Inhaltsverzeichnis

1 Einleitung

Die Erstellung von Stundenplänen an Hochschulen ist ein vielfältiges Problem. Neben dem Lösen von Raum- und Zeitkonflikten, soll den Bedürfnissen der Beteiligten möglichst entsprochen werden. Dies sind vor allem die Lehrenden und die Studierenden.

In dieser Arbeit soll auf praxisnahe Anforderungen an den Stundenplan der Technischen Hochschule Lübeck eingegangen werden. Neben der Erfüllung der grundlegenden Anforderungen an einen Stundenplan, wird gezeigt wie mit einer Bibliothek zur Lösung von Optimierungsproblemen, auch aufwendige Konstellationen eines Stundenplans umgesetzt und gleichzeitig Optimierungen aus Sicht der Beteiligten gefunden werden können. Daraus wird ein Programm entwickelt, das aus den Inhalten eines Stundenplans – genannt Stundenplandaten – einen Stundenplan erstellt, der den ermittelten Anforderungen entspricht.

Die Implementierung soll mithilfe der Bibliothek *OR-Tools* von Google und in der Programmiersprache Python erfolgen. Eine Ausgabe des erstellten Stundenplans soll in Form einer Excel Datei geschehen.

Der letzte Stand des erstellten Programms, vor der Abgabe dieser Arbeit, befindet sich in dem GitHub Repository `https://github.com/Jon2050/TimeTabling`.

1.1 Hintergrund

Die Suche nach Stundenplänen ist ein lange bekanntes Problem der Informatik. Die erstellten Stundenpläne sollen nicht nur grundlegende Anforderungen, wie das Ausschließen von Zeitkonflikten, erfüllen, sondern auch aus Sicht der Beteiligten, möglichst optimal gestaltet sein. Das verwandte, aus der theoretischen Informatik stammende Problem, zu entscheiden, ob es zu den Inhalten eines Stundenplans, einen Stundenplan gibt, das heißt eine Anordnung und Raumzuteilung aller Veranstaltungen, sodass es keine Zeit- beziehungsweise Raumkonflikte gibt, ist NP-Vollständig. [Lov10]

Mit Zeit- und Raumkonflikten, ist gemeint, dass jeder Lehrende und jeder Studierende, nie mehrere Veranstaltungen zum gleichen Zeitpunkt zugeteilt bekommt. Gleiches gilt für die Räume. Es darf immer nur eine Veranstaltung zu einem Zeitpunkt in einem Raum stattfinden. (Der „Raumkonflikt“ ist daher ein Zeitkonflikt)

Zur Lösungssuche soll die Bibliothek *OR-Tools* von Google verwendet werden. Sie ermöglicht es ein Problem auf deklarative Art und Weise durch Angabe eines Modells aus Variablen und Constraints anzugeben und lösen zu lassen. Einige Constraints können so angegeben werden, dass deren Erfüllung optional ist. [Gooa]

Auch gängige Softwarelösungen setzen auf Stundenplansuche mittels Constraint Programming und Optimierung. Das Problem wird in die Teile, Finden einer validen Lösung („feasibility“, Erfüllung harter Anforderungen) und Suche nach optimalen Lösungen („optimization“, Erfüllung optionaler/weicher Anforderungen), aufgeteilt. [Uni]

1.2 Motivation

Wie sich beim Fachbereich *Elektrotechnik und Informatik* der TH-Lübeck zeigt, werden Stundenpläne an Hochschulen oftmals noch von Hand erstellt. Dies ist aufwendig, ermöglicht es aber, viele individuelle Wünsche der Beteiligten zu berücksichtigen. Ein Stundenplanmodell einer Softwarelösung kann sehr spezielle Wünsche eventuell nicht abbilden. Außerdem benötigen einige Programme zur Stundenplanerstellung viel Zeit. Es ist nicht immer ersichtlich wann und ob überhaupt ein Stundenplan gefunden wird. Dennoch ermöglichen solche Softwarelösungen die Erstellung von Stundenplänen mit hoher Veranstaltungsdichte, die bei manueller Erstellung unter Umständen nicht gefunden würden. Außerdem nimmt der Aspekt der Optimierung einen höheren Stellenwert ein, da eine deutlich höhere Anzahl an Lösungen evaluiert werden kann.

In dieser Arbeit sollen zum einen die speziellen Anforderungen des genannten Fachbereichs an den Stundenplan ermittelt werden. Da diese möglichst vollständig in ein Programm zur Stundenplanerstellung umgesetzt werden sollen, wird sich gut erkennen lassen, wie gut sich die *OR-Tools* Bibliothek tatsächlich zur Erstellung von Hochschulstundenplänen eignet.

1.3 Ziele

Das erste Ziel wird die Ermittlung der Anforderungen und Besonderheiten des Stundenplans des Fachbereichs *Elektrotechnik und Informatik* sein. Diese sollen formal in ein Modell gefasst werden, sodass sie von einer Software umgesetzt werden können. Dabei wird es entscheidend sein, zwischen der Möglichkeit, spezielle Anforderungen umsetzen und alle Eventualitäten des Stundenplans abdecken zu können und der dadurch steigenden Komplexität des Stundenplanmodells, abzuwägen.

Diese Anforderungen werden in drei Gruppen eingeteilt. Grundlegendes Ausschließen von Zeitkonflikten und Umsetzung direkter Vorgaben, in welcher Form Veranstaltungen stattfinden sollen, sind die Bestandteile der ersten Gruppe. Die Anforderungen der zweiten Gruppe sollen garantieren, dass der Stundenplan für die beteiligten Lehrenden und Studierenden nicht zu unbequem ist. Die Anforderungen aus der dritten Gruppe sind optional und ermöglichen es, den Stundenplan für die Beteiligten zu optimieren. Dies kann zum Beispiel geschehen, indem Lücken für Studierende, oder späte Veranstaltungen am Tag vermieden werden.

