studierenden beitragen können.

1.2 Zielsetzung

In dieser Arbeit werden drei verschiedene Lösungsverfahren präsentiert, implementiert und miteinander verglichen. Eins davon ist das oben genannte Losverfahren, das den beiden anderen Verfahren im Folgenden gegenübergestellt wird. Um den Vergleich zu generalisieren, wird außerdem eine hohe Anzahl an Testdaten generiert, die jede mögliche Präferenztabelle und Kursbeliebtheit repräsentieren können. Unterschiedliche Vergleichskriterien (Zufriedenheit, Laufzeit, Skalierbarkeit, Erweiterbarkeit) werden bei allen Verfahren analysiert und ausgewertet, Einflussfaktoren (Kursbeliebtheit, Anzahl der Studenten, Freiheitsgrad der Wahl usw.) werden ermittelt.

1.3 Aufbau der Arbeit

In Kapitel 2 werden wir das praxisnahe Kursplanungsproblem präsentieren und analysieren, auf das Verhältnis zu anderen Optimierungsproblemen eingehen und verschiedene Lösungsansätze beschreiben.

Anschließend betrachten wir in Kapitel 3 eine Kursplanungssoftware für den Einsatz an jeweiligen Hochschulen aus der Sicht aller Beteiligten und versuchen, ihre Bedürfnisse in die System-Anforderungen einfließen zu lassen.

In Abschnitt 4 wird berichtet, wie wir diese Anforderungen bei der Implementierung des Systems umgesetzt haben.

Ferner geht es in Kapitel 5 um einen Datengenerator, der Input-Dateien für die bestehende Kursplanungsanwendung erzeugt. Diesen benötigen wir, um eine Basis für die umfassende Auswertung der drei Lösungsverfahren zu verschaffen und alle denkbaren Konstellationen zu überprüfen.

In Kapitel 6 werden wir die wichtigsten Aspekte und Kriterien erläutern, die dabei helfen, die Auswahl eines besten der drei Verfahren zu erleichtern, und in nachfolgendem Abschnitt 7 werden Faktoren und Randbedingungen analysiert, die diese Ergebnisse beeinflussen.

Schließlich widmet sich Kapitel 8 einer Zusammenfassung inklusive Ausblick und im Rahmen dieser Arbeit noch offener Probleme.

2 Grundlagen

In diesem Kapitel wird das praxisnahe Kursplanungsproblem präsentiert und analysiert, das Verhältnis zu anderen Optimierungsproblemen und verschiedene Lösungsansätze werden diskutiert.

2.1 Problembeschreibung

Jede Bildungseinrichtung hat ihre eigenen Lehrpläne, Abläufe und Regeln zum Aufbau der Stundenpläne, die sich in der Praxis von denen in anderen Institutionen deutlich unterscheiden. Die konkrete Problemstellung, auf welches wir uns im Folgenden konzentrieren werden, entstand an einer Hamburger Hochschule (Hartmann, 2014). Dort werden einmal pro Semester diverse Ausflüge angeboten, während derer eine Unterrichtseinheit erfolgt, und jeder Studierende darf auswählen, an welchem Ausflug er oder sie gerne teilnehmen möchte. Um das Verständnis zu verbessern und das Problem möglichst zu verallgemeinern, werden wir diese Ausflüge als „Kurse“ bezeichnen. Studierende aller Geschlechter werden hier einfachheitshalber „der Student“ genannt.

Trotz der Tatsache, dass die Anzahl der zur Verfügung stehenden Plätze für jeden Kurs begrenzt ist, wollte die Hochschulleitung die Freiheit der Wahl nicht auf nur eine Option beschränken. Dementsprechend darf jeder Student gleich drei Kurse priorisiert aussuchen, d.h. er gibt an, welchen Kurs er sich als erstes wünscht, welchen als zweites (wenn der erste aus irgendeinem Grund schon voll besetzt ist oder nicht stattfindet) usw. Somit werden die Studierenden, die einen sehr beliebten und möglicherweise schon ausgebuchten Kurs als ihre erste Wahl angegeben haben, trotzdem zufriedengestellt, indem sie eventuell einen anderen Wunschkurs bekommen können.

Die priorisierten Optionen werden in einer Excel-Tabelle erfasst, für ein Beispiel siehe Tabelle 1. Eine „0“ für einen Kurs bedeutet, dass man an diesem teilnehmen darf bzw. die Kursanforderungen erfüllt.

Für einige Studenten gibt es keine freie Wahl, da sie beispielsweise die Voraussetzungen nur für einen Kurs erfüllen oder nur dieser ihrem Curriculum entspricht. Dann dürfen sie nur diesen Kurs als ihre erste Priorität ankreuzen (z.B. Anke Krüger aus Tabelle 1), alle anderen werden mit „-1“ gekennzeichnet und dürfen den Betroffenen (den sogenannten „**Studenten ohne Auswahl**“) nicht zugewiesen werden. Dies kann bei einigen Studenten nicht nur einen Kurs betreffen, sondern mehrere (siehe Can Ögultürk aus Tabelle 1), aber aufgrund der trotzdem möglichen Prioritätsangabe werden solche Fälle nicht gesondert behandelt.

Name	Vorname	Cape Town	New York	Rio	Dublin	Saigon
Schmidt	Paul	1	2	3	0	0
Müller	Jan	3	0	1	0	2
Busch	Heike	0	2	0	1	0
Krüger	Anke	-1	1	-1	-1	-1
Ögultürk	Can	-1	-1	2	0	1

Tabelle 1. Beispiel für eine Präferenztabelle

Nachdem alle Daten von jedem Student erfasst sind, kommt es zur Zuweisung der Studenten zu den Kursen. Dabei ergibt sich die wichtigste Frage: welcher Student bekommt seinen ersten Wunsch und welcher nicht?

Eine weitere Schwierigkeit besteht darin, dass nicht alle angebotenen Kurse tatsächlich zustande kommen müssen. Der Grund dafür sind die begrenzten Ressourcen der Hochschule, weswegen nur ein Teil der Kurse, nämlich die beliebtesten bzw. die meist ausgewählten, stattfinden kann. Die Hochschulleitung entscheidet, wie viele Kurse angeboten werden und wie viele davon stattfinden müssen. Eine Antwort auf die Frage „Welche Kurse kommen nicht zustande?“ sollte ebenfalls ein Bestandteil des Zuweisungsverfahrens sein.

Die einfachste Vorgehensweise ist, jedem Student aus der Liste der Reihe nach seinen ersten Wunschkurs zuzuweisen; wenn das nicht geht (z.B. der Kurs ist bereits ausgebucht), den zweiten Kurs zuteilen usw. Das bedeutet allerdings, dass die ersten Studierenden aus der Liste immer ihre erste Wahl bekommen, dagegen sind die danach folgenden Personen benachteiligt und bekommen eventuell einen Kurs, den sie gar nicht erst nehmen wollten. Egal, ob die Liste alphabetisch sortiert ist oder nach dem Prinzip „First come first served“ bearbeitet wird, solche Zuweisung wurde von der Hochschulleitung als nachteilig abgelehnt.

Zum Einsatz kam eine andere Methode, das sogenannte **Losverfahren**, das manuell von einem Planer mithilfe der Excel-Tabelle, Papierzettel und einer Lostrommel ausgeführt wird. Zunächst werden anhand aller Präferenzen die „unbeliebtesten“ Kurse ermittelt und aus dem weiteren Verlauf ausgeschlossen. Anschließend wird jeweils ein Papierzettel mit dem Namen eines Studenten aus dem Gerät gezogen. Der Planer versucht, diesen Studenten zu den von

ihm gewünschten Kursen in deren Reihenfolge zuzuweisen, und wenn diese Kurse voll sind, bekommt er einen Kurs mit den bisher wenigsten Teilnehmern.

Diese manuelle Bearbeitung dauerte normalerweise ein bis zwei Tage, infolgedessen bekamen ca. 60% der Studenten ihren ersten Wunschkurs zugewiesen. Um eine Verbesserung zu erzielen, wurde uns ein Auftrag gegeben, die „state-of-the-art" Lösungsverfahren zu implementieren und das vielversprechendste Verfahren zum Einsatz zu bringen.

2.2 Mathematisches Modell

Gegeben sind folgende Parameter:

m – Anzahl Studenten; $m \in \mathbb{Z}^+$

n – Anzahl der angebotenen Kurse; $n \in \mathbb{Z}^+$

k – Anzahl Kurse, die zustande kommen; $k \in \mathbb{Z}^+$, $k \leq n$

t – Anzahl möglicher Präferenzwerte; $t \in \mathbb{Z}^+$, $t > 2$

$P = \{p_r \in \mathbb{Z}, p_r \geq -1 \mid r \in \overline{1,t}\} = \{1, 2, 3, \dots, 0, -1\}$ – Menge der Präferenzwerte

$a_{ij} \in P$ – **Präferenz** des Studenten i für Kurs j; $i \in \overline{1,m}$, $j \in \overline{1,n}$

c_j – Kapazität des Kurses j, d.h. maximale Anzahl der Teilnehmer; $c_j \in \mathbb{Z}^+$, $j \in \overline{1,n}$.

Dann bestimmen wir:

- Echte Obergrenze für die Präferenzwerte: $S = \max_{r \in \overline{1,t}}(p_r) + 1$
- **Priorität** des Studenten i für Kurs j in Punkten: $b_{ij} = \begin{cases} S - a_{ij}, \text{wenn } a_{ij} > 0 \\ a_{ij} \text{ sonst} \end{cases}$, wobei $i \in \overline{1,m}$, $j \in \overline{1,n}$.

So werden die Kurse für jeden Studenten absteigend mit den Punkten bewertet. Demnach bekommt jeder Kurs der ersten Wahl die höchste Punktzahl bzw. Priorität, wogegen nicht gewählte und verbotene Kurse keine bzw. negative Punkte bekommen. Sei beispielsweise $P = \{1, 2, 3, 0, -1\}$. Studenten dürfen also 3 Kurse auswählen und dabei ihre Präferenz angeben:

- ✓ 1 für den Kurs, den sie am liebsten besuchen würden,
- ✓ 2 – wenn der erste Kurs voll ist,
- ✓ 3 – wenn diese beiden ausgebucht sind,
- ✓ 0 – „egal",
- ✓ -1 – sie dürfen an diesem Kurs nicht teilnehmen.

Dementsprechend ist die Priorität des ersten Kurses 3, des zweiten 2, des dritten 1, den nicht ausgewählten 0, den unzulässigen -1, siehe auch Tabelle 2.

Präferenz	1	2	3	0	-1
Priorität	3	2	1	0	-1
Bedeutung	Kurs, den man am liebsten besuchen würde	wenn der erste Kurs voll ist	wenn die ersten zwei ausgebucht sind	Keine Angabe (egal)	darf an diesem Kurs nicht teilnehmen

Tabelle 2. Präferenzen vs. Prioritäten

Gesucht werden die Belegungen von folgenden Variablen:

- $x_{ij} = \begin{cases} 1, & wenn\ Student\ i\ an\ Kurs\ j\ teilnimmt \\ 0 & sonst \end{cases}, i \in \overline{1,m}, j \in \overline{1,n}$
- $y_j = \begin{cases} 1, & wenn\ Kurs\ j\ zustande\ kommt \\ 0 & sonst \end{cases}, j \in \overline{1,n}$

Anschließend können wir das Ziel formulieren – möglichst vielen Studenten ihren besten Wunsch erfüllen, also ihre **Zufriedenheit** maximieren (Hartmann, 2014):

(*) Maximiere $\sum_{i,j} b_{ij} * x_{ij}$ unter den Nebenbedingungen:

(1) höchstens k Kurse werden stattfinden: $\sum_{j=1}^{n} y_j \leq k$

(2) Jeder Student bekommt genau einen Kurs: $\sum_{j=1}^{n} x_{ij} = 1, \forall i \in \overline{1,m}$

(3) Jeder Student bekommt einen für ihn zulässigen Kurs: $\sum_{j=1}^{n} a_{ij} * x_{ij} \geq 0, \forall i \in \overline{1,m}$

(4) Kapazitäten aller Kurse werden nicht überschritten: $\sum_{i=1}^{m} x_{ij} \leq c_j * y_j, \forall j \in \overline{1,n}.$

Die Suche nach einer optimalen Zuordnung von Studenten zu den Kursen unter den oben genannten Nebenbedingungen wird in dieser Arbeit als **Kursplanungsproblem** bezeichnet.

In dieser Form bildet das Problem ein Spezialfall der *ganzzahligen Optimierung* (engl. integer linear programming, kurz ILP), die folgendermaßen formuliert wird (Domschke, et al., 2011). Sei $A = (a_{ij})_{m \times n}$ eine ganzzahlige Matrix der Größe $m \times n$, $b = (b_i)_m$ – ein m-dimensionaler ganzzahliger Vektor, $c = (c_j)_n$ und $x = (x_j)_n$ – n-dimensionale ganzzahlige Vektoren. A, b und c sind vorgegeben, x wird gesucht:

Maximiere $\sum_{j=1}^{n} c_j * x_j$ unter den Nebenbedingungen:

- $\sum_{j=1}^{n} a_{ij} * x_j \leq b_i, \forall i \in \overline{1,m}$
- $x_j \geq 0, \forall j \in \overline{1,n}$
- $x_j \epsilon \mathbb{Z}, \forall j \epsilon \overline{1,n}$

In der Matrixschreibweise entsprich dies:

Maximiere $c^T x$, wobei $Ax \leq b, x \geq 0, x_j \epsilon \mathbb{Z}, \forall j \epsilon \overline{1,n}$.

Alle Nebenbedingungen (engl. *Constraints*) in (*) lassen sich bekannterweise äquivalent in diese Form umwandeln, siehe (Domschke, et al., 2011).

Wenn die Komponenten des Lösungsvektors x nur die Binärwerte (0 oder 1) belegen dürfen, spricht man von einer *0-1 Programmierung* (engl. binary integer linear programming).

Man unterscheidet zwischen *harten* Constraints, die unbedingt erfüllt sein müssen, und *weichen* Constraints, die nicht zwingend zu erfüllen sind, meistens nur zu einem gewissen Ausmaß eintreten. Eine (gültige, zulässige) Lösung ist in dem Fall diejenige, bei der alle harten Constraints erfüllt sind. Eine optimale Lösung ist gültig und liefert den höchst möglichen Wert der Zielfunktion, d.h. möglichst viele weiche Constraints müssen zum maximalen Grad erfüllt sein. Auf die Frage, ob eine zulässige bzw. optimale Lösung für das jeweilige Problem existiert, werden wir später noch eingehen.

In der Problemstellung (*) betrachten wir die Nebenbedingungen (1) – (4) als harte Constraints, und die Zielfunktion (Zufriedenheit der Studenten) als weiches Constraint.

2.3 Verwandte Probleme

Kursplanung gehört zu einer breiten Klasse der automatisierten Stundenplanung, die seit den 1960er Jahren im Fokus der Forscher im Bereich Operations Research steht. Im Allgemeinen versteht man unter *Stundenplanung* (engl. automated timetabling) eine Zuordnung zwischen gegebenen Unterrichtseinheiten, Lernenden, Lehrkräften, Lernräumen und festen Zeitintervallen, typischerweise innerhalb einer Woche. Diese Zuordnung unterliegt einer Reihe von diversen Einschränkungen (siehe z.B. Nebenbedingungen in (*)), die man zu einem gewissen Grad einhalten muss. Jedes Land hat sein eigenes Schulsystem, jede Bildungseinrichtung hat ihre eigenen, historisch gewachsenen sowie individuellen, studienfachabhängigen Anforderungen an einen guten Stundenplan.

Zahlreiche Publikationen schlugen praxisnahe Algorithmen vor, die aufgrund der zu konkreten Anforderungen nur in ihren Hochschulen anwendbar wären. Zum Beispiel, in einigen Institutionen (Perzina, 2007) folgt jeder Student seinem individuellen Curriculum; es ist somit kaum möglich, zwei Studenten mit dem gleichen Stundenplan zu finden, was die Anzahl der bei der Planung entstehenden Konflikte stark erhöht. Andere Wissenschaftler berichteten von Algorithmen, die eine vereinfachte Variante des Problems lösten und eine

Anpassung an jegliche hochschulspezifische Bedürfnisse nur mit einem hohen Aufwand zuließen (Rossi-Doria, et al., 2003). Einige Forscher [(Burke, et al., 1998), (McCollum, 2007)] bemerkten, dass es angesichts dieser Situation sehr schwierig ist, die vorgeschlagenen Algorithmen miteinander zu vergleichen und allgemeinere Algorithmen überall anzuwenden.

Viele Autoren nutzen verschiedene Terminologien für ihre Planungsprobleme, wir werden uns hier an die Terminologie von Schaerf (Schaerf, 1999) und Carter (Carter, et al., 1998) anlehnen.

Zusammen mit der Prüfungsplanung (engl. examination timetabling) bilden Schulstundenplanung (school timetabling) und Hochschulstundenplanung (university course timetabling) die drei wichtigsten Klassen des Stundenplanungsproblems (Schaerf, 1999). Der Unterschied zwischen den letzten beiden besteht darin, dass die Schulklassen (im Gegensatz zu Kursgruppen) disjunkte Mengen der Schüler sind und feste Lernräume besitzen. Schullehrer unterrichten mehrere Klassen, die Hochschullehrkräfte dagegen nur eine Semestergruppe pro Fach. *Hochschulstundenplanung* wird wiederum in 5 Teilprobleme zergliedert (Carter, et al., 1998):

1) Zuordnung der Kurse bzw. Kursgruppen zu Zeitintervallen (und anschließend Studenten zu Kursen)
2) Zuordnung der Lehrveranstaltungen zu Zeitintervallen, nachdem Lehrkräfte den Kursen und den Klassen bereits zugewiesen sind
3) Zuordnung der Studenten zu Kursgruppen, die bereits geplant sind, entsprechend ihrer Kurswahl
4) Zuordnung der Lehrkräfte zu Zeitintervallen (nach Verfügbarkeit)
5) Zuordnung der Lehrveranstaltungen zu den Lernräumen

Die Problemstellung (*) ist ähnlich dem dritten Teilproblem, dessen Ziel ist allerdings, jedem Studenten einen möglichst konfliktfreien Stundenplan bereitzustellen und die gegebenen Kursgruppengrößen und Raumkapazitäten einzuhalten. Der Unterschied zwischen diesen Problemklassen besteht darin, dass letztere nicht die Präferenzen der Studenten berücksichtigt, sondern nur die Liste der Wunschkurse. Folglich wird ein Student keine anderen Kurse bekommen als die, die er sich gewünscht hat, und wenn er einen Wunschkurs nicht bekommt, wird ihm keine Alternative angeboten.

Der von uns gewählte Begriff „Kursplanung“ entspricht also nicht dem englischen Begriff „course timetabling“. In der Literatur werden andere Ausdrücke benutzt, um auf die Möglichkeit hinzudeuten, dass die Studenten (aber auch Professoren) ihre Präferenzen für bestimmte Objekte (Fächer, Zeiten oder Räume) angeben können, z.B. „demand-driven timetabling“ [(Cambazard, et al., 2005), (Carter, 2001), (Rudová, et al., 2003), (Sampson, et al., 1995)], „preferential course scheduling“ (Bloomfield, et al., 1979), „student sectioning“ [(Busam, 1967), (Carter, 2001), (Macon, et al., 1966)]. Gemeint sind hier dennoch meistens die Präferenzwerte 0 und 1 – entweder möchte ein Beteiligter ein Objekt bekommen oder

nicht. Unser Problem (*) lässt sich nicht auf diese Werte abbilden: lediglich ein Wunschkurs pro Person wäre in diesem Fall möglich, und wenn mehrere, dann ohne jegliche Favorisierung.

Alternative Bezeichnungen für „Kurs“ im Sinne von (*) sind „Wahlpflichtfach“, „Wahlfach“, „Ausflug“ und „Projekt“ zu nennen, diese sind unserer Meinung nach zu hochschulspezifisch und nicht allgemein genug, um alle Wahlveranstaltungen abzudecken.

2.4 Existenz einer Lösung

Oft ist bei solchen Planungsproblemen das Ziel, eine (beliebige) im obigen Sinne optimale Zuordnung zwischen den gegebenen Objekten (Kurse, Räume, etc.) zu finden. In diesem Zusammenhang spricht man von einem *Optimierungsproblem*. Andererseits könnte man sich auf nur eine (beliebige gültige) Lösung begrenzen und dadurch ein *Suchproblem* lösen. Zugleich existiert ein *Entscheidungsproblem*, das sich mit der Frage beschäftigt, ob es für jede konkrete Problemstellung eine (optimale) Lösung gibt (Schaerf, 1999).

Für weitere Betrachtungen benötigen wir einen Einblick in die Komplexitätstheorie und Nichthandhabbarkeit, deren Begrifflichkeiten wir aus (Hopcroft, et al., 2011) entnommen haben. Grundlegend sind dabei die Klassen $\mathcal{P}$ und $\mathcal{NP}$ der Probleme, die von einer deterministischen bzw. nichtdeterministischen Turing-Maschine in polynomialer Zeit lösbar sind. Es gilt: $\mathcal{P} \subseteq \mathcal{NP}$.

Ein Problem ist dann *NP-vollständig*, wenn:

1. Es zur Klasse $\mathcal{NP}$ gehört und
2. Es auf jedes andere NP-Problem polynomial reduzierbar ist, d.h. wir können alle seine Instanzen mit einem polynomialen Zeitaufwand in Instanzen jedes anderen Problems aus $\mathcal{NP}$ überführen, das dieselbe Antwort (ja-nein) liefert.

Wenn ein Problem nur die Bedingung 2. erfüllt, dann ist es *NP-hart*. Beide Problemarten heißen auch nicht handhabbar, da sie wahrscheinlich einen mindestens exponentiellen Aufwand erfordern.

Wenn es einem gelingt, für ein solches Problem einen allgemeinen effizienten Lösungsalgorithmus mit polynomialem Zeitaufwand zu entwickeln, dann wären alle NP-vollständige Probleme in Klasse $\mathcal{P}$ enthalten, und damit würde die bekannte Behauptung $\mathcal{P} = \mathcal{NP}$ bewiesen. Alle erfolglosen Versuche, diese Gleichung zu beweisen, deuten allerdings darauf hin, dass höchstwahrscheinlich $\mathcal{P} \neq \mathcal{NP}$. Das bedeutet, dass es keinen generellen Algorithmus gibt, der in polynomialer Zeit jede Probleminstanz lösen kann, und der Lösungsaufwand steigt exponentiell mit der wachsenden Instanzgröße (Lewis, 2008), sprich Anzahl der jeweiligen Variablen und Constraints.

Für die meisten allgemeinen Planungsprobleme bzw. ihnen zugrundeliegenden Entscheidungsprobleme wurde bewiesen (siehe z.B. (Even, et al., 1976), (Schaerf, 1999)),

dass sie NP-hart bzw. NP-vollständig sind. Für einige konkrete Problemvarianten ist es hingegen möglich, eine **Heuristik** zu finden, die eine optimale Lösung zwar nicht garantieren kann, aber meistens eine „sehr gute“ und für die Praxis fast immer ausreichende Lösung liefert (Lewis, 2008).

Zur Lösung der 0-1 Programmierung (siehe Kapitel 2.2), die bekannterweise NP-vollständig ist (Karp, 1972), sind neben den Heuristiken auch **exakte Methoden** im Einsatz. Diese berechnen immer die optimale Lösung, falls eine existiert, zugegebenermaßen ignoriert man den erheblichen Aufwand.

Satz 1. Wenn eine Lösung für (*) existiert, dann existiert auch eine optimale Lösung.

Beweisskizze. Tatsächlich, wenn es eine gültige Lösung gibt, können wir diese zur Konstruktion anderer Lösungen nutzen. Wir verschieben nämlich Studenten, die ursprünglich einen Kurs mit niedriger Priorität bekamen, zu einem anderen zulässigen Kurs mit höherer Priorität, sodass die Zielfunktion einen größeren Wert erreicht und kein Constraint verletzt wird. Da unsere Domäne endlich ist, bekommen wir endlich viele mögliche Lösungen, und eine oder mehrere liefern einen maximalen Wert der Zielfunktion und sind somit auch die optimalen Lösungen.

Was aber tun, wenn es keine Lösung gibt?

In der Praxis kommt sehr oft vor, dass es keine gültige Lösung für eine Probleminstanz der Form (*) existiert. Z.B. es gibt mehr Studenten als vorhandene Plätze oder es gibt zu viele Studenten, die nur bestimmte Kurse belegen dürfen. Im zweiten Fall wird es schwierig, diesen Studenten einen gültigen Kurs zuzuweisen, wenn ein solcher gerade ausgebucht ist oder aus den angebotenen Kursen schon rausfiel. Es kann also eine Situation auftreten, dass ein Student keinen Kurs zugewiesen bekommt. Damit wird das harte Constraint (2) im Problem (*) verletzt und folglich ist keine gültige sowie optimale Lösung möglich.

Wie geht man mit solchen Konstellationen um? Wenn nicht alle harte Constraints erfüllt sein können, müssen einige *relaxiert* (dt. aufgewichen, zu weichen Constraints gemacht) werden, um möglichst nah an die Lösung zu kommen.

Zum einen könnte man „ausnahmsweise“ mehr Kursplätze für solche benachteiligte Studenten anbieten. Hiermit wird das Constraint (4) relaxiert, (2) dagegen erfüllt. Solche Ausnahmen können allerdings die Kapazitätsgrenzen in die Höhe treiben. Zum anderen kann man alle Kurse anbieten und (1) relaxieren, was eventuell zu unterbuchten Kursen führt. Aufweichung von (3) verursacht ggf. Verstöße gegen die Hochschulordnung. Wenn die Kosten dafür nicht hinnehmbar sind, erlaubt man einfach die Verletzung von (2) und benachrichtigt alle Beteiligten im Voraus, dass nicht jeder Student einen Kurs bekommen kann. In der genannten Hochschule hat sich die Leitung für den letzten Weg entschieden.

Demnach sind die Nebenbedingungen (1), (3) und (4) aus dem Kapitel 2.2 harte, (2) – ein weiches Constraint. Eine übliche Vorgehensweise, an weiche Constraints heranzugehen, besteht darin, diese in die Zielfunktion aufzunehmen, und zwar mit einem Faktor, der jeden

Constraint-Verstoß bestraft (für Beispiele siehe (Aubin, et al., 1989), (Badri, et al., 1998), (Schaerf, 1999)).

Unser Problem (*) wird folgendermaßen in (**) umformuliert:

(**) Maximiere $\left[\sum_{i,j} b_{ij} * x_{ij}\right] - \alpha * \left[m - \sum_{i,j} x_{ij}\right]$, wobei $\alpha \geq 0$ im Voraus festgelegt wird, unter den Nebenbedingungen:

(1') höchstens k Kurse werden stattfinden: $\sum_{j=1}^{n} y_j \leq k$

(2') Jeder Student bekommt höchstens einen Kurs: $\sum_{j=1}^{n} x_{ij} \leq 1, \forall i \in \overline{1,m}$

(3') Jeder Student bekommt einen für ihn zulässigen Kurs: $\sum_{j=1}^{n} a_{ij} * x_{ij} \geq 0, \forall i \in \overline{1,m}$

(4') Kapazitäten aller Kurse werden nicht überschritten: $\sum_{i=1}^{m} x_{ij} \leq c_j * y_j, \forall j \in \overline{1,n}$.

Aus (2‘) folgt nämlich $1 - \sum_{j=1}^{n} x_{ij} \geq 0$ für alle Studenten i, also $m - \sum_{i,j} x_{ij} \geq 0$. Die zweite Klammer in der Zielfunktion wird dementsprechend 0, wenn alle Studenten je einen Kurs bekommen. Je mehr Studenten ohne Kurs bleiben, desto größer wird dieser Term, was den Gesamtwert der Zielfunktion vermindert. Constraints (1‘)-(4‘) sind nun hart.

Im Folgenden werden wir explizit darauf hinweisen, wenn eine Lösung optimal bzw. gültig im Sinne von (**) ist, ansonsten ist immer die Formulierung (*) gemeint.

2.5 Überblick der bekannten Lösungsverfahren

Die ersten Versuche, einen Stundenplan mithilfe des Computers zu erstellen, bestanden darin, die an der jeweiligen Institution bereits angesetzte manuelle Planungsverfahren nachzuimplementieren [(Bloomfield, et al., 1979), (Junginger, 1986)] oder Planer bei ihren Entscheidungen zu unterstützen (Tripathy, 1992). Des Weiteren nutzte man Computer und neue Technologien als Bearbeitungshilfe, um die großen Datenmengen an Informationen über Studenten, Dozenten, Räumen usw. zu speichern und zu verwalten [(Foulds, et al., 2000), (Johnson, 1993)].

Trotz der scheinbar verbreiteten Problemstellung gibt es in der Wissenschaft kaum Forschungsergebnisse, die sich mit dem Thema Kursplanung befassen. Im Fall, dass Studenten ihre Kurse selbst auswählen dürfen, handelt es sich um Listen der Kurse, die jeder Student ein Semester lang belegen möchte. Anschließend wird eine Gruppierung vorgenommen, sodass Studenten zu unterschiedlichen Gruppen eines Kurses zugeteilt werden und einen bestenfalls konfliktfreien Stundenplan bekommen (siehe auch Teilproblem 3) im Abschnitt 2.3). Die Lösungsstrategien dafür kann man grob in folgende Kategorien unterteilen (Carter, et al., 1998):

- ✓ exakte Algorithmen:

- 0-1 Programmierung (Tripathy, 1980)
- Netzwerkflüsse (Ostermann, et al., 1982)

✓ konstruktive Heuristiken:

- sequentielle Zuweisung (darunter Losverfahren-ähnliche Methoden (Macon, et al., 1966) und das hier beschriebene DROP-Verfahren), ggf. mit Backtracking (Sampson, et al., 1995)
- constraint logic programming, kurz CLP (Rudová, et al., 2003)
- unvollständiges Branch-and-Bound mit zwei Verbesserungsphasen (Laporte, et al., 1986)

✓ Verbesserungsheuristiken, auch *Metaheuristiken* genannt (Burke, et al., 2003):

- Genetische Algorithmen, kurz **GA** (Perzina, 2007)
- Simulated Annealing (simulierte Abkühlung), kurz **SA** (Elmohamed, et al., 1998)
- Tabu-Suche, kurz **TS** (Dowsland, 1998).

Nur einige dieser Strategien sind für die Kursplanung anwendbar, darunter 0-1 Programmierung, sequentielle Zuweisung, CLP sowie die Metaheuristiken. Die wohl bekannteste Problemlösung im Rahmen der CLP erfolgt mithilfe der Programmiersprache Prolog (siehe z.B. (The14)). Aufgrund den verhältnismäßig höheren Laufzeiten bei großen Instanzen haben wir auf den Einsatz dieser Methode verzichtet.

Die Idee hinter den Verbesserungsheuristiken ist folgende: Sei eine zulässige Lösung des Problems bereits erhalten. Diese wird schrittweise nach einem bestimmten Schema modifiziert, um eine bessere Lösung zu erreichen.

Bei SA und TS werden in jeder Iteration die „Nachbarn“ der Lösung durchsucht, also diejenigen Lösungen, die man mithilfe eines festgelegten Operators aus der ursprünglichen Lösung berechnet (Carter, et al., 1998). **SA** wählt einen zufälligen Nachbarn der Lösung aus. Wenn dieser einen besseren Wert der Zielfunktion f liefert ($\Delta f \geq 0$), dann wird er als Grundlage für die nächste Iteration benutzt, sonst ($\Delta f < 0$) wird er nur mit Wahrscheinlichkeit $e^{\Delta f / T}$ übernommen. Temperatur $T > 0$ wird nach jeder gegebenen Anzahl der Iterationen auf eine bestimmte Weise vermindert, infolgedessen wird eine geringere Verschlechterung der f eher am Anfang als am Ende akzeptiert (Elmohamed, et al., 1998). **TS** wählt seinerseits in jeder Iteration den besten Nachbarn der aktuellen Lösung. Um lokalen Optimum zu vermeiden, ist ein Schritt Richtung außerhalb der Nachbarschaft ebenfalls erlaubt. Dabei soll eventuelles Pendeln zwischen zwei Lösungen verhindert werden, indem man eine Tabu-Liste führt. Diese enthält in der klassischen Variante die Rückzüge (im Sinne des Übergangs im Suchraum von einer besseren Lösung zu ihrer Vorgängerin), die für eine variable Anzahl der Iterationen nicht erlaubt sind. Zahlreiche Modifikationen der Tabu-

Suche haben weitere Attribute und Eigenschaften der Tabu-Liste, um die besseren Ergebnisse zu erzielen (Dowsland, 1998).

Vorteile beider Methoden sind relativ leichte Implementierung (zumindest bei SA) und Kombinierbarkeit mit anderen Heuristiken. Nachteile sind jedoch viele interne Parameter (Nachbarschaftsoperator, Abkühlungstempo, Abbruchsbedingung, Attribute der Tabu-Liste), von denen die Lösungsqualität stark abhängig ist, hoher Ressourcenbedarf und ggf. mehrfache Durchläufe.

Genetische Algorithmen basieren auf der Idee der natürlichen Selektion und betrachten eine Population der Lösungen. Bei jedem Iterationsschritt werden zwei Lösungen ausgesucht (Selektionsoperator) und miteinander zusammengefügt (Rekombinationsoperator). Mit einer bestimmten Wahrscheinlichkeit werden diese Nachkommen mutieren (Mutationsoperator), einige Lösungen werden hingegen verworfen. Eine Definition dieser und weiterer Operatoren sowie aller zugehörigen Parameter macht die Implementation der GA in der Regel ziemlich umfassend, von langen Ausführungszeiten ganz zu schweigen, was nur bei sehr komplexen Problemstellungen zweckmäßig wäre (Perzina, 2007).

Aus den oben genannten Gründen haben wir uns in dieser Arbeit auf 0-1 Programmierung und sequentielle Zuweisungsheuristiken konzentriert.

2.6 Beschreibung der ausgewählten Lösungsverfahren

Im Hinblick auf unser Ziel haben wir uns entschieden, neben dem manuell eingesetzten Losverfahren auch andere Algorithmen zu implementieren, die eine bessere Lösung versprachen. Selbstständige Implementierung der Standard-Algorithmen für 0-1 Programmierung ist heutzutage nicht mehr notwendig, daher haben wir nach fertigen Lösungsansätzen gesucht.