Es wird sich zeigen, dass an einigen Stellen Kompromisse eingegangen werden müssen, um Auswirkungen der Kombination mehrerer Anforderungen und dadurch entstehende Spezialfälle, bewältigen zu können.

Hauptaufgabe wird es sein, alle Anforderungen zu implementieren. Dabei müssen diese so in das Modell der Constraint Programming Bibliothek übersetzt werden, dass dieses immer noch in akzeptabler Zeit lösbar ist. Ein erstrebenswertes Ziel ist eine Implementation, die für größere Stundenpläne, innerhalb einiger Minuten eine optimierte Lösung liefert.

Ein Bestandteil ist Angabe eines Datenmodells, welches die grundlegenden Informationen zu Veranstaltungen, Lehrenden, Studierenden und verfügbaren Räumen enthält.

Es muss zudem alle Angaben enthalten, die für die Umsetzung aller harten und weichen Anforderungen benötigt werden. Dieses Datenmodell wird entscheiden, wie genau die Angaben für den Stundenplan sein und welche Konstellationen im Stundenplan vorgegeben werden können. Dies können zum Beispiel Angaben sein, dass bestimmte Veranstaltungen zu bestimmten Zeiten, zeitgleich, oder nacheinander mit anderen Veranstaltungen stattfinden sollen.

Außerdem sollen gefundene Stundenpläne, für Menschen gut lesbar, ausgegeben werden können. Es soll möglich sein, dass Studierende und Lehrende dieser Ausgabe ihre individuellen Stundenpläne entnehmen können.

1.4 Gliederung

Zunächst wird in Abschnitt 2 ein zugrundeliegendes Stundenplanmodell erläutert. Dies beinhaltet die Einführung von Begriffen, die für diesen Stundenplan gelten und von Stundenplanobjekten. Letzteres sind die Teile, aus den der Stundenplan besteht. Außerdem wird auf Besonderheiten und spezifische Elemente des Stundenplans des Fachbereichs *Elektrotechnik und Informatik* der Technischen Hochschule Lübeck eingegangen. Aus diesen und allgemeinen Überlegungen, wie ein Stundenplan für Hochschulen optimiert werden kann, werden in Abschnitt 2.3 harte Anforderungen an den Stundenplan spezifiziert. In Abschnitt 2.4 folgt die Definition optionaler Anforderungen, welche die Voraussetzung für eine Optimierung des Stundenplans darstellen. Anschließend wird gezeigt, wie alle Bestandteile des Stundenplans durch ein Datenbankmodell modelliert werden. Dies ist notwendig, da bei einer automatisierten Stundenplanerstellung, eine Datenbasis vorhanden sein muss. Diese Daten enthalten zum Beispiel die Information welche Veranstaltungen, mit welchen Teilnehmern, stattfinden sollen.

Der Abschnitt 3 beschäftigt sich mit der eigentlichen Implementierung des Programms zur automatischen Erstellung von Stundenplänen. Die Umsetzung jeder Anforderung wird gezeigt. Es wird des Weiteren gezeigt, wie das erstellte Programm verwendet wird.

Im Abschlussabschnitt wird kurz auf die Leistungsfähigkeit der gefundenen Programmlösung erlangte Erkenntnisse eingegangen.

2 Stundenplan und Datenmodell

Dieser Abschnitt beschäftigt sich mit dem verwendetem Modell eines Stundenplans. Das Modell besteht dabei aus Vorgaben, wie der Stundenplan auszusehen hat und bestimmt damit den Rahmen, in dem das Programm einen Stundenplan finden soll.

Es wird erläutert welche Voraussetzungen für den Stundenplan umgesetzt werden müssen. Diese Voraussetzungen werden als Bedingungen, Constraints genannt, formuliert. Einige sind dabei ganz grundsätzlicher Natur und sollten im Grunde für jeden Stundenplan gelten. Diese grundsätzlichen Constraints werden im Folgenden auch Basisanforderungen genannt. Andere Constraints sind spezieller und dienen dazu die Güte des zu erstellenden Stundenplans zu erhöhen. Dabei werden gewissermaßen Wünsche der beteiligten Individuen umgesetzt, die von dem Stundenplan betroffen sind. Dies sind in erster Linie die Lehrenden und die Studierenden. Auf diese Weise soll ein Stundenplan gefunden werden, der für die Beteiligten möglichst angenehm ist. Diese Constraints werden im Folgenden erweiterte Anforderungen genannt. Die dritte Stufe bilden die optionalen Anforderungen. Sie haben den gleichen Zweck wie die erweiterten Anforderungen, müssen aber anders als diese nicht zwingend eingehalten werden. Dies ist notwendig, da nicht für alle Stundenplandaten ein optimaler Stundenplan möglich ist. Die optionalen Anforderungen sind gewichtet und bilden die Stellschraube, mit der ein optimierter Stundenplan gefunden werden soll. Diese Anforderungen und ihre Gewichtungen sind das Maß, mit dem die Güte des Stundenplans im Kontext der Suche gemessen wird.

Zunächst werden einige grundlegende Begriffe eingeführt. Dann folgt eine Erklärung der einzelnen Anforderungen, die bei der Erstellung des Stundenplans umzusetzen sind. Dazu wird zunächst auf spezielle Anforderungen an den Stundenplan der TH-Lübeck eingegangen, aus denen sich viele der Constraints ergeben. Aufgeteilt in harte und optionale Constraints, folgt dann die Erklärung jeder einzelnen Anforderung. Die Benennung ebendieser erfolgt in Englisch. Zum einen da deutsche Namen länger und umständlicher wären, zum anderen da die Namen im Programmcode Wiederverwendung finden können, welcher vollständig in englischer Sprache abgefasst ist.