Zahlreiche Softwarepakete und Programmbibliotheken, die sogenannten *Solver*, implementieren allgemeine sowie einzelne Lösungsstrategien für viele bekannte Problemklassen wie lineare Optimierung, ganzzahlige lineare Optimierung (insbesondere 0-1 Programmierung), quadratische Optimierung etc. Einige Systeme, wie z.B. CPLEX (CPLEX14), GLPK (GLPK14) und TOMLAB (TOM14), bieten neben dem Solver auch eigene Modellierungssprache an.

Solver arbeiten mit den exakten Algorithmen, demzufolge haben wir das Kursplanungsproblem (*) mithilfe eines Solvers gelöst, um das optimale Ergebnis (falls es existiert) als Basis für den Vergleich der anderen Lösungen zu verwenden.

Unsere Wahl fiel auf den Solver namens GLPK (GNU Linear Programming Kit), ein weit verbreitetes Open Source Softwarepaket bzw. eine C-Bibliothek, die Schnittstellen zu Programmiersprachen wie C/C++, Java, Ruby, Python, Mathlab etc. anbietet. Die eingebaute Modellierungssprache nennt sich MathProg bzw. GMPL (GNU Mathematical Programming Language) und ist eine Teilmenge von AMPL (GMPL14). GLPK beinhaltet solche exakte

Methoden wie Branch-and-Bound, Branch-and-Cut, primales und duales Simplex-Verfahren und Innere-Punkte-Verfahren.

Für den Fall, dass es keine Lösung einer Probleminstanz gibt, musste eine Heuristik gefunden werden, die eine bessere Lösung berechnet, als beim Losverfahren. Der unten beschriebene DROP-Algorithmus wurde vom gleichnamigen Algorithmus für das Vehicle Routing Problem inspiriert und an die Kursplanung angepasst (Hartmann, 2014). Die Lösungsidee ist gewissermaßen das Gegenteil von Losverfahren: Anstatt einen Kurs nach dem anderen zu befüllen, lassen wir zunächst alle Kurse zustande kommen und weisen allen Studenten ihre Lieblingskurse zu. Dann wird die Anzahl der Kurse und Teilnehmer auf die gewünschte Größe reduziert – es wird also entscheiden, wer einem Kurswechsel unterliegt.

Somit bekommen wir für jede Probleminstanz drei Lösungen: eine optimale mithilfe der exakten Methoden (Standard-Implementation im Solver) und zwei zulässige mithilfe der Heuristiken (Los- und DROP-Verfahren).

2.6.1 Losverfahren

Diese Heuristik beantwortet die zwei wichtigsten Fragen sequentiell, d.h. zuerst wird bestimmt, welche Kurse zustande kommen, und dann erfolgt die Zuweisung der Studenten zu den Kursen.

Pseudocode (Hartmann, 2014):

1) Wähle k Kurse aus n: mit dem höchsten Beliebtheitsgrad B und die Kurse, die angeboten werden müssen (diejenigen, die für Studenten ohne Auswahl einzig möglich sind)

2) Solange bis alle Studenten an der Reihe waren:

   a) Ziehe zufällig einen Studenten und weise ihm den Kurs mit der höchsten Priorität zu, sofern dort noch Platz ist.

   b) Wenn kein Kurs mit Priorität > 0 mehr frei ist, wähle den Kurs mit der bisher niedrigsten Teilnehmerzahl.

   c) Wenn alle Kurse ohne Priorität (=0) voll sind oder wenn nur die Kurse noch frei sind, an welchen er nicht teilnehmen darf (-1), dann bekommt er keinen Kurs zugewiesen.

Wie bestimmen wir, welche Kurse am beliebtesten sind? Entweder betrachtet man die Summe aller angegebenen Prioritätswerte für jeden Kurs oder man berücksichtigt Anzahl der Studenten pro Kurs, die diesen als ihre erste Präferenz angegeben haben. Demzufolge werden wir den **Beliebtheitsgrad** B für den Kurs $j \in \overline{1,n}$ auf zwei Weisen berechnen:

- $B_1(j) = \sum_{i=1}^{m} b_{ij}$,
- $B_2(j) = \sum_{i=1}^{m} \left\lfloor \frac{b_{ij}+1}{S} \right\rfloor = \left|\{i \in \overline{1,m} \mid b_{ij} = 3\}\right|$, wobei S ist die echte Obergrenze für die Präferenzwerte (siehe Kapitel 2.2), und die eckigen Klammern stehen für die Abrundungsfunktion.

Im Kapitel 6.1 werden wir feststellen, welche Auswirkung die Wahl zwischen B_1 und B_2 hat.

Dennoch ist beim Schritt 1 auf den ersten Blick unklar, ob zuerst die beliebtesten Kurse ausgewählt werden, und dann diejenigen für Studenten ohne Auswahl, oder andersherum.

Bei der ersten Variante bekommen die meisten ihren Wunschkurs, Studenten ohne Auswahl dagegen gar keinen, wenn ihr Kurs soeben ausgebucht ist.

Die Hochschule hat sich folgerichtig für die zweite Variante entschieden und zuerst die Kurse zugelassen, die für Studenten ohne Auswahl als einzige infrage kämen. Diese Kurse sind zwar eventuell unbeliebt und müssen danach mit den anderen Studenten befüllt werden, für die es ihre niedrige Priorität gewesen war. Diese anderen Studenten sind mit derartigen Verteilung jedoch einigermaßen zufrieden, und viele bekommen einen zulässigen Kurs, was nicht der Fall wäre, wenn solche Kurse ausfallen und Studenten ohne Auswahl sitzen bleiben.

Man könnte sich auch eine dritte Variante überlegen, bei der abgewogen wird, wie beliebt jeder Kurs ist (im Sinne von B_1 oder B_2) und wie viele Studenten ohne Auswahl sich diesen Kurs wünschen. Anhand diesen Kennzahlen erfolgt die Entscheidung, welcher Kurs zustande kommt. Das wäre unserer Meinung nach eine gewisse Verbesserung, ist freilich nicht zielführend, da wir nicht das manuell eingesetzte Verfahren verbessern, sondern bessere Heuristiken einsetzen wollten.

Wie erfolgt nun die Verteilung der Studenten auf die k Kurse? Einerseits möchte man sicherstellen, dass Studenten ohne Auswahl ihren einzig erlaubten Kurs mit höchster Wahrscheinlichkeit auch bekommen, falls er zustande kommt, und andererseits eine zufällige Reihenfolge der Studentenauslosung beibehalten. Dies haben wir folgendermaßen erreicht: Nachdem die Studenten aus der Input-Datei sequentiell ausgelesen werden, vertauscht man zwei beliebige Studenten in der Liste miteinander, und zwar oft genug, damit fast jeder Student an der Reihe ist. Somit vermeiden wir die mögliche Vorsortierung der Studentenliste, z.B. alphabetische oder die Anmeldungsreihenfolge. Die in diesem Schritt entstandene Studentenliste wird in allen Algorithmen als Grundlage wiederverwendet. Anschließend werden die Studenten ohne Auswahl nach vorne vertauscht und bekommen während der sequentiellen Abarbeitung der Liste als ersten ihre Wunschkurse.

Nach diesem Verfahren kann eine Situation vorkommen (siehe Schritt 2.c), dass ein Student gar keinen Kurs zugewiesen bekommt. Dementsprechend liefert das Losverfahren nicht unbedingt eine gültige Lösung für die Problemstellung (*), aber eine gültige Lösung für (**).

2.6.2 Branch-and-Bound-Verfahren

Wie aus dem Namen des Verfahrens hervorgeht, beruht er auf zwei Lösungsprinzipien, bei deren Beschreibung wir uns an (Domschke, et al., 2011) orientiert haben:

- *Branching* (verzweigen*)*

Sei P_0 – das Ausgangsproblem, das wir lösen wollen, und $X(P_0)$ – Menge der zulässigen Lösungen. P_0 wird in k Teilprobleme $P_1, P_2, \dots, P_k$ zerlegt, sodass $X(P_0) = \bigcup_{i=1}^{k} X(P_i)$. Dabei soll möglichst gelten: $X(P_i) \cap X(P_j) = \emptyset$ für alle $i \neq j, i, j \in \overline{1,k}$. Die Probleme $P_1, P_2, \dots, P_k$ können wir analog weiter verzweigen und dadurch einen Baum konstruieren, siehe Abbildung 1. Alle Probleme werden der Reihe nach abgearbeitet, und es wird jedes Mal entschieden, ob das aktuelle Problem weiter verzweigt wird oder nicht.

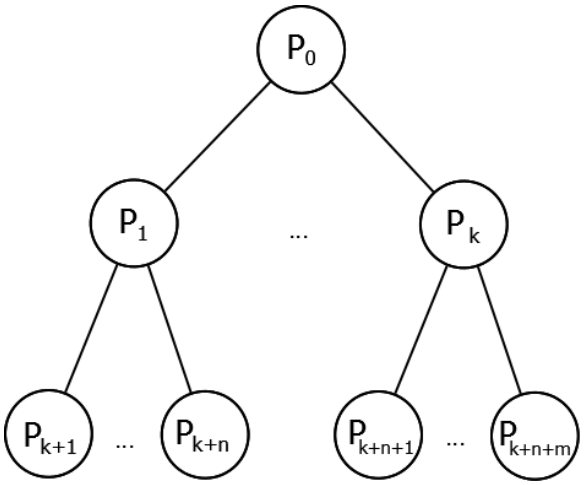

Abbildung 1. Baum der Probleme nach (Domschke, et al., 2011)

- *Bounding* (Schranken berechnen)

Hier werden die Schranken für Zielfunktionswerte berechnet, um die Verzweigung des Baumes zu begrenzen und letztendlich die optimale Lösung zu finden.

Die untere Schranke $\underline{F}$ ist der aktuell optimale Wert der Zielfunktion f im Ausgangsproblem. Bei der Initialisierung des Algorithmen nimmt man entweder $\underline{F} = -\infty$ oder einen größeren Wert, den man aus einer bereits vorhandenen zulässigen Initiallösung berechnet. Um eine solche Initiallösung zu erhalten, wird zum Beispiel eine Heuristik verwendet oder die relaxierte Variante von P_0 gelöst.

Die obere Schranke $\overline{F_i}$ wird für jedes Problem P_i bestimmt und stellt einen Zielfunktionswert von einer optimalen Lösung x des aus P_i entstandenen relaxierten Problems P_i' dar ($x \in$

$X(P_i')$). Dabei gilt $X(P_i) \subseteq X(P_i')$, da P_i' weniger bzw. weichere Constraints enthält. Wenn eine der folgenden Bedingungen zutrifft, wird das Problem P_i nicht weiter verzweigt:

- **a)** $\overline{F_i} \leq \underline{F}$. Die optimale Lösung des Teilproblems ist also nicht besser als die aktuell beste Lösung.
- **b)** $\overline{F_i} > \underline{F}$ und $x \in X(P_i) \subseteq X(P_0)$. Die optimale Lösung des Teilproblems ist besser als die aktuell beste Lösung und ist für das Ausgangsproblem zulässig. In diesem Fall setzt man $\underline{F} = \overline{F_i}$ und merkt x.
- **c)** $X(P_i') = \emptyset$. P_i' und somit auch P_i hat keine Lösung.

Der Ablauf des Verfahrens ist in der Abbildung 2 dargestellt.

Da es in unserem Fall um ein ganzzahliges Optimierungsproblem handelt, kommen verschiedene Relaxationsmöglichkeiten infrage. Zum einen kann man jedes Constraint weglassen und es ggf. in die Zielfunktion mit einem Faktor aufnehmen (siehe Kapitel 2.4). Zum anderen könnte man auf Ganzzahligkeitsbedingungen verzichten, d.h. anstatt des Typconstraints $x \in \{0,1\}$ betrachtet man $0 \leq x \leq 1, x \in \mathbb{R}$. Ein auf diese Weise relaxiertes Problem lässt sich mithilfe des Innere-Punkte-Verfahrens oder des Simplex-Verfahrens in polynomialer Zeit lösen (Domschke, et al., 2011).

Diese letzte Relaxationsstrategie kommt zum Einsatz, um ein Problem P_i zu verzweigen. Man findet nämlich eine optimale Lösung $x = (x_1, x_2, \dots, x_s)$ für das auf diese Art relaxierte P_i' (ohne Ganzzahligkeitsbedingungen) und wählt die Variable $x_j, j \in \overline{1,s}$ mit der größten Abweichung zu einer ganzen Zahl. Sei es $a < x_j < b$ mit $a, b \in \mathbb{Z}$. Es muss entschieden werden, ob x_j den Wert a oder b annehmen soll. Dafür zerlegen wir P_i in zwei Teilprobleme P_{i1} und P_{i2}, jedes davon aus denselben Constraints wie P_i besteht, mit zusätzlich $x_j \leq a$ bei P_{i1} und $x_j \geq b$ bei P_{i2} entsprechend.

Die Auswahl eines Problems aus der Liste, das als nächstes verzweigt werden soll, erfolgt (Domschke, et al., 2011):

- nach dem Prinzip „Last In – First Out“ (Teifensuche im Baum). Entweder bildet man für jedes gewählte Problem nur ein Teilproblem für den nächsten Schritt und legt das erste in die Liste zurück, bis alle seine Teilprobleme untersucht werden (wie in der Abbildung 2), oder man verzweigt das Problem vollständig, löscht es aus der Liste und arbeitet mit seinen Teilproblemen weiter.
- nach dem Prinzip „Maximum Upper Bound“ (Breitensuche). Aus der Liste wird immer ein Problem mit der größten oberen Schranke ausgewählt, weil eine seiner Lösungen eventuell am nächsten zu der optimalen Lösung vom Ausgangsproblem ist.
- Als Kombination oder Variation der oben genannten Prinzipien.

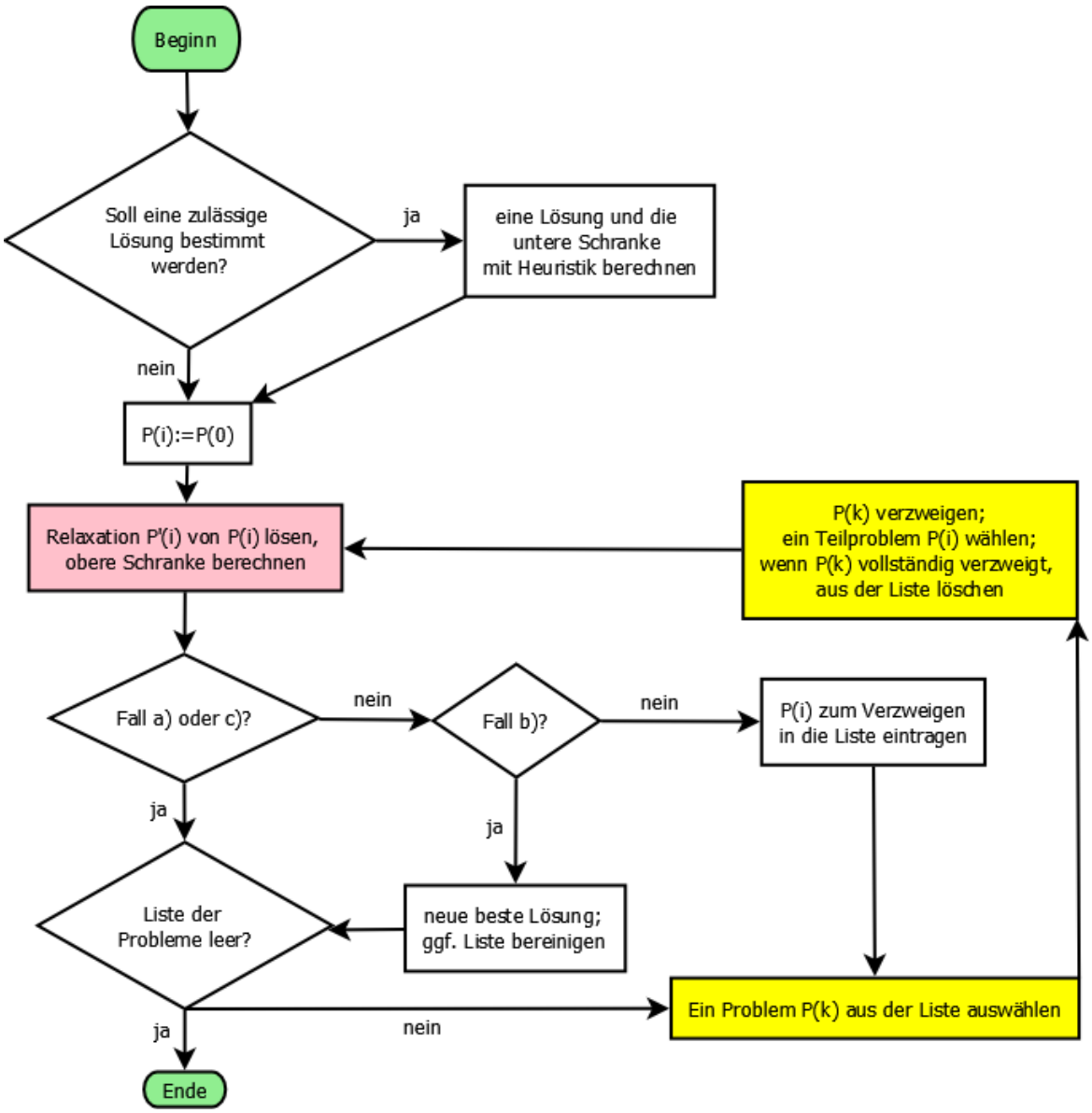

Abbildung 2. Ablauf des Branch-and-Bound-Verfahrens nach (Domschke, et al., 2011)

Wir sehen, dass das Branch-and-Bound-Verfahren nicht nur eins, sondern mehrere aus einem Ausgangsproblem entstandene Optimierungsprobleme betrachtet und löst und dadurch unter Umständen Speicherplatzprobleme verursachen kann. Es könnte jedoch mit anderen Lösungsmethoden kombiniert werden, um bessere Performanz zu erzielen.