Als letzter Teil des Abschnitts 2, erfolgt eine Beschreibung des Datenmodells, in dem die Stundenplandaten enthalten sind und dessen grundlegende Struktur im eigentlichen Programm Verwendung findet.

2.1 Verwendete Begriffe und Stundenplanobjekte

Dieser Abschnitt enthält Erläuterungen der wichtigsten verwendeten Begriffe und Objekte.

Stundenplandaten Die Stundenplandaten bezeichnen die Datengrundlage, aus welcher der Stundenplan erstellt werden soll. Sie enthalten alle Kurse die stattfinden sollen, sowie alle beteiligten Lehrenden, Studierende – eingeteilt in Semestergruppen – und Räume. Außerdem enthalten die Stundenplandaten Vorgaben für viele erweiterte und einige optionale Anforderungen. Beispiele für diese Daten sind, dass ein Kurs A zweimal in

der Woche für eine bestimmte Semestergruppe und mit einem bestimmten Lehrendem stattfinden soll, oder dass ein bestimmter Lehrender maximal drei 90-minütige Blöcke mit Vorlesungen an einem Tag halten soll.

Stundenplanmodell Modellierung und Vorgabe wie der Stundenplan auszusehen hat und welche Konstellationen vorkommen dürfen oder sollen. Eine Konstellation beschreibt wie, beziehungsweise wann, Kurse stattfinden sollen (gegebenenfalls relativ zu anderen Kursen). Das Stundenplanmodell wird durch die Stundenplandaten vorgegeben. Eine hohe Flexibilität des Stundenplanmodells erlaubt es zum einen, möglichst genaue Vorgaben zu machen in welchem Rahmen bestimmte Kurse stattfinden sollen, während auf der anderen Seite genügend Raum zur Lösungsfindung und Optimierung bleiben soll. Durch die eingeschränkte Komplexität des Modells sind jedoch möglicherweise einige ganz konkrete Konstellationen oder deren Vorgabe nicht immer möglich.

Stundenplanobjekte Als Objekte des Stundenplans werden die Elemente der Stundenplandaten bezeichnet, die eigenständig und mit bestimmten Eigenschaften im Datenmodell und als Python Klassen im eigentlichen Programm enthalten sind. Dies sind die Lehrenden (`Teacher`), Semestergruppen (`SemesterGroup`), Kurse (`Course`), Unterrichtseinheiten (`Lesson`) und Räume (`Room`).

Lehrende sind Personen die Unterrichtseinheiten halten, zum Beispiel Professoren, Wissenschaftliche Hilfskräfte oder Laboringenieure. Es wird jedoch nicht direkt zwischen diesen Kategorien unterschieden. Besonderheiten für Professoren wie der Studientag, können daher theoretisch für jede Lehrkraft angegeben werden. Die Begriffe Lehrende, Lehrkräfte und Teacher (im Kontext von Implementierungsdetails in Bezug auf die entsprechende Python Klasse `Teacher`), werden synonym verwendet.

Semestergruppe Eine Semestergruppe steht repräsentativ für Studierende, die an den Veranstaltungen teilnehmen. Sie fasst dabei Studenten zusammen, die üblicherweise zusammen Veranstaltungen besuchen. Dies sind für jeden Studiengang die einzelnen Semester. Dies stellt eine Vereinfachung dar, da nicht jeder Studierende einzeln berücksichtigt werden muss. Im Falle von Vertiefungsrichtungen kann es notwendig sein einzelne Semester eines Studiengangs noch einmal in die Vertiefungen aufzuteilen. Im Zusammenhang mit den Programmdetails wird für Semestergruppen auch der Begriff SemesterGroup, wie die verwendete Python Klasse `SemesterGroup`, verwendet.

Kurs Kurse sind die einzelnen Fächer im Stundenplan. Sie sind unabhängig von einzelnen Studiengängen oder Semestern definiert, sodass auch mehrere Semestergruppen an einem Kurs teilnehmen können. Außerdem gehören zu jedem Kurs Räume, in denen diese stattfinden können und Lessons, welche die tatsächlichen Veranstaltungen eines Kurses darstellen. Die Entsprechung im Python Programm ist die Klasse `Course`.

Lesson Eine Lesson ist eine Unterrichtseinheit eines Kurses und damit der Begriff für eine einzelne Veranstaltung, die innerhalb des Zeitraums, den der Stundenplan beschreibt,

stattfindet. Sie ist das Stundenplanobjekt, das letztlich im Stundenplan eingetragen wird. Zu einem Kurs können mehrere Lessons gehören. Eine Semestergruppe, die an einem Kurs teilnimmt, nimmt automatisch an allen Lessons dieses Kurses teil. Eine Lesson kann mehr als einen Timeslot belegen, also länger als 90 Minuten dauern. Diese werden dann als mehrstündige Lessons bezeichnet. Jeder Lesson sind Lehrende zugeordnet, die daher für den Zeitraum, zu dem die Lesson stattfindet, belegt sind. `Lesson` ist auch der Name der entsprechenden Python Klasse.

Lessons werden von Lehrenden unterrichtet. Dabei können einer Lesson auch mehrere Lehrende zugeordnet sein, die dann alle zum Zeitpunkt der Lesson belegt sind.

Beziehungen der Stundenplanobjekte Die einzelnen Attribute der Stundenplanobjekte und deren genaue Beziehungen zueinander sind dem Abschnitt 2.6 Umsetzung als Datenbankmodell zu entnehmen.

Vorlesungen (Lectures) Für einige Constraints muss zwischen Vorlesungen und anderen Lessons unterschieden werden. Lessons können daher die Eigenschaft *`lecture`* haben, die sie als Vorlesung kennzeichnet.

Timeslot Die einzelnen Zeitabschnitte, in die der Stundenplan eingeteilt ist werden als Timeslots bezeichnet. Im Falle des Stundenplans der TH-Lübeck sind dies an jedem Wochentag 6 90-minütige Abschnitte. In jedem gefundenem Stundenplan muss jede Lesson einem oder mehreren aufeinanderfolgenden (im Falle von mehrstündigen Lessons) Timeslots zugeordnet sein. Der Begriff Zeitslot wird synonym verwendet. Die zugehörige Python Klasse ist gleichnamig `Timeslot`.

Block Als Block werden mehrere Lessons bezeichnet, die direkt nacheinander stattfinden. Zum Beispiel bilden zwei aufeinanderfolgende Lessons einen Block oder in dem Fall auch Doppelblock. Die Länge der Lessons ist dabei unerheblich, der Block bezieht sich nur auf die Lessons an sich, nicht auf die Timeslots die diese belegen.

Studierendentag Ein Studierendentag bezeichnet einen Wochentag mit Veranstaltungen aus Sicht einer Semestergruppe oder eines Studierenden. Ein Studierendentag beginnt mit der ersten Veranstaltung des Studierenden und endet mit der letzten Veranstaltung des Studierenden. Die Studierendentage können daher kürzer oder länger und unterschiedlich voll, das heißt mit unterschiedlich vielen belegten Timeslots bestückt sein. Ein Wochentag ohne Veranstaltungen für den Studierenden wird als leerer Studierendentag bezeichnet. Der Studierendentag eines Studierenden ist nicht zwangsläufig identisch mit dem Studierendentag der Semestergruppe zu der er gehört, da es Veranstaltungen gibt, an denen nicht alle Studierenden einer Semestergruppe teilnehmen. Dies sind zum Beispiel einige Praktika.

Stundenplanvorlage Neben der persönlichen Befragung, diente auch ein bereits fertiger Stundenplan des Fachbereichs EI der TH-Lübeck eines vergangenen Semesters als

Vorlage und zur Ermittlung von Konstellationen die durch das Stundenplanmodell abgedeckt sein sollen. Wenn von einer Stundenplanvorlage gesprochen wird, ist stets dieser Stundenplan gemeint.

Kurstyp Die einzelnen Kurse eines Stundenplans können in verschiedene Typen eingeordnet werden. In der Stundenplanvorlage sind dies unter anderem Vorlesungen, Praktika, Übungen und Projekte. Der Kurstyp ist jedoch lediglich für eine übersichtlichere Ausgabe relevant und findet bei der Stundenplansuche keine Beachtung.

Constraint Mit Constraints sind zum einen die Anforderungen und Bedingungen gemeint, die für den Stundenplan gelten müssen. Jedes Constraint hat einen (englischen) Namen und ist in den Abschnitten 2.3 Harte Anforderungen und 2.4 Optionale Anforderungen aufgeführt. Die Anforderungen an den Stundenplan sind dabei in Basisanforderungen, erweiterte Anforderungen und optionale Anforderungen aufgeteilt. Die Basisanforderungen gelten im Grunde für jeden denkbaren Stundenplan, während die erweiterten und die optionalen Anforderungen entweder bestimmte Szenarien, das heißt Konstellationen die immer oder unter bestimmten Umständen gelten sollen, des Hochschulstundenplans der TH-Lübeck umsetzen oder der Erhöhung der Stundenplangüte dienen. Constraint, Anforderung, Bedingung und Vorgabe werden synonym verwendet.

Constraint werden jedoch auch einzelne Bestimmungen genannt, die dem OR-Tools Modell hinzugefügt werden und die sich dann immer auf Variablen des OR-Tools Modells beziehen. Diese OR-Tools Constraints dienen dazu die Constraints des Stundenplans umzusetzen.

Stundenplangüte, Optimierung Ziel der automatisierten Stundenplanerstellung ist es nicht nur überhaupt irgendeinen möglichen Stundenplan zu finden, sondern einen Stundenplan, der für die Beteiligten (Lehrende und Studierende) möglichst angenehm ist. Viele der erweiterten Anforderungen dienen dazu den Stundenplan so zu formen. Das Maß, das beschreibt, wie gut diese Anforderungen umgesetzt sind und (in der Theorie) wie bequem der Stundenplan für die Beteiligten ist, wird im Folgendem als Stundenplangüte bezeichnet. Neben den erweiterten Anforderungen wird die Stundenplangüte auch durch die optionalen Anforderungen erhöht. Bei diesen wird ihr sogar durch die Optimierungsfunktion ein Wert zugewiesen.

Fülle des Stundenplans Nicht für alle denkbaren Stundenplandaten ist auch ein Stundenplan möglich. Unter Umständen kann nicht für alle Veranstaltungen ein Zeitpunkt gefunden werden, zu dem ein geeigneter Raum, der oder die Lehrenden und die Semestergruppe(n) noch nicht mit anderen Veranstaltungen belegt sind. Je mehr Veranstaltungen einzelner der zuvor genannten Stundenplanobjekte zugeordnet sind, desto weniger mögliche Stundenpläne gibt es insgesamt, bis zu dem Punkt zu dem es keinen mehr gibt. Mit Stundenplanfülle ist daher gemeint, wie stark belegt oder ausgelastet einzelne Stundenplanobjekte (Räume, Lehrende, Semestergruppen) sind. Eine hohe Stundenplanfülle kann auch durch eine hohe Vernetzung vieler dieser Objekte entstehen. Sie steigt jedoch nicht automatisch mit der Anzahl an Stundenplanobjekten beziehungsweise

Veranstaltungen und Kursen. Da die Anzahl möglicher Lösungen mit zunehmender Stundenplanfülle sinkt, dauert auch die Lösungssuche im Durchschnitt länger. Außerdem wird ab einem gewissen Punkt die Stundenplangüte (hier nur die der optionalen Anforderungen) niedriger wenn die Stundenplanfülle steigt, da eine optimale Lösung, das heißt die Erfüllung aller optionaler Constraints, nicht mehr möglich ist.