2.6.3 DROP-Verfahren

Dieses Algorithmus fängt bei einer unzulässigen Lösung an, bei der alle Studenten ihren ersten Wunsch erfüllt bekommen. Da es dabei eventuell zu viele und/oder zu volle Kurse geben wird, reduzieren wir im nachfolgenden Schritt die Kursanzahl von n auf die gegebene Maximalzahl k und anschließend die Teilnehmerzahlen auf die festgelegten maximalen Werte, sodass die Lösung zumindest im Sinne von (**) zulässig wird.

Pseudocode (Hartmann, 2014):

1) Bilde eine Startzuordnung:

- **a)** Alle Kurse kommen zustande: $y_j := 1, \forall j \in \overline{1,n}$,
- **b)** jeder Student erhält den Kurs mit der höchsten Priorität

2) Wiederhole, bis die Kursanzahl den Wert k erreicht:

- **a)** Berechne für jeden aktuell angebotenen Kurs j die Anzahl $N(j)$ der Teilnehmer, die in keinen anderen der aktuell angebotenen Kurse wechseln könnten, weil diese für sie unzulässig sind.
- **b)** Wähle den Kurs j^* mit $N(j^*) = \min_j \{N(j) \mid y_j = 1\}$. Sollte es mehrere solche Kurse geben, wähle daraus den Kurs j^* mit der kleinsten Teilnehmerzahl. Kurs j^* wird nun gestrichen.
- **c)** Wähle für jeden Teilnehmer aus j^* den bestmöglichen Kurs aus den anderen aktuell angebotenen Kursen, also:
  - **i)** Wenn es mindestens einen noch nicht voll besetzten Kurs mit Priorität > 0 für ihn gibt, wähle den Kurs mit höchster Priorität.
  - **ii)** Wenn es keinen Kurs mit Priorität > 0 für ihn gibt, wähle aus den für diesen Studenten erlaubten Kursen einen zufälligen und möglichst noch nicht voll besetzten Kurs.
  - **iii)** Wenn es keinen erlaubten Kurs für ihn gibt, erhält er keinen Kurs.
- **d)** Streiche Kurs j^*: $y_{j^*} := 0$.

3) Für jeden angebotenen Kurs j, dessen Teilnehmerzahl größer ist als die erlaubte: reduziere die Teilnehmerzahl je Kurs auf die jeweils gegebene Maximalzahl c_j.

- **a)** Wiederhole, bis die Teilnehmerzahl für diesen Kurs zulässig ist:

i) Wähle einen Teilnehmer mit der niedrigsten (nicht negativen) Priorität für j

ii) Weise ihm einen anderen angebotenen Kurs j' zu, für den gilt:

- die maximale Teilnehmerzahl dort ist noch nicht erreicht
- die Priorität dieses Studenten für j' ist so groß wie möglich.
- Wenn es keinen Kurs mit einer positiven Priorität für diesen Studenten mehr gibt, der noch frei ist, dann wähle einen zufälligen Kurs unter den erlaubten,
- Wenn es keinen erlaubten Kurs für ihn gibt, erhält er keinen Kurs.

Es kann also passieren, dass der Student aus seinem Lieblingskurs nach dem Schritt 2 oder 3 gestrichen wird und keinen anderen Kurs bekommt. Das DROP-Verfahren berechnet also nicht zwangsläufig eine gültige Lösung für die Problemstellung (*), aber eine gültige Lösung für (**) und ist in dem Sinne eine Heuristik dem Los-Verfahren ähnlich.

3 Anforderung an eine (fiktive) Planungsumgebung

Wenn man das Ziel verfolgen würde, eine Kursplanungssoftware für den Einsatz an jeweiligen Hochschulen zu erschaffen, sollte man ein solches System aus der Sicht aller Beteiligten betrachten und auf ihre Bedürfnisse eingehen.

Aus dem Blickwinkel der Hochschulleitung sollte das System folgende Anforderungen erfüllen:

1. Eingabe aller relevanten Parameter des Problems ermöglichen, nämlich:
   a. Anzahl der Studenten
   b. Anzahl der angebotenen Kurse
   c. Maximale Anzahl k der Kurse, die stattfindenden sollen
   d. Wie viele Präferenzwerte darf ein Student angeben
   e. Kapazität jedes Kurses
2. Anbindung an eine Datenbank, die Studenten- und Kursdaten verwaltet (in unserem Fall ist das eine Excel-Tabelle), einrichten
3. Zum Zeitpunkt der Kursvergabe die gesammelten Präferenzdaten samt aktueller Parameter mithilfe der Optimierungsalgorithmen abarbeiten
4. Eine oder mehrere resultierenden Zuordnungen der Studenten zu den Kursen ausgeben (in Form der Teilnehmerliste je Kurs), ggf. mit der Möglichkeit, nachträglich manuelle Korrekturen vorzunehmen

5. Diese Zuordnung soll für die Beteiligten *akzeptabel* sein, Abweichungen sind nicht erwünscht:
   a. Es werden höchstens k Kurse stattfinden
   b. Nur wenige oder im Idealfall gar keine Studenten bleiben ohne Kurs
   c. Jeder Student bekommt einen für ihn zulässigen Kurs
   d. Kapazitäten aller Kurse werden nicht überschritten
   e. Maximale Zufriedenheit der Studenten mit den erhaltenen Kursen ist das Ziel
6. Die Benutzer (Planer) sollen über die Bedienung des Systems aufgeklärt werden
7. Unter den gängigen Betriebssystemen (insbesondere Windows-Familie) lauffähig sein
8. Eingabefehler sowie Laufzeitfehler (Berechnungsfehler) anzeigen
9. Die eingesetzte Algorithmen liefern das Ergebnis verhältnismäßig schnell (innerhalb einer Sekunde für ca. 200 Studenten und 10 Kurse) und benötigen wenig Speicher
10. Sie zeigen an, wenn es keine Lösung gibt.

Das System aus der Sicht der Studierenden soll wiederum nachfolgende Eigenschaften aufweisen:

I. Nach der Abgabe der Präferenzen innerhalb wenigen Tagen jedem den zugeteilten Kurs bekanntgeben

II. Bekanntgabe in Form einer übersichtlichen Tabelle (siehe z.B. Tabelle 3), damit jeder weiß, an welchen Kursen er und seine Freunde teilnehmen werden

III. Die Kursverteilung erfolgt unvoreingenommen, also z.B. nicht nach Leistungen, persönlichen Beziehungen zu dem Planer oder nach Alphabet, und wird nicht von der Anmeldungsreihenfolge beeinflusst

IV. Den Studenten ohne Auswahl einen Platz in ihrem Kurs reservieren bzw. bei der Zustimmung der Fakultät nachträglich in die Teilnehmerliste manuell aufnehmen, wenn dieser Kurs schon voll ist.

Name	Vorname	Cape Town	New York	Rio	Dublin	Saigon
Schmidt	Paul	x				-
Müller	Jan			x		-
Busch	Heike				x	-
Krüger	Anke		x			-
Ögultürk	Can			x		-

Tabelle 3. Beispiel für eine Tabelle mit der resultierenden Kurszuordnung

4 Implementierung

In diesem Abschnitt wird berichtet, wie wir die oben genannten Anforderungen bei der Implementierung des Systems umgesetzt haben.

Aufgrund der gewünschten hohen Performanz und Portierbarkeit (Anforderungen 7, 9 und I aus dem Kapitel 3) haben wir uns auf die Programmiersprache C entschieden. Sie ist mit dem von uns verwendeten GLPK-Solver kompatibel und lässt das Parsen der Eingabedatei im csv-Format zu. Dieses Format lässt sich wiederum aus der vorgeschriebenen Excel-Tabelle leicht umwandeln, was die Anforderungen 1 und 3 erfüllt, und die Parameterübergabe erfolgt ebenfalls mithilfe der csv-Datei.

4.1 Funktionsweise des Programms

Der Solver erfordert neben den Eingabedaten (inkl. aktuelle Präferenztabelle des Semesters, vgl. Tabelle 1) auch das mathematische **Modell** des Optimierungsproblems, das mittels der Modellierungssprache GMPL erfasst werden soll. Das Modell beinhaltet Variablendeklaration und definiert die Zielfunktion sowie alle Constraints, wie es im Kapitel 2.2 beschrieben wurde. Da das Modell (sprich eine gmpl-Datei) sich kaum im Laufe der Zeit ändert, im Gegenteil zu **Daten**, also den Präferenztabellen samt Eingabeparameter, die jedes Semester aktualisiert werden müssen, haben wir es von diesen Daten getrennt. Der Solver benötigt jedoch beides, und zwar ebenfalls in Form einer gmpl-Datei, deswegen werden beide Dateien in eine zusammengefasst, um anschließend mithilfe des Solvers gelöst zu werden.

Daraus ergibt sich folgender Verlauf, siehe Abbildung 3. Als Input dienen das Modell und die konkreten Eingabedaten, d.h. die aktuelle Präferenztabelle und alle Parameter des Modells (Anzahl der Studenten, Kurse usw.) sind definiert. Nachdem das C-Programm gestartet wird, parst es die csv-Datei, speichert alle notwendige Parameter für beide Heuristiken und erzeugt aus der gmpl-Datei mit dem Modell eine andere gmpl-Datei, die aus dem Modell und Daten besteht. Diese gmpl-Datei wird als Input für den Solver benutzt. Alle drei Verfahren liefern je

ein Ergebnis, also eine csv-Datei mit der resultierenden Zuordnung der Studenten zu den Kursen sowie Liste der Kurse, die zustande gekommen sind. Dies entspricht den Anforderungen 4 und 5 aus Kapitel 3.

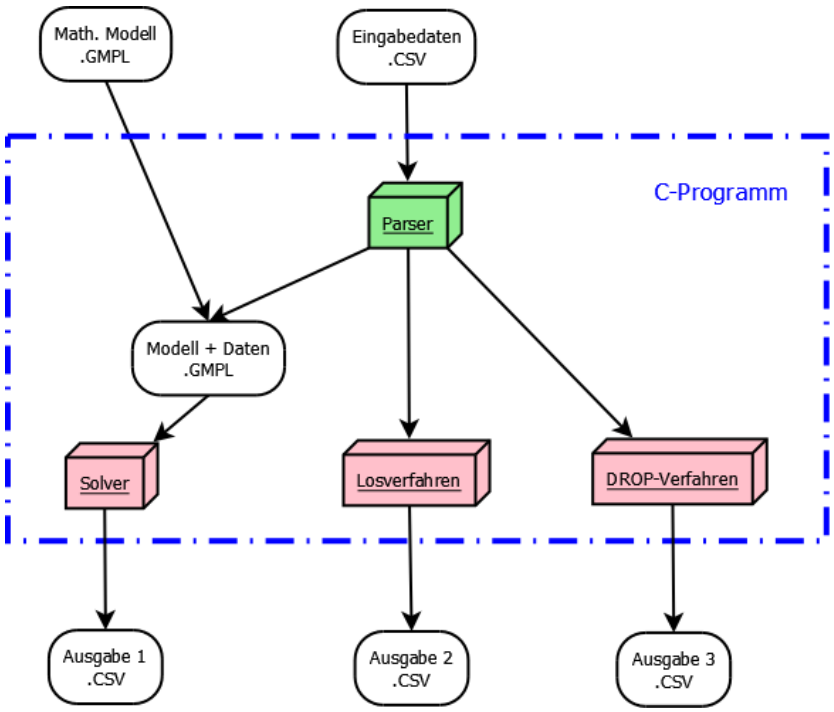

Abbildung 3. Verlauf des Programms

Anforderung 2 schreibt die Möglichkeit vor, die Eingabedatei des aktuellen Semesters auswählen zu können. Diese Funktionalität ist mit einer GUI in C++ realisiert, siehe Abbildung 4.

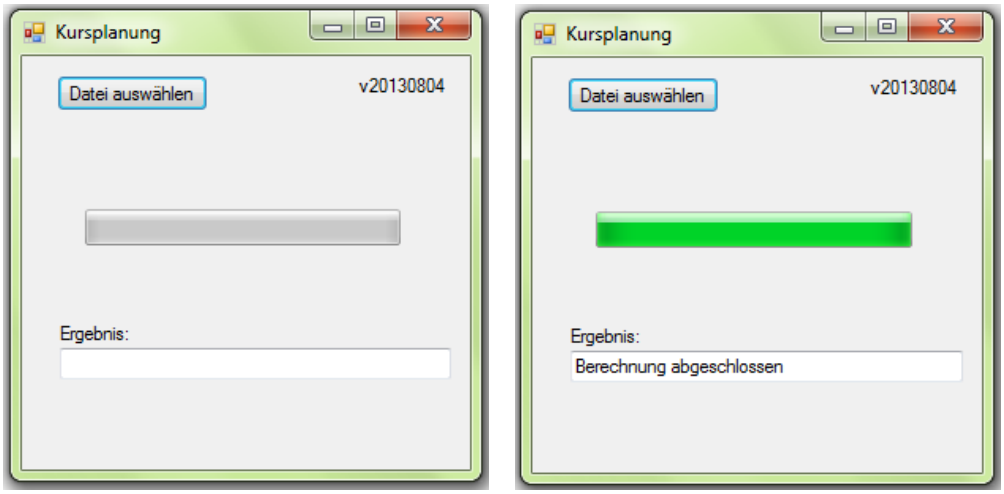

Abbildung 4. GUI

Wenn die csv-Datei ausgewählt wird, startet sich die Anwendung und berechnet die entsprechenden Lösungen. Den Fortschritt beobachtet man in einem Statusbalken in der Mitte, wobei eventuelle Fehlermeldungen (Eingabefehler sowie Laufzeitfehler, siehe Anforderung 8 und 10) in unterem Feld angezeigt werden. Die Ausgabedateien werden in demselben Ordner erzeugt, wo sich die Input-Datei befand. Das Modell liegt dagegen im Ordner der Anwendung.

In dieser Hinsicht sind alle Anforderungen der Planer in unserer Anwendung erfüllt. Auf der anderen Seite haben wir gleichzeitig Anforderungen III und IV der Studenten berücksichtigt, indem wir die Reihenfolge der Eingabedatei nach dem Zufallsprinzip ändern und Studenten ohne Wahl nach vorne vertauschen (siehe Kapitel 2.6.1). Auch Anforderung II lässt sich schnell umsetzen, da wir alle dafür benötigten Daten bereits errechnet haben.

Wenn man beispielsweise zum Testzweck N Dateien auf einem Schlag abarbeiten möchte, stößt man auf das Problem der Namensgebung. Wie sollen diese z.B. hundert Dateien heißen, damit sie einerseits zusammengehören und vom Programm alle erkannt werden, und andererseits verschiedene Ausgaben bewirken? Wir haben uns ein einfaches Schema überlegt:

```
<Basisname><Laufende Nummer>.csv,
```

wobei laufende Nummer alle Werte von 1 bis N darstellt.

Um die Bearbeitung mehrerer Dateien zu ermöglichen, ist der Aufruf des C-Programms von der Konsole ebenfalls möglich:

```
solver.exe Test 100 1
```

Dabei ist „solver.exe“ der Name unseres C-Programms (nicht mit dem Solveraufruf zu verwechseln, der mit anderer Syntax erfolgt), danach folgen der Basisname der csv-Input-Datei, Anzahl solcher Dateien (N) und die Variante des Losverfahrens, die wir anwenden wollen. Zur Erinnerung, im Kapitel 2.6.1 wollten wir feststellen, welche Auswirkung die Wahl zwischen den beiden Formeln zum Berechnen des Beliebtheitsgrads (B_1 und B_2) auf die Kursauswahl hat. Der dritte Parameter nimmt also die Werte 1 oder 2 wahr. Im obigen Beispiel werden alle Dateien von „Test1.csv“ bis „Test100.csv“ abgearbeitet, und zwar mit dem Solver, Losverfahren mit B_1 (Beliebtheit anhand aller Prioritätswerte) und dem DROP-Verfahren.

Um Solver aus dem C-Programm zu starten, benötigen wir zusätzlich eine batch-Datei des folgenden Inhalts:

```
"C:\Program Files (x86)\GnuWin32\bin\glpsol" --math Test1_Modell_Daten.gmpl
```

Das obige Pfad zeigt an, am welchen Ort der Solver installiert wurde, der Befehl --math liest die angegebene gmpl-Datei, wo in unserem Fall das Modell mit den Daten gespeichert ist und der Solveraufruf erfolgt.

4.2 Datenmodell

Um das bessere Verständnis zu verschaffen, geben wir hier einen kurzen Ausschnitt aus den oben beschriebenen Dateien.