2.2 Besonderheiten des Stundenplans TH-Lübeck, EI

Hier sollen einige Anforderungen und deren Hintergrund vorab erläutert werden. Diese beziehen sich auf Besonderheiten eines Hochschulstundenplans oder auf besondere Anforderungen des Stundenplans der TH-Lübeck, genauer, des Fachbereichs Elektrotechnik und Informatik.

Studientage Jeder Professor hat pro Woche einen sogenannten Studientag, an dem für ihn keine Veranstaltungen eingeplant werden sollen. Zu Beginn des Semesters geben alle Professoren eine Erst- und Zweitwahl für den Wochentag an, an dem ihr Studientag im kommenden Semester liegen soll. Diese Wünsche der Professoren sind Teil des Stundenplanmodells. Bei Lehrkräften, denen kein Studientag zusteht, erfolgt auch keine Angabe der bevorzugten Studientage. In den erweiterten Anforderungen gibt es das StudyDayConstraint (2.3.5). Dieses garantiert, dass an einem der beiden gewählten Tage keine Veranstaltungen für die jeweilige Lehrkraft stattfinden. Die Erstwahl wird an dieser Stelle noch nicht bevorzugt. Dies geschieht als optionale Anforderung durch das PreferFirstStudyDayChoiceConstraint (2.4.1). Es ist ferner erlaubt, das Erst- und Zweitwahl denselben Wochentag benennen. In diesem Fall ist dies zwingend der Studientag.

Aufteilung der Semestergruppen (Praktika, Übungen) Im Allgemeinen wird angenommen, dass alle Studierenden desselben Studiengangs und desselben Semesters an denselben Veranstaltungen teilnehmen. Daher sind diese Studenten in Semestergruppen zusammengefasst. Der Fall, dass zum Beispiel einzelne Studierende im dritten Semester einen Kurs aus dem ersten Semester nachholen wollen, wird bei dieser Stundenplanerstellung nicht berücksichtigt. Es gibt jedoch Ausnahmen die Vorgesehen sind und bei denen Studierende derselben Semestergruppe zeitgleich an verschiedenen Lessons teilnehmen können sollen. Dies können Lessons desselben oder von unterschiedlichen Kursen sein. Das sind zum einen Übungen und Praktika, ggf. Laborveranstaltungen oder Wahlpflichtkurse. Bei Übungen und Praktika werden die Studierenden einer Semestergruppe in Untergruppen eingeteilt und nehmen an den Veranstaltungen zu unterschiedlichen Zeiten und/oder unterschiedlichen Orten teil. Wenn mehrere Lehrende und mehrere Räume für die Veranstaltung zur Verfügung stehen, können auch solche Lessons desselben Kurses gleichzeitig stattfinden. Das zugehörige Constraint heißt PartSemesterGroupConstraint (2.3.4). Die Ermöglichung dieser Ausnahme führt zu einem flexibleren Stundenplan und verbessert die Chance für gegebene Stundenplandaten einen Stundenplan zu finden, beziehungsweise die Güte zu erhöhen, da die Fülle des Stundenplans durch die zusätzlichen Optionen sinkt. Es soll jedoch möglichst sichergestellt werden, dass alle Studierenden einer Semestergruppe alle Praktika und Übungen dieser Art auch tatsächlich besuchen

können. Dazu mehr im Abschnitt über die Implementierung für Semestergruppen, SemesterGroupTimeConstraints und PartSemesterGroupTimeConstraint (3.4.4).

Im Datenmodell besitzen Lessons ein Flag, welches angibt, ob die gesamte oder nur ein Teil der Semestergruppe oder der Semestergruppen, an der Lesson teilnimmt bzw. teilnehmen. Im Folgenden wird daher häufig zwischen Lessons unterschieden, bei denen dieses Flag gesetzt beziehungsweise nicht gesetzt ist. Es wird dann von Lessons an denen ganze Semestergruppen oder Lessons an denen Teilgruppen teilnehmen gesprochen. (Auch kurz: Teilgruppen-Lessons)

Veranstaltungen nur vormittags Bei der manuellen Stundenplanerstellung im Fachbereich EI der TH-Lübeck, wird bei Vorlesungen meist darauf geachtet, dass diese möglichst vormittags stattfinden. Diese Möglichkeit soll auch die automatisierte Erstellung bieten. Umgesetzt wird dies durch das OnlyForenoonConstraint (2.3.8). Um möglichst flexibel zu bleiben und die Lösungen nicht zu sehr einzuschränken (was der Fall wäre, wenn alle Vorlesungen automatisch nur vormittags stattfinden dürften), kann diese Vorgabe für jeden Kurs durch ein Flag aktiviert werden. Es sollte jedoch nicht bei allen Vorlesungen verwendet werden. Bei einer üblichen Stundenplanfülle ergäben sich dann oftmals, kaum erfüllbare Stundenplanmodelle. Dieses Constraint könnte alternativ auch als optionale Anforderung implementiert werden. In diesem Fall könnten mehr, im Prinzip auch alle Vorlesungen mit der Anforderung belegt werden. Ein Vorteil der Umsetzung als hartes Constraint ist jedoch, dass für einzelne Kurse bestimmt werden kann, dass diese definitiv am Vormittag stattfinden. Dies ermöglicht so genauere und gleichzeitig flexiblere Vorgaben an den Stundenplan.

Gleichzeitige Veranstaltungen Obwohl der Stundenplan des Fachbereichs zur Zeit der Anforderungsanalyse einen einwöchigen Stundenplan verwendet, kommen Szenarien vor, in denen sich Veranstaltungen von Woche zu Woche abwechseln, oder sich von der ersten zur zweiten Semesterhälfte unterscheiden. In diesen Fällen sind in den Stundenplänen unter Umständen für einen Lehrenden, eine Semestergruppe oder einen Raum mehrere Lessons zur selben Zeit eingetragen (2.3.10 LessonsAtSameTimeConstraint). Da dies gegen die Basisanforderungen verstoßen würde, mussten diese angepasst werden, sodass es Ausnahmen für diese Lessons gibt. Des Weiteren müssen auch alle Constraints angepasst werden, die in irgendeiner Form die Anzahl Lessons pro Tag zählen. Die Möglichkeit diese Lessons zeitgleich in den Stundenplan eintragen zu können hat daher eine Auswirkung auf viele andere Constraints.

Sich ausschließende Kurse pro Tag Bei einigen Veranstaltungen, die in Laboren stattfinden und üblicherweise von Laboringenieuren gehalten werden, sollen diese möglichst keine verschiedenen Veranstaltungen dieser Art an einem Tag haben, da die Vorbereitung bei diesen oft aufwendiger ist. Um das im Stundenplan umzusetzen, gibt es bei jedem Kurs ein Flag, das gesetzt werden kann. Für einen Lehrenden können dann keine Veranstaltungen mehrerer Kurse am selben Tag stattfinden, bei denen dieses Flag gesetzt ist. Wohl aber mehrere Veranstaltungen desselben Kurses. Die Menge aller Kurse wird so in zwei Gruppen eingeteilt. Solche die mit allen anderen Kursen zusammen an

einem Tag stattfinden können und solche, die nicht zusammen an einem Tag stattfinden, wenn derselbe Lehrende für deren Veranstaltungen eingetragen ist. Dies wird durch das OneCoursePerDayTeacherConstraint (2.3.13) beschrieben.

2.3 Harte Anforderungen

In diesem Abschnitt werden die harten Anforderungen beschrieben. Sie zeichnen sich dadurch aus, dass sie in jedem Fall eingehalten werden müssen. Es wird niemals ein Stundenplan gefunden werden, bei dem auch nur eine dieser Anforderungen nicht erfüllt ist. Sie bilden den Hauptteil der automatisierten Stundenplansuche, da durch sie die Vorgaben festgelegt sind, wie der Stundenplan auszusehen hat. Das heißt welche Merkmale vorkommen sollen und welche nicht.

Bei den harten Anforderungen wird zwischen den Basisanforderungen und den erweiterten Anforderungen unterschieden.

Als **Basisanforderungen** werden Bedingungen bezeichnet, die üblicherweise für jeden Stundenplan gelten. Dabei ist es unerheblich, ob es sich um einen Stundenplan für Schulen oder Hochschulen handelt. Die Basisanforderungen sind im einzelnen durch die Constraints RoomTimeConstraint, TeacherTimeConstraint und SemesterGroupTimeConstraint gegeben. Diese drei Constraints sind sich jeweils sehr ähnlich. Sie verhindern jeweils das gleichzeitige stattfinden von Lessons mit bestimmten Eigenschaften. Die Basisanforderungen bedürfen keinerlei zusätzlicher Informationen aus den Stundenplandaten und gelten aus sich selbst heraus. Auf ein Constraint, welches angibt, dass alle Veranstaltungen der Stundenplandaten auch tatsächlich in den Stundenplan eingetragen werden, wird verzichtet. Dies ist selbstverständlich und ist bei der gewählten Implementierung auch implizit gegeben.

Erweiterte Anforderungen bilden zusätzliche Bedingungen. Viele von Ihnen würden bei einer manuellen Stundenplanerstellung gar nicht explizit angegeben werden. Einige dienen dazu spezielle Szenarien eines Hochschulstundenplans oder ganz spezielle Anforderungen des Stundenplans des Fachbereichs EI der TH-Lübeck umzusetzen. Während andere dazu dienen den Stundenplan nicht zu unangenehm für die Beteiligten werden zu lassen und damit die Stundenplangüte zu erhöhen. Es kommt vor, dass erweiterte Anforderungen Ausnahmen für andere (auch Basis-) Anforderungen erfordern. Die meisten erweiterten Anforderungen basieren auf zusätzlichen Angaben, die in den Stundenplandaten enthalten und an einzelne Stundenplanobjekte angehängt sind.

Zwar müssen die harten Anforderungen immer erfüllt werden, doch da sich viele Anforderungen auf Daten und Flags in den Stundenplandaten beziehen, können sie auf diese Weise gewissermaßen für einzelne Stundenplanobjekte auf die sie sich beziehen, ein- oder ausgeschaltet sein. So dienen viele Constraints dazu, spezielle Konstellationen und Vorgaben des Stundenplanmodells umzusetzen.

In Tabelle 1 ist eine Übersicht über alle harten Constraints zu sehen. In der Spalte Inhalt ist jeweils der Verweis auf den Abschnitt aufgeführt, in dem das Constraint er-

klärt wird. Die Spalte Implementierung verweist auf den jeweiligen Abschnitt in dem die Implementierung des Constraints in Python und mit der OR-Tools Bibliothek gezeigt wird. Diese sind alle in Abschnitt 3.4 HardConstraints, Implementierung zu finden.