4.2.1 Modell.gmpl

(geschrieben in der GMPL-Sprache (Hartmann, 2014))

```
1. param m integer;       /* Studenten */
2. param n integer;       /* Kurse */
3. set W;                 /* Prioritaetswerte */
4.
5. param a{1..m,1..n};       /* Praeferenz von Student i fuer Fach j */
6. param S := 4;             /*Echte Obergrenze fuer Praeferenzwerte */
7. param b{i in 1..m, j in 1..n} := if a[i,j] > 0 then S-a[i,j] else a[i,j];
                          /*Prioritaet von Student i fuer Kurs j*/
8. param C{1..n};    /* Kapazitaet von Kurs j */
9. param K;          /*max.Anzahl Kurse, die zustande kommen sollen*/
10.
11. param Titel{1..n} symbolic;  /* Titel des Kurses */
12. param Name{1..m, 1..2} symbolic;    /* Name von Student i; Spalte
    1=Nachname, Spalte 2=Vorname */
13.
14. var x{1..m,1..n} binary; /*1, wenn Student i Kurs j bekommt, sonst 0*/
15. var y{1..n} binary;      /*1, wenn der Kurs j angeboten wird, sonst 0*/
16.
17. maximize Gesamtprio: sum{i in 1..m, j in 1..n} b[i,j] * x[i,j];
18.
19. s.t. Max_Anzahl_Kurse_gesamt:             sum{j in 1..n} y[j] <= K;
20. s.t. Ein_Kurs_pro_Student{i in 1..m}: sum{j in 1..n} x [i,j] = 1;
21. s.t. Kein_unzulaess_Kurs{i in 1..m}: sum{j in 1..n} a[i,j] * x[i,j]>= 0;
22. s.t. Kapazitaet_je_Kurs{j in 1..n}: sum{i in 1..m} x[i,j] <= C[j] * y[j];
23.
24. solve;
```

4.2.2 Input.csv

Diese Datei bekommen wir von der Hochschulleitung, und ihre Struktur bleibt immer gleich (R, T, V stehen für Namen, S, U, W für Vornamen, X, Y, Z für Kurstitel):

```
1. Anzahl Studenten;200;;;;;;;;;
2. Anzahl Wahlfaecher;10;;;;;;;;;
3. max. Wahlfaecher;8;;;;;;;;;
4. ;;;;;;;;;;
5. Wahlfach;;X;Y;…;Z
6. max. Teilnehmer;;30;20;…;25
7. R;S;2;0;1;3;0;0;0;0;0;0
8. T;U;3;1;0;0;0;0;0;2;0;0
9. …
10. V;W;0;1;2;3;0;0;0;0;0;0
11. ;;;;;;;;;;
12. Hinweis:;;;;;;;;;;
13. 1;erste Praeferenz;;;;;;;;;
14. 2;zweite Praeferenz;;;;;;;;;
15. 3;dritte Praeferenz;;;;;;;;;
16. 0;keine Praeferenz, Teilnahme aber moeglich;;;;;;;;;
17. -1;Teilnahme nicht erlaubt;;;;;;;;;
```

4.2.3 Input_Modell_Daten.gmpl

Bemerkung: Reihenfolge der Studenten in dieser Datei ist per Zufall vertauscht. Hier stehen A, C, E für Namen, B, D, F für Vornamen.

```
1. <Modell.gmpl>
2. data;
3. param m := 200;
4. param n := 10;
5. param K := 8;
```

```
6. set W :=  1     2     3     0     -1;
7. param a:
8.     1     2     3     4     5     6     7     8     9     10 :=
9. 1   1     0     2     0     3     0     0     0     0     0
10.2   1     -1    3     -1    -1    2     -1    -1    -1    -1
11.…
12.2001      3     2     0     0     0     0     0     0     0
13.;
14.
15.param C :=
16.1   30
17.2   20
18.…
19.10  25
20.;
21.
22.param Titel :=
23.1   'X'
24.2   'Y'
25.…
26.10  'Z'
27.;
28.
29.param Name :
30.    1     2 :=
31.1   'A'    'B'
32.2   'C'    'D'
33.…
34.200 'E'    'F'
35.;
36.
37.end;
```

4.2.4 Output.csv

Sie ist für jeden Algorithmus gleich aufgebaut:

```
1. Zustandegekommen:
2. Titel;Anz. Kurse;Teilnehmerzahl
3. X;1;30
4. Y;1;20
5. …
6. Z;1;3
7.
8. Nicht zustandegekommen:
9. I
10. J
11.
12. Statistik:
13. Praeferenz;Anzahl;Anteil
14. 1;145;0.725
15. 2;48;0.240
16. 3;7;0.035
17. 0;0;0.000
18. -1;0;0.000
19. keine;0;0.000
20.
21. X
22. Name;Vorname;Praeferenz
23. A;B;1
24. C;D;1
25. …
26. Y
27. Name;Vorname;Praeferenz
28. E;F;3
…<usw.>
```

5 Generierung der Testdaten

Um eine Basis für die umfassende Auswertung der drei Lösungsverfahren zu verschaffen und alle denkbare Konstellationen zu überprüfen, haben wir einen Datengenerator in C implementiert, der die Input-Dateien für die bestehende Kursplanungsanwendung erzeugt. Diese Dateien sind genau so aufgebaut wie Input.csv (siehe Kapitel 4.2.2), nur die Namen der Studenten und Kursen sind anonymisiert und durchnummeriert.

Im Folgenden wird ein Algorithmus beschrieben, der eine, für jede Input-Datei unterschiedliche Präferenztabelle generiert. Sei m – Anzahl der Studenten, n – Anzahl der Kurse, $P = \{1, 2, 3, 0, -1\}$ – Menge der Präferenzwerte, Q – Anteil der Studenten, die keine freie Wahl haben. Der letzte Parameter besagt, dass jeder $[{}^{m}/_{Q}]$ -te Student (also einer aus $[{}^{m}/_{Q}]$) nicht für alle Kurse zugelassen ist und demzufolge nicht jeden Kurs wählen darf.

Pseudocode:

Für jeden Student i von 1 bis m:

1. Sei r – eine zufällige Zahl zwischen 1 und Q.
2. Wenn $r = 1$, dann hat der aktuelle Student keine freie Wahl, sondern ist nur für s Kurse zugelassen.
   a. Setze s – eine zufällige Zahl zwischen 1 und 3.
3. Wenn $r > 1$, dann kann der Student frei 3 Kurse wählen.
   a. Setze $s = 3$.
4. Wähle s zufällige Kurse aus der Kursliste L.
5. Trage diese Kurse mit den entsprechenden Präferenzen (1, 2 und/oder 3) in die Präferenztabelle.
6. Wenn $r = 1$, dann bekommen alle anderen Kurse die Präferenz -1, sonst 0.

Wir merken, dass die Anzahl der Studenten ohne Wahl laut dem Schritt 2.a im Durchschnitt den Wert $[Q/3]$ erreicht.

Implementiert man diesen Algorithmus wie eben beschrieben, merkt man erst bei dem Vergleich der drei Lösungsverfahren, dass das Losverfahren eine Lösung liefert, die fast genauso gut ist wie die optimale Lösung des Solvers. Das liegt daran, dass alle Kurse nahezu gleich beliebt sind, d.h. mit der gleichen Wahrscheinlichkeit von jedem Student gewählt werden. Dadurch wird das Problem der überbuchten Kurse nicht entstehen, und das Losverfahren, das selbst nach einem Zufallsprinzip vorgeht, findet fast jedem Studenten einen zulässigen Platz.

In der Praxis wird eine solche Situation kaum entstehen, dass alle angebotenen Kurse gleich beliebt sind. Aus diesem Grund haben wir nach Möglichkeiten gesucht, die unterschiedliche Beliebtheit der Kurse auszudrücken und in den Generator zu integrieren.

Unsere Annahme besteht darin, dass die Kursbeliebtheit normal verteilt ist. Deswegen fiel unsere Wahl auf die Gauß-Funktion, die diese Normalverteilung beschreibt:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

Aufgrund der gewünschten Symmetrie haben wir $\mu = 0$ festgelegt. Die Gauß-Funktion ist somit nur von σ abhängig: je höher σ wird, desto flacher die Grafik der Funktion fällt (siehe Abbildung 5). Außerdem haben wir ein Koeffizient hinzugefügt, um die Höhe der Grafik festzulegen:

$$f^*(x) = \frac{m}{2} \cdot \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x}{\sigma}\right)^2}$$

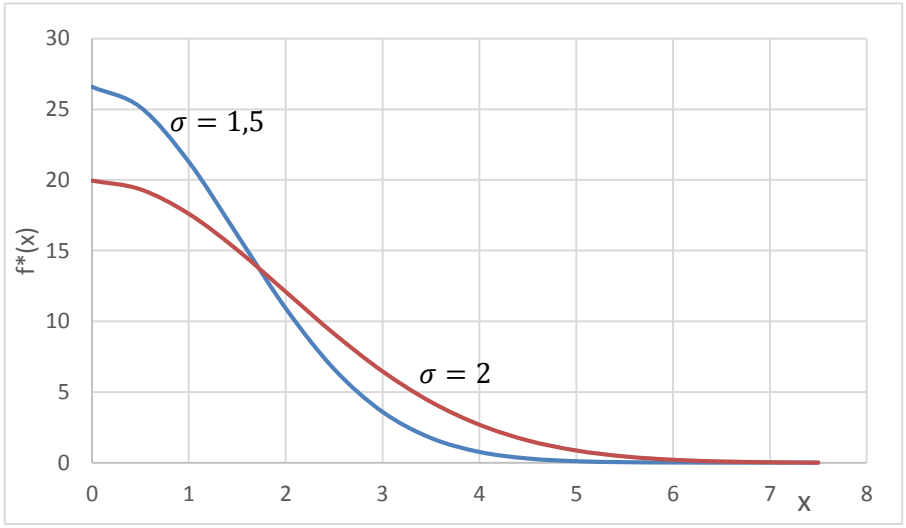

Abbildung 5. Gauß-Funktion

Wir definieren außerdem ein Intervall $[0, V]$ auf der x-Achse, mithilfe dessen die Beliebtheitswerte ermittelt werden. Und zwar wird für jeden Kurs j ein $x = x_j = \frac{j}{n}V \in [0, V]$ und $f^*(x)$ berechnet. In der Abbildung 5 ist $V = 5$. Wenn wir auf das Intervall verzichten würden, hätten wir bei sehr großen Werten n einen hohen Anteil an Kursen, deren Beliebtheit infinitesimal ist.

Des Weiteren müssen wir die Beliebtheitswerte normieren, damit die Summe alle Werte 1 bzw. 100% ist. Somit wird folgender Wert als die Beliebtheit von j genommen:

$$BL(j) = \frac{f^*(x_j)}{\sum_j f^*(x_j)} \cdot 100\%$$

Wenn wir nun das Intervall $[0, 100]$ in n Abschnitte aufteilen, deren Länge jeweils $BL(j)$ ist, mit Kursennamen „beschriften“ und eine zufällige Zahl innerhalb dieses Intervalls wählen, dann fällt diese Zahl mit der Wahrscheinlichkeit $BL(j)$ auf den Abschnitt des Kurses j. Wenn wir nach diesem Prinzip die Kursliste L aus dem Generator-Algorithmus aufbauen, dann wird jeder Kurs seiner Beliebtheit entsprechend (mit jeder Präferenz) ausgewählt. Die tatsächliche Verteilung der Kursbeliebtheit nach der Ausführung des Algorithmus ist allerdings aufgrund des Zufälligkeitsfaktors von der vorgegebenen abweichend, siehe Abbildung 6.

Demzufolge sind folgende Parameter für die Datengenerierung notwendig:

- ✓ Dateiname
- ✓ Anzahl der zu generierenden Dateien A
- ✓ Anzahl der Studenten m
- ✓ Anzahl der Kurse n
- ✓ Maximale Anzahl k der Kurse, die stattfindenden sollen
- ✓ maximale Anzahl c der Teilnehmer pro Kurs (Kapazität des Kurses)
- ✓ σ und V für die Gauß-Funktion
- ✓ Q für die Bestimmung der Studenten, die keine freie Wahl haben.

Um den Aufruf des Generators nicht mit vielen weiteren Parametern zu überlasten, haben wir uns entschieden, die maximale Kurskapazität als eine Konstante festzulegen, die für alle Kurse gleich ist. Der Generator wird wie folgt von der Konsole aufgerufen:

```
generate.exe Test 100 200 10 8 30 1.5 5 10
```

Die resultierenden Dateinamen sind: „Test1.csv“, „Test2.csv“ usw. bis „Test100.csv“.

Auch hier, wie in C-Programm aus dem Kapitel 4.1, wird geprüft, ob diese Eingabeparameter den festgelegten Kriterien entsprechen:

- Dateiname nicht leer,
- $A, m, n, k, c, Q \in \mathbb{Z}^+$,
- $k \leq n$,
- $\sigma, V \in \mathbb{R}^+$.

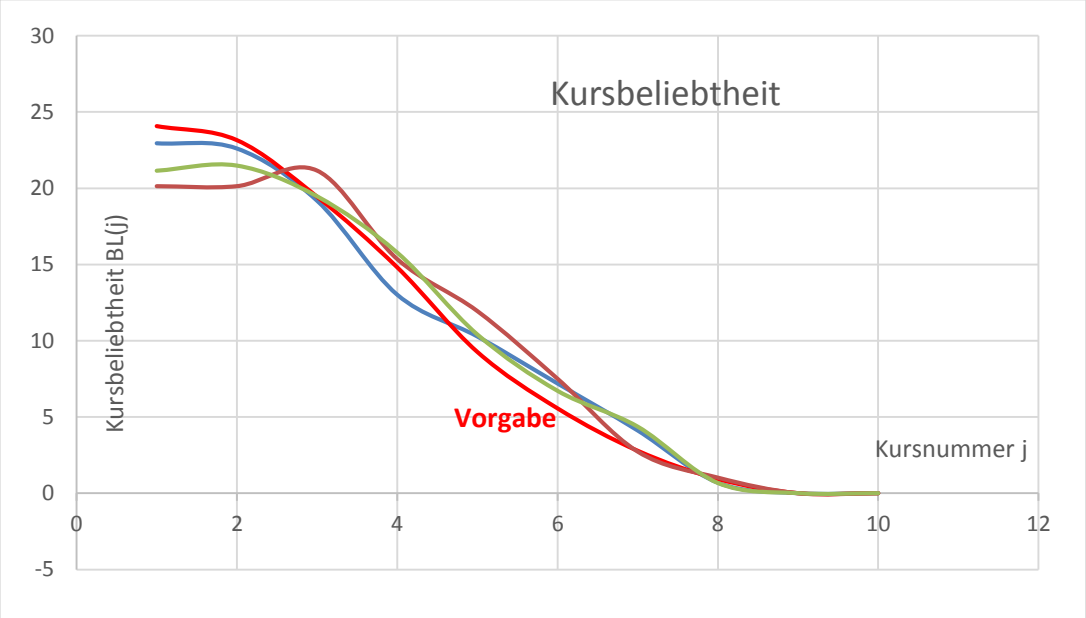

Abbildung 6. Tatsächliche Kursbeliebtheit im Vergleich mit den Vorgabewerten

6 Auswertung und Vergleich der Lösungsverfahren

Wir haben bereits festgestellt, dass unsere Anwendung den hochschulspezifischen Anforderungen entspricht. In diesem Kapitel werden wir die wichtigsten Aspekte und Kriterien erläutern, die dabei helfen, die Auswahl eines besten der drei Verfahren zu erleichtern.

6.1 Zufriedenheit

Zunächst werden wir unsere drei Verfahren anhand der echten Daten auswerten, die wir von der Hochschule bekommen haben. Aufgrund des Datenschutzes werden hier die Studentennamen nicht in Betracht gezogen.

Es werden die Angaben der insgesamt 160 Studenten bearbeitet und 11 Kurse bzw. Ausflüge angeboten: Cape Town, New York, Rio, Kuala Lumpur, Saigon, Rotterdam, Dublin, Katmandu, Paris, Hongkong, Kabul. Maximale Teilnehmerzahl ist für alle Kurse auf 20 festgelegt, und es sollen lediglich 9 aus 11 Kurse stattfinden. Anzahl der Studenten ohne Wahl beträgt 28 und betrifft die Kurse Rotterdam, Dublin, Katmandu und Hongkong mit entsprechend 8, 16, 1 und 3 Plätzen, die für diese Studenten „reserviert“ werden müssten.

Aus diesen Informationen kann man schließen, dass es mindestens eine zulässige Lösung existiert. Den Studenten ohne Auswahl wird nämlich ein Platz in ihrem Wunschkurs angeboten, die restlichen 52 freien Plätze dort werden mit anderen (beliebigen) Studenten besetzt. Somit finden 4 Kurse mit maximaler Auslastung statt (Rotterdam, Dublin, Katmandu und Hongkong), und die verbleibenden 80 Studenten kann man in weitere 4 Kurse verteilen. Da nur 8 aus 11 Kurse stattfinden, voll ausgebucht sind und alle Studenten einen Platz in einem für sie zulässigen Kurs bekommen, ist diese Lösung gültig. Folglich existiert auch eine

(oder mehrere) optimale Lösung (siehe Kapitel 2.4), die mithilfe des Solvers geliefert wird. Diese können wir der Tabelle 4 entnehmen.