Constraint Name	Inhalt	Implementierung
RoomTimeConstraint	2.3.1	3.4.5
TeacherTimeConstraint	2.3.2	3.4.3
SemesterGroupTimeConstraint	2.3.3	3.4.4
PartSemesterGroupConstraint	2.3.4	3.4.4
StudyDayConstraint	2.3.5	3.4.6
NotAvailableRoomTimeConstraint	2.3.6	3.4.7
NotAvailableTeacherTimeConstraint	2.3.7	3.4.1
OnlyForenoonConstraint	2.3.8	3.4.1
LessonTakePlaceConstraint	2.3.9	3.4.1
LessonsAtSameTimeConstraint	2.3.10	3.4.1
CourseAsBlockConstraint	2.3.11	3.4.8
BlockInSameRoomConstraint	2.3.12	3.4.8
OneCoursePerDayTeacherConstraint	2.3.13	3.4.15
MaxLessonsPerDayTeacherConstraint	2.3.14	3.4.10
MaxLecturesPerDayTeacherConstraint	2.3.15	3.4.14
MaxLecturesAsBlockTeacherConstraint	2.3.16	3.4.13
MaxLessonsPerDaySemesterGroupConstraint	2.3.17	3.4.11
MaxLessonsPerDayCourseConstraint	2.3.18	3.4.12
ConsecutiveLessonsConstraint	2.3.19	3.4.9

Tabelle 1: Übersicht der harten Anforderungen

2.3.1 RoomTimeConstraint

Zu jedem Zeitpunkt kann in einem Raum immer nur einzelne Veranstaltung eines einzelnen Kurses stattfinden. Diese erste Basisanforderung stellt demnach für jeden Timeslot und jeden Raum sicher, dass wenn dort bereits eine Lesson eingetragen ist, keine zweite eingetragen wird. Für alle Paare von Lessons aus der Menge aller stattfindenden Lessons muss gelten, dass entweder der Zeitpunkt zu dem sie stattfinden oder der Ort (Raum) unterschiedlich ist. Dies ist in Formel 1 in prädikatenlogischer Form dargestellt. Das Prädikat $IstLesson(l)$ gilt, wenn es sich bei l um eine Lesson des Stundenplans handelt. Das Prädikat $RaumUngleich(l_1, l_2)$ gibt für zwei Lessons an, dass diese in unterschiedlichen Räumen, $ZeitUngleich(l_1, l_2)$, dass sie zu unterschiedlichen Zeitpunkten stattfinden. Für mehrstündigen Lessons bedeutet letzteres, dass es keinerlei zeitliche Überschneidung gibt.

$$\forall l_1 \forall l_2 : IstLesson(l_1) \land IstLesson(l_2) \land l_1 \neq l_2 \Longrightarrow$$
$$RaumUngleich(l_1, l_2) \lor ZeitUngleich(l_1, l_2)$$

Formel 1: RoomTimeConstraint

Wie im Absatz über Gleichzeitige Veranstaltungen in Abschnitt 2.2 erwähnt, sollen Ausnahmen für dieses Constraint möglich sein. Dazu sollen im Datenmodell Mengen von Lessons angegeben werden können. Alle Lessons einer Menge sollen dann zeitgleich stattfinden (anfangen), auch wenn sie im selben Raum stattfinden. Genaueres zu dieser Ausnahme wird in Abschnitt 2.3.10 LessonsAtSameTimeConstraint erläutert. Für dasRoomTimeConstraint bedeutet diese Ausnahme, dass es jeweils für alle Paare von Lessons aus diesen Mengen nicht gelten soll.

Diese Ausnahme ist in Formel 2 enthalten. Das Prädikat $InZeitgleichMenge(l_1, l_2)$ bedeutet, dass sich die beiden Lessons l_1 und l_2 in derselben Menge von Lessons befinden, die durch Vorgabe des LessonsAtSameTimeConstraints zeitgleich beginnen sollen.

$$\forall l_1 \forall l_2 : IstLesson(l_1) \land IstLesson(l_2) \land l_1 \neq l_2 \land$$
$$\neg InZeitgleichMenge(l_1, l_2) \Longrightarrow$$
$$RaumUngleich(l_1, l_2) \lor ZeitUngleich(l_1, l_2)$$

Formel 2: RoomTimeConstraint, erweitert

Zur Implementierung des RoomTimeConstraints, siehe Abschnitt 3.4.5 RoomTimeConstraints.

2.3.2 TeacherTimeConstraint

Das TeacherTimeConstraint stellt sicher, dass niemals mehrere Lessons zeitgleich stattfinden die von denselben Lehrenden gehalten werden. Dies ist eine grundlegende Anforderung an jeden Stundenplan, da Lehrende für gewöhnlich nicht mehrere Veranstaltungen gleichzeitig unterrichten können. Anders ausgedrückt bedeutet dies, dass für alle Paare von Lessons entweder die Lehrenden oder die Timeslots der beiden Lessons unterschiedlich sein müssen. (Formel 3)

$$\forall l_1 \forall l_2 : IstLesson(l_1) \land IstLesson(l_2) \land l_1 \neq l_2 \Longrightarrow$$
$$LehrenderUngleich(l_1, l_2) \lor ZeitUngleich(l_1, l_2)$$

Formel 3: TeacherTimeConstraint

Das Prädikat $IstLesson(l)$ gilt, wenn es sich bei l um eine Lesson des Stundenplans handelt. $LehrenderUngleich(l_1, l_2)$ gibt für jeweils zwei Lessons an, dass es keinen Lehrenden gibt, der beide unterrichtet. Das Prädikat $ZeitUngleich(l_1, l_2)$ gilt für zwei Lessons deren Zeiträume, zu denen sie stattfinden, sich nicht überschneiden.