Bei jedem Lauf unserer Anwendung wird eine Statistik ausgegeben, siehe Tabelle 4. Diese besagt, wie viele Studenten mit welcher Präferenz einen Kurs bekommen haben, und gibt je einen resultierenden Wert der Zielfunktion pro Verfahren an. Zur Erinnerung, die beiden Losverfahren unterscheiden sich in der Bestimmung der Kurse, die nicht stattfinden, mithilfe des Beliebtheitsgrads B_1 und B_2, siehe Kapitel 2.6.1.

Präferenz	1	2	3	0	-1	Zielfunktion	Kriterium 1
Solver	100	56	4	0	0	416	36
Los2	103	39	12	6	0	399	22
Los1	96	33	18	13	0	372	-12
DROP	116	20	3	2	19	372	84

Tabelle 4. Statistik anhand der echten Daten

Anhand dieser Tabelle sehen wir, dass obwohl das DROP-Verfahren den meisten Studenten ihre erste Präferenz bietet, ist er einer der schlechtesten im Bezug zu dem erreichten Wert der Zielfunktion, und 19 Studenten sind sogar ohne Kurs geblieben. Im Gegensatz dazu ermöglicht der Solver allen Studenten, an einem ihrer Wunschkurse teilzunehmen.

Was ist in unseren Augen die beste dieser vier Lösungen? Mit welcher Lösung wären die Planer und die Studenten am liebsten zufrieden? Eine eindeutige Antwort auf diese Frage gibt es nicht, da jeder seine subjektiven **Kriterien** bei der Entscheidung anwendet, z.B.:

1) Man würde einerseits sagen, die beste Lösung sei diejenige, bei der möglichst viele Studenten ihre erste Präferenz bekommen, und andere Präferenzwerte sind nicht erwünscht. Demnach gibt es einen Punkt für jede Erfüllung des ersten Wunsches, und je höher die gesamte Punktzahl ist, desto besser die Lösung ist. Für die Erfüllung des zweiten Wunsches wird ein Punkt abgezogen, des dritten – zwei Punkte, und für den nicht gewählten Kurs (Präferenz 0) – drei Punkte. Wenn allerdings einige Studenten keinen Kurs bekommen, dann lässt das sich einfach nicht vermeiden.

2) Andererseits würde man jede Präferenz akzeptieren und dabei möglichst die besten Wünsche erfüllen. Dieses Kriterium entspricht unmittelbar der Berechnung der Zielfunktion, wo alle erfüllten gewichteten Präferenzen (genauer gesagt alle Prioritäten) summiert werden.

Nach dem ersten Kriterium ist die Lösung des DROP-Verfahrens die beste, siehe letzte Spalte der Tabelle 4. Man merkt auch, dass diese Lösung die niedrigsten Zahlen für Präferenzwerte 2, 3 und 0 enthält. Das liegt an dem Ablauf des Verfahrens: Zunächst bekommen alle Studenten ihre erste Präferenz, und damit ist das Kriterium zu 100 % erfüllt. In den nächsten

Schritten werden einige Kurse gestrichen bzw. ihre Kapazität abgeglichen, die Betroffenen sind in andere Kurse verteilt und bekommen dort eventuell keinen Platz. Dadurch bleiben aber die meisten in ihrem Wunschkurs, weshalb das DROP-Verfahren so gut abschneidet.

Das zweite Kriterium (sprich Zielfunktion) stellt das DROP- und das erste Losverfahren auf den letzten Platz.

Die interessante Erkenntnis dabei ist, dass das zweite Losverfahren (bei dem diejenige Kurse zustande kommen, die am meisten mit Präferenz 1 gewählt werden) deutlich besser ist als das erste (der alle Präferenzen berücksichtigt). Das erklärt sich dadurch, dass beim ersten Losverfahren auch ein solches Kurs j stattfinden kann, der weniger mit Präferenz 1 gewählt wurde, dafür aber beispielsweise sehr oft mit Präferenz 3. Da das Verfahren jedoch versucht, jedem Student einen Kurs mit der höchsten Priorität zuzuweisen, wird er an dieser Stelle viele Studenten mit ihrer dritten Wahl zufriedenstellen müssen, weil sie sonst keine höhere Priorität bekommen würden (die entsprechenden Kurse finden eventuell nicht statt, da sie weniger gewählt wurden als j).

Das zweite Losverfahren wählt im Gegenteil die Kurse aus, die die meisten Studenten mit Präferenz 1 bekommen wollen. Davon müssen einige in andere Kurse verteilt werden, wenn kein freier Platz mehr da ist, aber die meisten werden ihren Wunschkurs bekommen.

Aufgrund des Nichtdeterminismus, der in allen drei Verfahren vorhanden ist (auch im Solver, der zwischen mehreren optimalen Lösungen wählen muss), liefern sie nach jedem Lauf nicht unbedingt dasselbe Ergebnis. Um festzustellen, ob die oben genannte Überlegungen nicht nur in diesem Einzelfall gültig sind, haben wir die Algorithmen mithilfe der generierten Daten getestet und die durchschnittliche Werte über 100 Input-Dateien ausgerechnet. Bei diesem Test haben wir dieselben Eingabeparameter verwendet, wie bei den echten Daten:

```
generate.exe Test 100 160 11 9 20 0.6 1 28
```

Die letzten drei Parameter lassen sich aus den folgenden Werten ermitteln: Beliebtheit $BL(j)$ für jeden Kurs j (also Anzahl der Studenten, die diesen Kurs mit einer positiven Präferenz gewählt haben) und Anzahl der Studenten ohne Wahl. Die Beliebtheitskurven sind in der Abbildung 7 dargestellt. Der Parameter $V = 1$ passt die entsprechende Gauß-Funktion mit $\sigma = 0{,}6$ (Erfahrungswert) an die praxisnahe Beliebtheitskurve an.

Präferenz	1	2	3	0	-1	Zielfunktion	Kriterium 1	Laufzeit (ms)
Solver	134,55	24,95	0,49	0	0	454,04	108,62	471,67
Los2	135	20,9	2,41	1,6	0,1	449,11	104,48	1,28
Los1	133,16	19,64	4,17	2,92	0,13	442,8	96,42	2,24
DROP	140,19	13,28	0,8	1,88	3,85	444,08	119,67	1,18

Tabelle 5. Statistik anhand der generierten Daten mit denselben Parametern

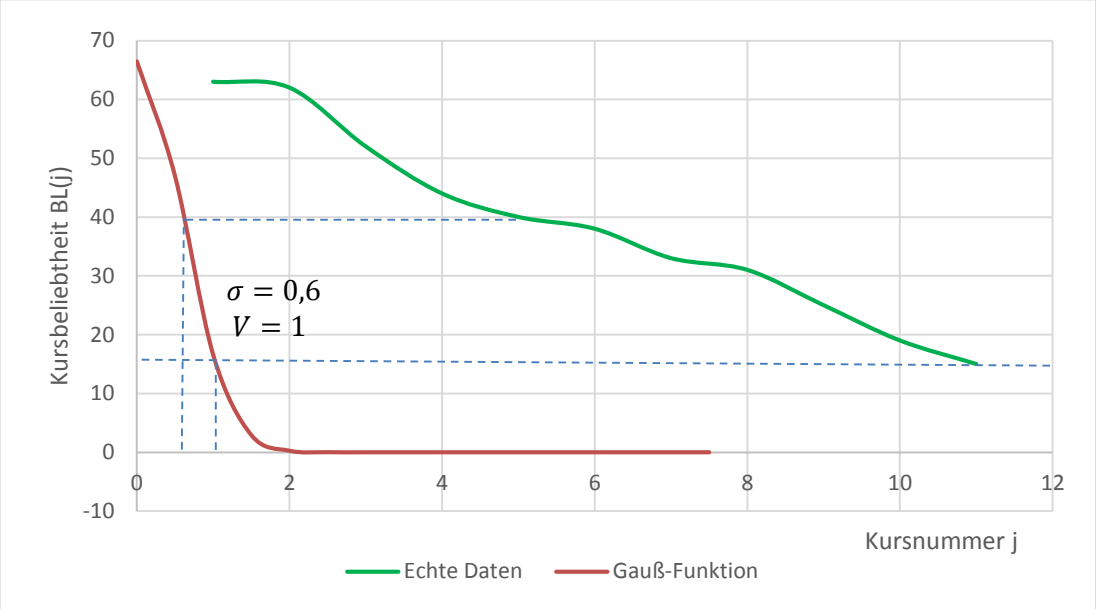

Abbildung 7. Approximation der Beliebtheitskurve aus der Praxis mithilfe der Gauß-Funktion

In der Tabelle 5 sind die Ergebnisse der Datengenerierung und Auswertung zusammengefasst. Es fällt einem auf, dass obwohl die analysierte Zusammenhänge zwischen den Verfahren noch gültig sind, bekommen deutlich mehr Studenten ihre Wunschkurse, was sich auch in den Werten der Zielfunktion und des Kriteriums 1 wiederspiegelt. Man kann also davon ausgehen, dass sich die Daten aus der Praxis oft sehr stark unterscheiden, aber auch im schlechtesten Fall bleibt das beschriebene Verhältnis bestehen.

6.2 Laufzeit

Wie in Kapitel 2.4 bereits angesprochen, kann der Solver aufgrund der Zugehörigkeit des Kursplanungsproblems zu der Klasse von NP-vollständigen Optimierungsproblemen keine optimale Lösung in polynomialer Zeit finden. Er benötigt also im allgemeinen Fall einen exponentiellen Zeitaufwand, der in Einzelfällen durch Variation oder Kombination der Lösungsstrategien reduziert werden kann.

Dagegen lässt sich die Zeitkomplexität der beiden Heuristiken leicht aus dem Code ermitteln. Sei m – Anzahl der Studenten und n – Anzahl der Kurse. Andere Eingabeparameter nehmen im schlechtesten Fall diese beiden Werte an und werden als solche nicht betrachtet.

- Losverfahren: Die gesamte Zeitkomplexität ergibt sich aus $\mathcal{O}(m \cdot n + n^3)$ für Schritt 1 in zwei Etappen und $\mathcal{O}(m \cdot n^2)$ für Schritt 2 und beträgt $\mathcal{O}((m + n) \cdot n^2)$.

- DROP-Verfahren: $\mathcal{O}(m \cdot n)$ (Schritt 1), $\mathcal{O}(m \cdot n^5)$ (Schritt 2) und $\mathcal{O}(m \cdot n^4)$ (Schritt 3) resultieren in der gesamten Komplexität $\mathcal{O}(m \cdot n^5)$.

Es empfiehlt sich also bei sehr großen Probleminstanzen eine gute und schnelle Lösung mithilfe der Heuristiken zu finden und diese als Basis für die exakten Verfahren wie Branch-and-Bound zu nutzen.

Was den Speicherbedarf anbetrifft, verfügen wir über Statistiken, die der Solver in der Konsole ausgibt, siehe Abbildung 8. Demnach benötigt der Solver für die Lösung einer Probleminstanz der Größe wie im Kapitel 6.1 ca. 2,6 MB.

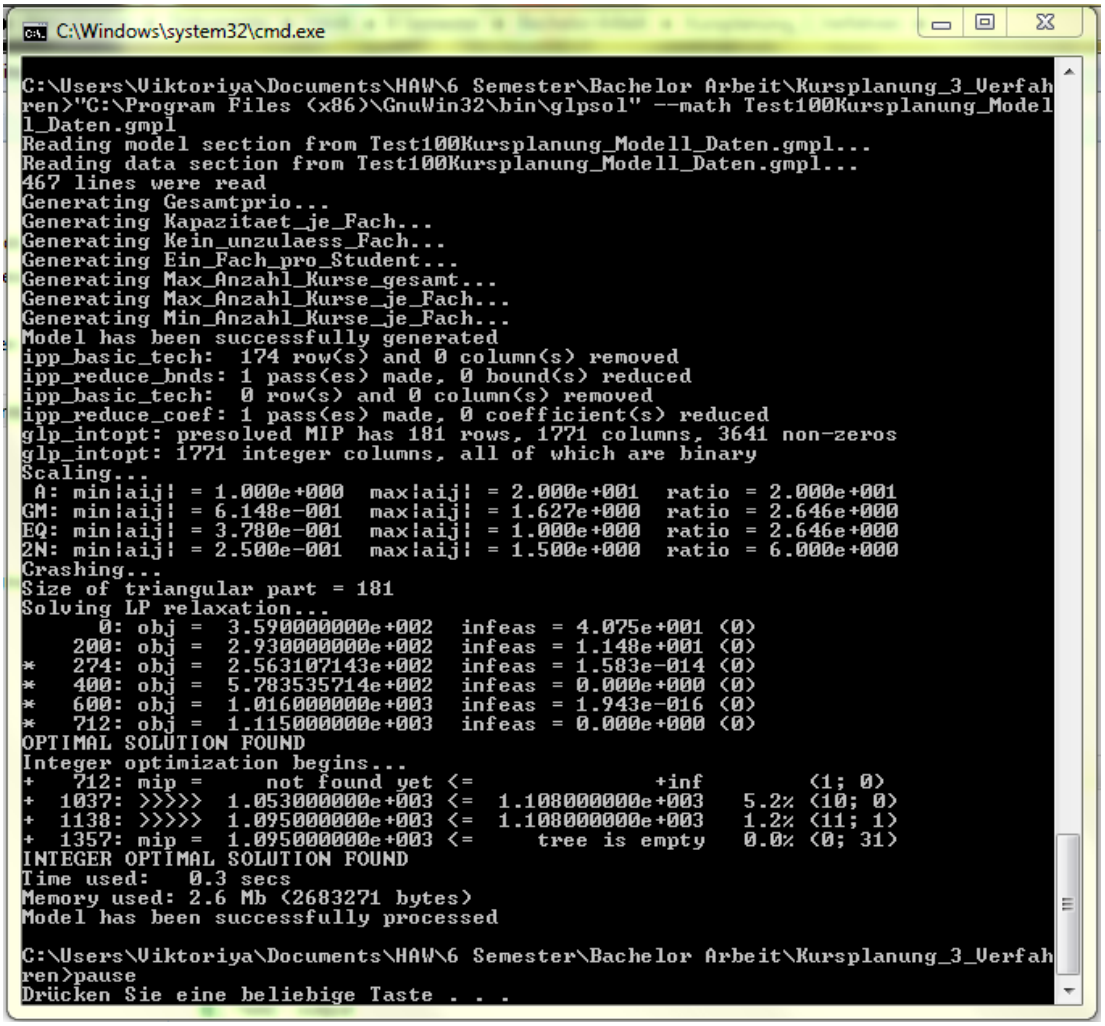

Abbildung 8. Ausgabe des Solvers

6.3 Skalierbarkeit

Spricht man von der Skalierbarkeit des Optimierungsproblems, wird die Veränderung der Laufzeit bei steigender Anzahl der dazugehörigen Variablen und Constraints gemeint, weil von diesen Werten die Eingabegröße abhängig ist.

Anzahl der Variablen im Kursplanungsproblem (*) beträgt $m \cdot n + n = (m+1)n$; Anzahl der Constraints ist $1 + m + m + n + (m+1)n = 1 + 2(m+n) + m \cdot n$, wobei die ersten vier Summanden den Constraints (1)-(4) entsprechen, und der letzte Summand steht für das binäre Constraint $x_{ij}, y_j \in \{0,1\}$. Beide Problemgrößen gehören damit zur Größenklasse $\mathcal{O}(m \cdot n)$, d.h. das Problem (*) skaliert quadratisch bei steigenden m und n.

Wenn wir nur m und n variieren wollen, gibt es für die restlichen Parameter bestimmte Wertebereiche, die eingehalten werden müssen, um das Ergebnis nicht zu beeinträchtigen. Beispielsweise beträgt eine plausible Kursausfallrate 50 %, sodass jeder Kurs mit der Wahrscheinlichkeit ½ stattfindet. Daraus ergibt sich $n \geq k \geq [n/2]$. Außerdem sollte Kapazität jedes Kurses nicht zu restriktiv sein, damit alle Studenten theoretisch einen Platz bekommen können, unabhängig davon, ob sie dort zugelassen sind: $m \geq c \geq [m/k]$. Des Weiteren haben wir $Q = 5$, $\sigma = 1$ und $V = 2$ festgelegt. Der Einfluss dieser drei Werte auf die Laufzeit wird im Kapitel 7 näher untersucht.

Die Ergebnisse dieser Auswertung, nämlich die Laufzeit im Durchschnitt über 100 generierten Input-Dateien in Millisekunden, sind in der Tabelle 6 zu sehen. Für den Lauf mit $m = 300$, $k = 50$ benötigte der Solver 930 MB Speicher und ca. 16 Minuten Zeit. Die Grenze der Ausführbarkeit liegt bei $m = 330$ und $k = 60$, nach ca. 3 Stunden 40 Minuten liefert der Solver eine Fehlermeldung, dass kein Speicher mehr verfügbar ist. Die genutzte Speicherkapazität lag bei 1766,3 MB. Laufzeit steigt also exponentiell, siehe Abbildung 9.

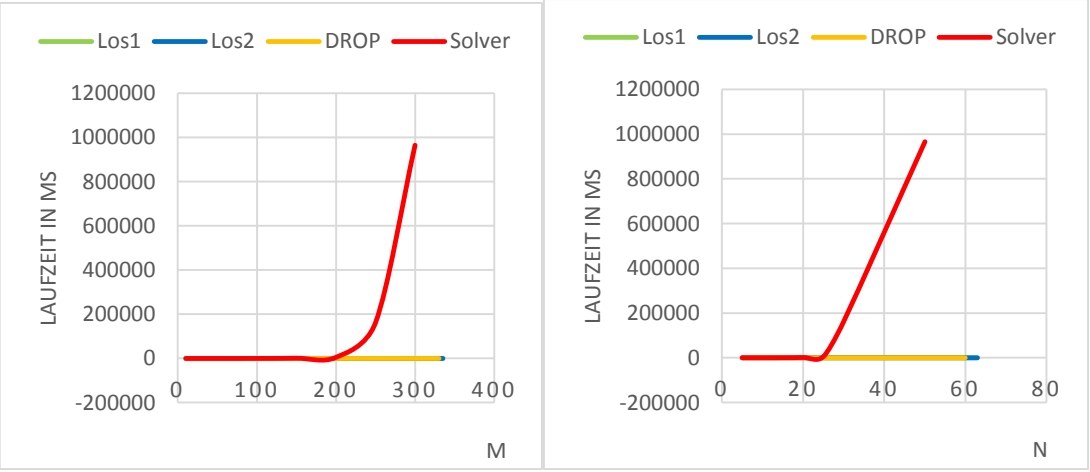

Abbildung 9. Laufzeit in Abhängigkeit von m und n (mit Solver)

m	n	k	c	Solver	Los1	Los2	DROP
10	5	4	7	164,1	1	0,9	0,7
50	10	7	10	215,8	1,1	0,8	0,9
100	15	12	10	407,5	1,1	1,1	1
150	20	15	15	1379,8	1,4	1,6	1,2
200	25	19	15	5317,5	1,5	1,8	1,7
250	30	20	18	162378	1,6	2	2,6
300	50	40	10	965443	3	4	4,3
330	60	45	10	---	4	6,1	8,1
335	63	45	10	---	5	7	---
340	65	50	10	---	---	---	---
350	70	50	10	---	---	---	---

Tabelle 6. Laufzeit (ms) in Abhängigkeit von m und n

Auch die beiden Heuristiken stürzen spätestens bei 340 Studenten und 65 Kursen ab. Das liegt aber eher an ungeschickten Speichernutzung als an den Verfahren selbst, da ihre Laufzeit bei großen Instanzen immer noch im Millisekundenbereich bleibt, siehe Abbildung 10.

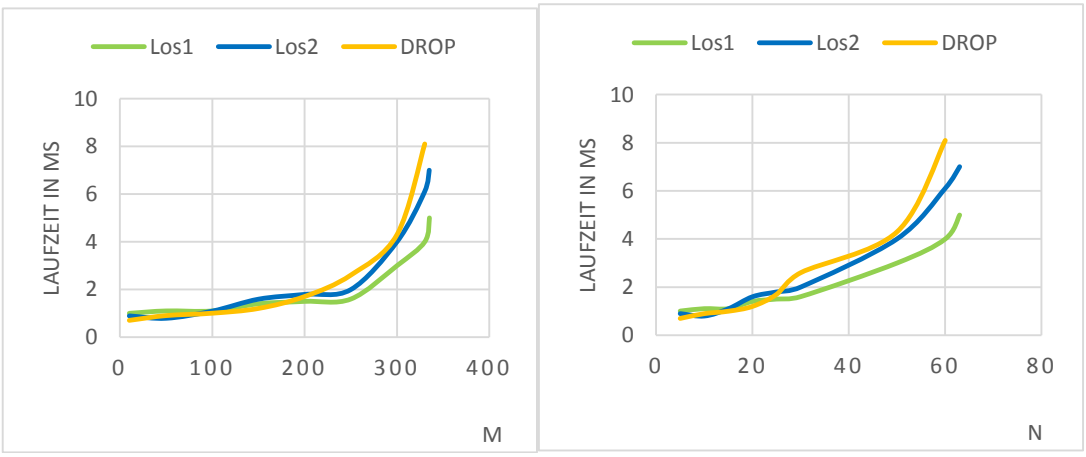

Abbildung 10. Laufzeit in Abhängigkeit von m und n (ohne Solver)

6.4 Erweiterbarkeit

Bisher haben wir uns mit einem Kursplanungsproblem auseinandergesetzt, das an einer Hamburger Hochschule entstanden ist. Daher stellt sich die Frage, ob die drei eingesetzten Verfahren nur für diese Hochschule geeignet sind oder ob sie sich auf abweichende Anforderungen anpassen lassen.

Aus diesem Grund werden wir hier auf die teilweise ähnliche Situation an der Hochschule für Angewandte Wissenschaften Hamburg (kurz – **HAW**) eingehen und zeigen, dass auch unter anderen Bedingungen die beschriebenen Algorithmen anwendbar sind. An der HAW werden Bachelor-Studierende des Departments Informatik ein Mal pro Semester aufgefordert, Wahlpflichtfächer (*WPs*), Projekte (*POs*) und gesellschaftswissenschaftliche Fächer (*GWs*) zu wählen.

Ohne Beschränkung der Allgemeinheit werden wir die Zuweisung von Studenten zu WPs näher betrachten. Jeder Student gibt seine drei Präferenzen für die WPs online an. In einigen Studiengängen muss man zwei WPs pro Semester bekommen, sodass sie nicht an demselben Tag platziert sind. In diesem Fall nennen die Studenten fünf Präferenzen anstatt drei. Dabei sind diese fünf Präferenzen nicht differenziert, d.h. es ist nicht bekannt, welche Werte ein Student für seinen ersten WP angegeben hat und welche für den zweiten. Die maximale und minimale Anzahl der Teilnehmer pro WP sind festgelegt. Die letztere beträgt höchstens 10 und dient dazu, die WPs mit einer kleineren Teilnehmerzahl nicht stattfinden zu lassen. Nach den internen Richtlinien werden auch solche WPs nicht stattfinden, die wenig Teilnehmer haben, für die das WP ihre erste Präferenz darstellt. Allerdings wird es garantiert, dass jeder Student einen Wunschkurs bekommt.

Diese Randbedingungen lassen sich in unser Kursplanungsproblem folgendermaßen integrieren. Um auszudrücken, dass jeder Student keinen Kurs ohne Präferenz bekommen soll (in dem Sinne wäre solcher Kurs für ihn „unzulässig“), würde man in der Präferenztabelle nur Werte $\{1,2,3,4,5,-1\}$ erlauben und alle Kurse ohne Angabe der Präferenz mit „-1“ kennzeichnen. Des Weiteren würden wir in der Situation mit zwei WPs die fünf Präferenzwerte zwischen den WPs differenzieren. Der Grund dafür ist, dass der Student dann nicht entscheiden muss, welchen Kurs er am liebsten hätte, wenn er zwei Kurse gleich präferiert und an beiden teilnehmen möchte. Der Vorschlag wäre, die Studenten aufzuklären, die Werte 1, 2 und 3 als Präferenzen für den ersten Kurs anzugeben, 4 und 5 für den zweiten. Dann würde man jeden Studenten, der zwei Kurse bekommen muss, in unserer Präferenztabelle (siehe z.B. Tabelle 1) doppelt erfassen: als einen Studenten mit Präferenzen 1, 2 und 3 und als einen fiktiven Studenten mit Präferenzen 1 (statt 4) und 2 (statt 5) entsprechend. Damit würde er nach unseren Algorithmen zwei Kurse bekommen – einen pro Tabelleneintrag, siehe Tabelle 7.

Da wir keine Begrenzung bezüglich der minimalen Teilnehmerzahl in unserem Modell haben, würden wir die Algorithmen zunächst bei $n = k$ laufen lassen, um festzustellen, welche Kurse

unterbucht sein werden. Wenn diese Kursanzahl bestimmt ist, starten wir den nächsten Lauf mit entsprechend geringerem k.

Name	Vorname	Cape Town	New York	Rio	Dublin	Paris	Kabul	Saigon
~~Schmidt~~	~~Paul~~	~~1~~	~~2~~	~~3~~	~~4~~	~~5~~	~~-1~~	~~-1~~
Schmidt	Paul1	1	2	3	-1	-1	-1	-1
Schmidt	Paul2	-1	-1	-1	1	2	-1	-1
…	…	…	…	…	…	…	…	…

Tabelle 7. Umgang mit fünf Präferenzen

Zum Zweck des Tests haben wir solche Präferenztabellen mit $V = 1$ generiert, d.h. jeder Student hat keine freie Wahl, die anderen Parameter sind praxisnah:

```
generate.exe Test 100 200 15 15 15 1.0 2 200
```

Man behält die minimale Teilnehmerzahl, z.B. 9, bis nach dem ersten Anwendungslauf, dessen Ergebnis 1 bis 3 unterbuchte Kurse aufweist. Wenn wir jetzt $k = 15 - 2 = 13$ und $c = 16$ (damit jeder einen Platz hat) setzen, findet der Solver in 99 % der Fälle keine optimale Lösung. Wenn die Heuristiken immer noch die Lösung mit einem unterbuchten Kurs berechnen, kann man nachträglich die Studenten manuell in andere Kurse verteilen.

Die Messwerte der Testläufe sind in der Tabelle 8 präsentiert. Der obere Teil entspricht $k = c = 15$, der untere $k = 13$, $c = 16$. Wir sehen hier, dass der Anteil der erfüllten Wünsche verhältnismäßig gesunken ist, das passiert aufgrund der Streichung der Kurse und dadurch weggefallenen Plätze. Trotzdem würden wir die Lösung des DROP-Verfahrens als Basis für die manuellen Änderungen bevorzugen, da er die meiste Wünsche erfüllt.

Heuristik	Praef 1	Praef 2	Praef 3	Praef 0	Praef -1	Rechenzeit (ms)
Solver	0,7875	0,1935	0,019	0	0	541,7
Los2	0,805	0,11	0,03	0	0,055	1,1
Los1	0,74	0,16	0,01	0	0,09	2,7
DROP	0,8345	0,042	0,017	0,01	0,0965	1,2
Solver	0	0	0	0	0	0
Los2	0,745	0,12	0,0235	0	0,1115	1,9
Los1	0,665	0,145	0,035	0	0,155	2
DROP	0,8245	0,0415	0,0145	0	0,1195	0,8

Tabelle 8. Simulation der Wahl von WPs an der HAW (Präferenzverteilung in %)

Wir kommen daher zu dem Schluss, dass man unter diesen Umständen die drei beschriebenen Verfahren auch in anderen Institutionen widerverwenden kann.

7 Analyse der Einflussfaktoren

Im vorigen Kapitel haben wir die drei vorgestellten Verfahren miteinander verglichen und die Zusammenhänge und Besonderheiten aufgedeckt. Jetzt ist es wichtig zu verstehen, inwieweit diese Zusammenhänge von den Randbedingungen beeinflusst werden.

7.1 Kursbeliebtheit

In der Praxis kommt eher selten vor, dass Studenten sich die angebotenen Kurse gleich gerne wünschen. Welches Verfahren berechnet also die bessere Lösung bei gleicher oder, im Gegenteil, stark unterschiedlicher Kursbeliebtheit?

Wenn man die gleichmäßige Kursbeliebtheit simulieren möchte, sollte man einen flachen Abschnitt der Gauß-Funktion nehmen, z.B. mit $\sigma = 2$ und $V = 0{,}1$, siehe Abbildung 5. Wie im Kapitel 5 schon erwähnt, wenn die Kurse fast gleich beliebt sind, unterscheiden sich die Ergebnisse der Heuristiken nicht so stark von Ergebnissen des Solvers.

Wir sind wie üblich vorgegangen und vier Testläufe mit jeweils 100 Dateien durchgeführt. Bei jedem Testlauf galt $m = 200, n = 16, k = 14, c = 17, Q = 20$, nur die Werte σ und V variierten, siehe Tabelle 9. Die Beliebtheit der Kurse war je nach Testlauf immer stärker unterschiedlich, siehe Abbildung 11. Die resultierenden Zahlen sind in der Tabelle 9 zu finden.

Es ist selbstverständlich, dass je höher die Abweichung bei Beliebtheit der Kurse ist, desto niedrigeren Wert der Zielfunktion wir erreichen. Wenn viele Studenten denselben Kurs wählen, deren Kapazität aber begrenzt ist, dann erzeugt dies Unzufriedenheit. Einem fällt darüber hinaus auf, dass bei flacheren Beliebtheitskurve das Losverfahren Nr. 2 dem DROP-Verfahren ähnlich ist und die zweitbesten Lösungen liefert. Bei wachsender Unbeliebtheit bleibt es allerdings zurück und berechnet die schlechtesten Lösungen mit dem Losverfahren Nr. 1. Zusammengefasst: Wenn man davon ausgeht, dass die angebotenen Kurse gleich

beliebt sind, wird man mit allen Algorithmen qualitativ ähnliche Lösungen erhalten. Bei starken Ungleichheit sollte man lieber den Solver oder das DROP-Verfahren bevorzugen.

Heuristik	Praef 1	Praef 2	Praef 3	Praef 0	Praef -1	σ	V	Zielfunktion
Solver	0,8965	0,103	0,0005	0	0	2	0,1	579,2
Los1	0,883	0,101	0,013	0,0025	0,0005			572,7
Los2	0,8955	0,088	0,0155	0,001	0			**575,6**
DROP	0,9135	0,0625	0,0145	0,0075	0,002			**575,6**
Solver	0,8915	0,1055	0,003	0	0	2	1,5	577,7
Los1	0,886	0,0965	0,0155	0,001	0,001			573,1
Los2	0,892	0,0915	0,015	0,0005	0,001			**574,6**
DROP	0,908	0,07	0,0135	0,0045	0,004			**574,7**
Solver	0,9	0,099	0,001	0	0	1,7	2	579,8
Los1	0,897	0,0885	0,011	0,003	0,0005			**575,7**
Los2	0,897	0,089	0,01	0,004	0			**575,8**
DROP	0,92	0,0595	0,0125	0,0035	0,0045			577,4
Solver	0,8485	0,1515	0	0	0	1,5	3	569,7
Los1	0,817	0,131	0,03	0,007	0,015			**545,6**
Los2	0,8265	0,108	0,0345	0,0295	0,0015			**545,7**
DROP	0,8785	0,052	0,023	0,0405	0,006			551,3

Tabelle 9. Präferenzverteilung je nach Beliebtheit (in %)

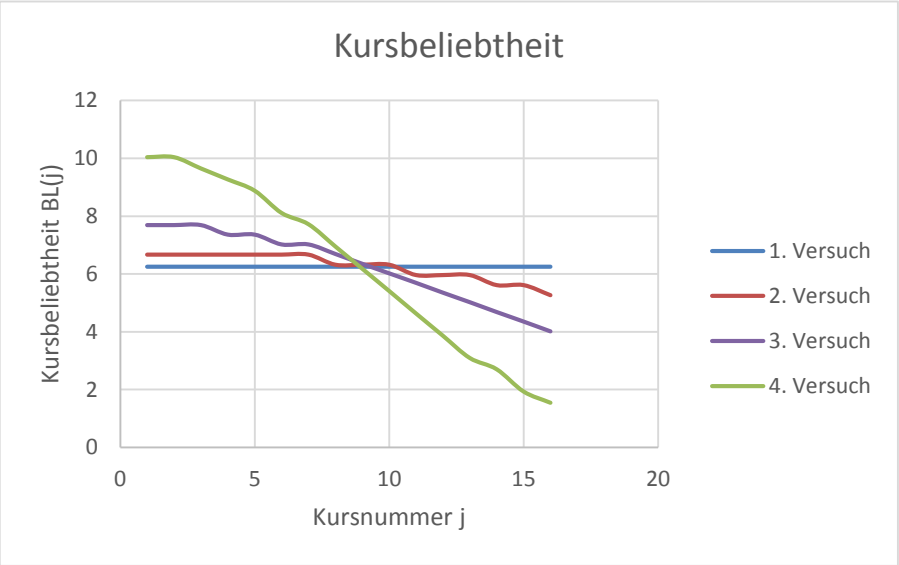

Abbildung 11. Unterschiedliche Kursbeliebtheit

7.2 Freiheitsgrad der Wahl

Welche Auswirkung hat es, wenn ein bestimmter Anteil der Studenten nicht für alle Kurse zugelassen ist? Dieser Anteil wird bei der Datengenerierung mit dem Eingabeparameter $Q \in [1, m]$ beschrieben. Alle Werte $Q \geq m$ verursachen, dass alle Studenten nur positive Präferenzen für höchstens drei Kurse angeben können, aller anderen Kurse werden mit „-1“ gekennzeichnet. Wir haben diverse durchschnittliche Belegungen von Q bei konstanten $m = 200,\ n = 16,\ k = 14,\ c = 17,\ \sigma = 1{,}7,\ V = 2$ untersucht, siehe Tabelle 10.

Heuristik	Praef 1	Praef 2	Praef 3	Praef 0	Praef -1	Rechenzeit (ms)	Q	Zielfunktion
Solver	0,9025	0,096	0,0015	0	0	803,6	1	580,2
Los1	0,8875	0,1015	0,0105	0,0005	0	1,4		575,2
Los2	0,8965	0,0855	0,015	0,003	0	1,4		575,1
DROP	0,92	0,0515	0,0195	0,009	0	1,6		576,5
Solver	0,8845	0,113	0,0025	0	0	779,6	50	576,4
Los1	0,8825	0,096	0,0165	0,003	0,002	1,5		570,8
Los2	0,881	0,096	0,016	0,0045	0,0025	1,6		569,7
DROP	0,909	0,049	0,0195	0,0055	0,017	0,9		565,5
Solver	0,437	0,0615	0,0015	0	0	708,5	100	287,1
Los1	0,874	0,0985	0,01	0,0045	0,013	1,4		563,2
Los2	0,874	0,103	0,012	0,0035	0,0075	1,4		566,5
DROP	0,91	0,0485	0,0135	0,0035	0,0245	1,3		563,2
Solver	0	0	0	0	0	707,4	150	0
Los1	0,8545	0,0995	0,0105	0	0,0355	1,7		547,5
Los2	0,8545	0,0995	0,0105	0	0,0355	1,5		547,5
DROP	0,9115	0,0335	0,005	0	0,05	1,2		551,3
Solver	0	0	0	0	0	862,9	200	0
Los1	0,853	0,0965	0,0065	0	0,044	1,4		542,9
Los2	0,853	0,0965	0,0065	0	0,044	1,5		542,9
DROP	0,9025	0,0395	0,004	0	0,054	1		547,3

Tabelle 10. Präferenzverteilung je nach Freiheitsgrad der Wahl (in %)

Wir stellen fest, dass der Solver schon bei 50 % Studenten ohne freien Wahl ($Q = 100$) nur in der Hälfte aller Fälle eine optimale Lösung findet. Das Constraint (3), das nur einen erlaubten Kurs pro Student zulässt (siehe Kapitel 2.2), ist in solchen Situationen sehr prohibitiv. Qualität der Losung bei Heuristiken sinkt dagegen nicht so drastisch mit dem wachsenden Q, lediglich um 5 - 5,6 %. Das beste Ergebnis bei hohen Q zeigt wieder das DROP-

Verfahren, und beide Losverfahren liefern interessanterweise immer die gleichen Lösungen. Folglich würden wir für solche Situationen, wo die Studenten keine freie Wahl haben bzw. nur positive Präferenzen angeben dürfen (wie an der HAW), am besten das DROP-Verfahren zur Problemlösung anwenden.

8 Zusammenfassung

8.1 Fazit

In dieser Arbeit wurde das an manchen Hochschulen entstehende Kursplanungsproblem analysiert und drei Lösungsansätze miteinander verglichen. Um einen mühsamen manuellen Lösungsprozess zu ersetzen, wurde eine Planungsanwendung implementiert, die allen von der Hochschule gestellten spezifischen Anforderungen entspricht und diese Lösungsansätze beinhaltet. Die Anwendung integriert die auf dem Markt vorhandene fertige Open-Source-Optimierungssoftware (den GLPK-Solver mit Branch-and-Bound-Algorithmus), die Nachimplementierung des an der Hochschule eingesetzten Lösungsverfahrens (Losverfahren) und die speziell für das Problem entworfene Heuristik aus dem Bereich Operations Research (DROP-Verfahren).

Um diese drei Verfahren miteinander vergleichen zu können und jede Konstellation der Eingabeparameter zum Ausdruck bringen, haben wir darüber hinaus einen Datengenerator implementiert. Dieser liefert innerhalb weniger Sekunden eine gewünschte Anzahl der Dateien, die anschließend als Input für unsere Anwendung dienen. Mithilfe des Generators wurde die Auswertung der verschiedenen Einflussfaktoren sowie Eigenschaften der drei Algorithmen vorgenommen.

Die exakten Methoden wie Branch-and-Bound haben exponentiellen Ressourcenaufwand und Sensibilität gegenüber dem geringen Freiheitsgrad der Studentenwahl aufgewiesen. Dennoch berechnen sie immer eine optimale Lösung, falls eine solche existiert.

Im Gegensatz dazu sind die verwendeten Heuristiken blitzschnell und liefern auch dann ein sehr gutes Ergebnis, wenn es keine optimale Lösung gibt. Unter den Heuristiken zeigt das nachimplementierte Losverfahren die schlechtesten Ergebnisse mit ca. 70-80 % Erfüllung der

ersten Wünsche. Dies stellt allerdings eine gewisse Verbesserung dar, da in der Realität oft nur 60 % der höchsten Prioritäten manuell erfüllt werden. Des Weiteren ist das Losverfahren Nr. 2, wonach nur die ersten Präferenzen für die Aussortierung der Kurse berücksichtigt werden, erfolgreicher als sein gleichnamiger Konkurrent, der alle drei Präferenzwerte beachtet. Beste heuristische Lösung, die nach einigen subjektiven Kriterien sogar besser empfunden wird als die optimale Lösung des Solvers, wird vom DROP-Verfahren (ca. 90% der erfüllten ersten Wünsche) erzeugt. Wenn die angebotenen Kurse von den Studenten gleich beliebt sind, liefern alle drei Verfahren ähnliche Werte, wobei dasselbe Verhältnis zwischen den Lösungen besteht.

Deswegen empfiehlt es sich, bei größeren Probleminstanzen oder vielen restriktiven Constraints die DROP-Heuristik anstatt des Solvers einzusetzen, eventuell mit einer nachträglichen manuellen Korrektur oder in Kombination mit den anderen exakten Methoden, die eine gute Initiallösung benötigen.

8.2 Ausblick

Im Kapitel 2.5 beschriebenen Metaheuristiken, wie z.B. Simulierte Abkühlung, Tabu-Suche und Genetische Algorithmen, die für allgemeinere Stundenplanprobleme erfolgreich angewandt wurden, können unter Umständen auch für das Kursplanungsproblem nützlich sein. Allerdings versprechen sie keine optimale Lösung, benötigen meist erheblichen Zeit- und Speicheraufwand, und sämtliche Parameter müssen an das Problem angepasst werden.

Da das Kursplanungsproblem in der Literatur noch nicht ausführlich erforscht ist, hoffen wir, dass diese Metaheuristiken sowie auch andere Optimierungsverfahren zur dessen Lösung in Zukunft genutzt werden.

8.3 Offene Probleme

Die Performanz der heuristischen Algorithmen könnte unserer Meinung nach durch das Nutzen der geeigneten Datenstrukturen noch verbessert werden. Zudem könnte man moderne Datenbanken anstatt einer Excel-Datei zum Einsatz bringen, um Kontrolle über Parameter, Auswertung und Korrektur des Resultats handhabbarer zu gestalten. Trotz unserer Annahme, dass für die Wahl eines Kurses fünf Präferenzwerte (1, 2, 3, 0, -1) eine ausgeglichene Zahl ist, bleibt die Frage, welchen Einfluss eine größere Anzahl der Präferenzwerte auf Zufriedenheit der Studenten hat, ebenfalls offen.

9 Symbolverzeichnis

$m \in M$	m ist eine Element der Menge M
$M_1 \subseteq M_2$	M_1 ist eine Teilmenge von M_2
$M_1 \cap M_2$	Schnittmenge von M_1 und M_2
$\bigcup_{i=1}^{n} M_i$	Vereinigung der Mengen $M_1, \dots, M_n$
$\mathbb{Z}$ bzw. $\mathbb{R}$	Menge aller ganzen bzw. reellen Zahlen
$\mathbb{Z}^+$ bzw. $\mathbb{R}^+$	Menge aller positiven ganzen bzw. reellen Zahlen
$i \in \overline{1,m}$ oder $i \in [1,m]$	i gehört zum abgeschlossenen Intervall von 1 bis m
$\forall$	Allquantor („für jeden“)
$(a_{ij})_{m\times n}$	Eine Matrix der Größe $m \times n$ (m Zeilen und n Spalten), deren Elemente a_{ij} sind: $\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}$
$(b_i)_m$	Ein m-dimensionaler Vektor mit Koordinaten $(b_1, b_2, \dots, b_m)$
c^T bei $c = (c_j)_n$	Ein zu c transponierter Vektor: $\begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix}$
$\sum_{i=1}^{m} x_{ij}$ oder $\sum_i x_{ij}$	$x_{1j} + x_{2j} + \cdots + x_{mj}$
$\sum_{i,j} x_{ij}$	$\sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij}$
$[a], a \in \mathbb{R}$	Abrundungsfunktion, also $\max\{b \in \mathbb{Z} \mid b \leq a\}$

10 Abbildungsverzeichnis

Abbildung 1. Baum der Probleme nach (Domschke, et al., 2011) 16
Abbildung 2. Ablauf des Branch-and-Bound-Verfahrens nach (Domschke, et al., 2011) 18
Abbildung 3. Verlauf des Programms 24
Abbildung 4. GUI 24
Abbildung 5. Gauß-Funktion 31
Abbildung 6. Tatsächliche Kursbeliebtheit im Vergleich mit den Vorgabewerten 33
Abbildung 7. Approximation der Beliebtheitskurve aus der Praxis mithilfe der Gauß-Funktion 37
Abbildung 8. Ausgabe des Solvers 38
Abbildung 9. Laufzeit in Abhängigkeit von m und n (mit Solver) 39
Abbildung 10. Laufzeit in Abhängigkeit von m und n (ohne Solver) 40
Abbildung 11. Unterschiedliche Kursbeliebtheit 44

11 Tabellenverzeichnis

Tabelle 1. Beispiel für eine Präferenztabelle……4
Tabelle 2. Präferenzen vs. Prioritäten……6
Tabelle 3. Beispiel für eine Tabelle mit der resultierenden Kurszuordnung……22
Tabelle 4. Statistik anhand der echten Daten……35
Tabelle 5. Statistik anhand der generierten Daten mit denselben Parametern……36
Tabelle 6. Laufzeit (ms) in Abhängigkeit von m und n……40
Tabelle 7. Umgang mit fünf Präferenzen……42
Tabelle 8. Simulation der Wahl von WPs an der HAW (Präferenzverteilung in %)……42
Tabelle 9. Präferenzverteilung je nach Beliebtheit (in %)……44
Tabelle 10. Präferenzverteilung je nach Freiheitsgrad der Wahl (in %)……45

12 Literaturverzeichnis

TOMLAB Optimization. [Online] [Zitat vom: 1. Oktober 2014.] http://tomopt.com/tomlab/.

The GNU Prolog web site. [Online] [Zitat vom: 1. Oktober 2014.] http://www.gprolog.org/.

CPLEX Optimizer. [Online] IBM. [Zitat vom: 1. Oktober 2014.] http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/index.html.

GLPK (GNU Linear Programming Kit). [Online] GNU Project - Free Software Foundation (FSF). [Zitat vom: 1. Oktober 2014.] http://www.gnu.org/software/glpk/.

GLPK/GMPL (MathProg). [Online] Wikibooks. [Zitat vom: 1. Oktober 2014.] http://en.wikibooks.org/wiki/GLPK/GMPL_%28MathProg%29.

Aubin, Jean und Ferland, Jacques A. 1989. A large scale timetabling problem. *Computers & Operations Research.* 1989, Bd. 16, 1, S. 67-77.

Badri, Masood A., et al. 1998. A multi-objective course scheduling model: Combining faculty preferences for courses and times. *Computers & Operations Research.* 1998, Bd. 25, 4, S. 303-316.

Bloomfield, Stefan D. und McSharry, Michael M. 1979. PREFERENTIAL COURSE SCHEDULING. *Interfaces.* 1979, Bd. 9, 4, S. 24-31.

Burke, E.K., Kendall, G. und Soubeiga, E. 2003. A Tabu-Search Hyperheuristic for Timetabling and Rostering. *Journal of Heuristics.* 2003, Bd. 9, 6, S. 451-470.

Burke, Edmund, Kingston, Jeffrey und Pepper, Paul. 1998. A standard data format for timetabling instances. [Hrsg.] Edmund Burke und Michael Carter. *Practice and Theory of Automated Timetabling II.* 1998, Bd. 1408, S. 213-222.

Busam, Vincent A. 1967. An algorithm for class scheduling with section preference. *Commun. ACM.* 1967, Bd. 10, 9, S. 567-569.

Cambazard, Hadrien, et al. 2005. Interactively Solving School Timetabling Problems Using Extensions of Constraint Programming. [Hrsg.] Edmund Burke und Michael Trick. *Practice and Theory of Automated Timetabling V.* 2005, Bd. 3616, S. 190-207.

Carter, Michael. 2001. A Comprehensive Course Timetabling and Student Scheduling System at the University of Waterloo. [Hrsg.] Edmund Burke und Wilhelm Erben. *Practice and Theory of Automated Timetabling III.* 2001, Bd. 2079, S. 64-82.

Carter, Michael und Laporte, Gilbert. 1998. Recent developments in practical course timetabling. [Hrsg.] Edmund Burke und Michael Carter. *Practice and Theory of Automated Timetabling II.* 1998, Bd. 1408, S. 3-19.

Domschke, Wolfgang und Drexl, Andreas. 2011. *Einführung in Operations Research.* 8. Auflage. Berlin Heidelberg : Springer-Lehrbuch, 2011.

Dowsland, Kathryn. 1998. Off-the-peg or made-to-measure? timetabling and scheduling with SA and TS. [Hrsg.] Edmund Burke und Michael Carter. *Practice and Theory of Automated Timetabling II.* 1998, Bd. 1408, S. 37-52.

Elmohamed, M., Coddington, Paul und Fox, Geoffrey. 1998. A comparison of annealing techniques for academic course scheduling. [Hrsg.] Edmund Burke und Michael Carter. *Practice and Theory of Automated Timetabling II.* 1998, Bd. 1408, S. 92-112.

Even, S., Itai, A. und Shamir, A. 1976. On the Complexity of Timetable and Multicommodity Flow Problems. *SIAM Journal on Computing.* 1976, Bd. 5, 4, S. 691-703.

Foulds, L.R. und Johnson, D.G. 2000. SlotManager: a microcomputer-based decision support system for university timetabling. *Decision Support Systems.* 2000, Bd. 27, 4, S. 367-381.

Hartmann, Sönke. 2014. Selecting and Assigning Elective Courses at Universities (Working Paper). 2014.

Hopcroft, J.E., Motwani, Rajeev und Ullman, Jeffrey D. 2011. *Einführung in die Automatentheorie, formale Sprachen und Berechenbarkeit.* 3. Auflage. München : Pearson Studium, 2011.

Johnson, David. 1993. A Database Approach to Course Timetabling. *The Journal of the Operational Research Society.* 1993, Bd. 44, 5, S. 425-433.

Junginger, Werner. 1986. Timetabling in Germany - A Survey. *Interfaces.* 1986, Bd. 16, 4, S. 66-74.

Karp, Richard M. 1972. Reducibility among combinatorial problems. [Hrsg.] R. Miller und J. Thatcher. *Complexity of Computer Computations.* New York : Plenum Press, 1972, S. 85-103.

Laporte, Gilbert und Desroches, Sylvain. 1986. The problem of assigning students to course sections in a large engineering school. *Computers & Operations Research.* 1986, Bd. 13, 4, S. 387-394.

Lewis, Rhydian. 2008. A survey of metaheuristic-based techniques for University Timetabling problems. *OR Spectrum.* 2008, Bd. 30, 1, S. 167-190.

Macon, N. und Walker, E. E. 1966. A Monte Carlo algorithm for assigning students to classes. *Commun. ACM.* 1966, Bd. 9, 5, S. 339-340.

McCollum, Barry. 2007. A Perspective on Bridging the Gap Between Theory and Practice in University Timetabling. [Hrsg.] Edmund Burke und Hana Rudová. *Practice and Theory of Automated Timetabling VI.* 2007, Bd. 3867, S. 3-23.

Ostermann, R. und de Werra, D. 1982. Some experiments with a timetabling system. *OR Spectrum.* 1982, Bd. 3, 4, S. 199-204.

Perzina, Radomír. 2007. Solving the University Timetabling Problem with Optimized Enrollment of Students by a Self-adaptive Genetic Algorithm. [Hrsg.] Edmund Burke und Hana Rudová. *Practice and Theory of Automated Timetabling VI.* 2007, Bd. 3867, S. 248-263.

Rossi-Doria, Olivia, et al. 2003. A Comparison of the Performance of Different Metaheuristics on the Timetabling Problem. [Hrsg.] Edmund Burke und Patrick De Causmaecker. *Practice and Theory of Automated Timetabling IV.* 2003, Bd. 2740, S. 329-351.

Rudová, Hana und Murray, Keith. 2003. University Course Timetabling with Soft Constraints. [Hrsg.] Edmund Burke und Patrick De Causmaecker. *Practice and Theory of Automated Timetabling IV.* 2003, Bd. 2740, S. 310-328.

Sampson, Scott E., Freeland, James R. und Weiss, Elliott N. 1995. Class Scheduling to Maximize Participant Satisfaction. *Interfaces.* 1995, Bd. 25, 3, S. 30-41.

Schaerf, A. 1999. A Survey of Automated Timetabling. *Artificial Intelligence Review.* 1999, Bd. 13, 2, S. 87-127.

Tripathy, Arabinda. 1980. A Lagrangean Relaxation Approach to Course Timetabling. *The Journal of the Operational Research Society.* 1980, Bd. 31, 7, S. 599-603.

—. 1992. Computerised decision aid for timetabling — a case analysis. *Discrete Applied Mathematics.* 1992, Bd. 35, 3, S. 313 - 323.

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, den _____ _____