Auch bei dieser Basisanforderung gilt die Ausnahme für alle Lessons die mittels des LessonsAtSameTimeConstraint verknüpft sind. Bei diesen Lessons dürfen die Lehrenden gleich sein, auch wenn sie gleichzeitig stattfinden.

Die Implementierung dieses Constraints ist in Abschnitt 3.4.3 TeacherTimeConstraint erläutert.

2.3.3 SemesterGroupTimeConstraint

Das SemesterGroupTimeConstraint verhindert das gleichzeitige Stattfinden von Lessons an denen die gleiche Semestergruppe teilnimmt. Im Gegensatz zum RoomTimeConstraint und zum TeacherTimeConstraint gibt es bei diesem Constraint zusätzliche Ausnahmen. Lessons an denen nur ein Teil einer Semestergruppe teilnimmt, dürfen unter Umständen gleichzeitig stattfinden. Dies wird im Abschnitt 2.3.4 PartSemesterGroupConstraint genauer beschrieben. Durch das Datenmodell muss ersichtlich sein, ob eine Lesson für gesamte oder nur für Teile von Semestergruppen veranstaltet wird. Alle Lessons an denen die gleiche Semestergruppe als gesamte Gruppe teilnimmt, dürfen nicht gleichzeitig stattfinden. Außerdem darf eine Lesson an der die gesamte Gruppe teilnimmt, nicht gleichzeitig mit einer Lesson stattfinden, an der eine Teilgruppe derselben Semestergruppe teilnimmt. Die Ausnahme, dass Lessons gleichzeitig stattfinden dürfen (und müssen) wenn sie mit dem LessonsAtSameTimeConstraint verknüpft sind, bleibt bestehen. Abgesehen von letztgenannter Ausnahme, gilt die prädikatenlogische Formel 4.

$$\forall l_1 \forall l_2 : IstLesson(l_1) \wedge IstLesson(l_2) \wedge l_1 \neq l_2 \ \wedge GleicheSemestergruppe(l_1, l_2) \wedge (\neg IstTeilgruppenLesson(l_1) \vee \neg IstTeilgruppenLesson(l_2)) \Longrightarrow ZeitUngleich(l_1, l_2)$$

Formel 4: SemesterGroupTimeConstraint

Die Prädikate der Formel 4:

- $IstLesson(l)$: Gibt an, dass es sich um eine Lesson handelt.
- $GleicheSemestergruppe(l_1, l_2)$: Es gibt mindestens eine Semestergruppe, die an Lesson l_1 und Lesson l_2 teilnimmt.
- $IstTeilgruppenLesson(l)$: Gibt an, dass es sich um eine Lesson handelt an der nur Teile von Semestergruppen teilnehmen.
- $ZeitUngleich(l_1, l_2)$: Es gibt keinen Timeslot, zu dem beide Lessons l_1 und l_2 stattfinden.

Die Formel 4 besagt, dass alle Paare von Lessons, an denen dieselben Semestergruppen teilnehmen und mindestens eine der Lessons eine Lesson ist an denen gesamte Gruppen teilnehmen, nicht zeitgleich stattfinden dürfen.

Die Implementierungsdetails sind in Abschnitt 3.4.4 SemesterGroupTimeConstraints und PartSemesterGroupTimeConstraint zu finden.

2.3.4 PartSemesterGroupConstraint

Dieses Constraint bezieht sich auf Lessons an denen nur ein Teil einer Semestergruppe teilnimmt. Es gilt immer jeweils für Lessons an denen die gleiche Semestergruppe teilnimmt. Einige Lessons sollen so markiert sein können, dass an ihnen nur Teile von Semestergruppen teilnehmen. Es wird angenommen, dass ein einzelner Studierender immer nur an einer Teilgruppen-Lesson eines Kurses teilnehmen muss[1]. Zwei Teilgruppen-Lessons sollen gleichzeitig stattfinden können, auch wenn an ihnen die gleiche Semestergruppe teilnimmt. Es wird außerdem angenommen, dass es von Lessons mit dieser Markierung immer mindestens zwei für einen Kurs gibt. Da es bisher kein Constraint gibt, welches das gleichzeitige Stattfinden von Teilgruppen-Lessons verhindert, muss dies nicht ausdrücklich erlaubt werden. Das SemesterGroupTimeConstraint bezieht sich ja nie auf mehrere Lessons die alle Teilgruppen-Lessons sind. Da in einigen Fällen aber nicht alle Studenten an allen Kursen teilnehmen könnten, sollen dabei zwei Bedingungen für das gleichzeitige Stattfinden der Teilgruppen-Lessons gelten. Gemeint sind immer Lessons an denen gleiche Semestergruppen teilnehmen:

- Wenn Lessons verschiedener Kurse durch dieses Constraint zeitgleich stattfinden, soll aus jedem Kurs zu diesem Zeitpunkt nur eine Lesson stattfinden.
- Ist eine Teilgruppen Lesson mehrstündig, darf keine Lesson eines anderen Kurses zeitgleich mit dieser stattfinden.

Der erste Fall verhindert zum Beispiel Konstellationen, bei denen alle Lessons eines Kurses gleichzeitig stattfinden und noch mindestens eine Lesson eines anderen Kurses. Dabei könnten einige Studenten einen der beiden Kurse nicht belegen.

Bei mehrstündigen Lessons eines Kurses wäre es möglich, dass während der Dauer dieser Lesson alle Lessons eines anderen Kurses stattfinden. Auch dann könnten nicht alle Studenten der Semestergruppe Lessons beider Kurse besuchen. Da die Lessons des anderen Kurses dabei selber nicht zeitgleich stattfinden müssen, widerspricht dies nicht der ersten Bedingung. Die zweite Bedingung schließt genau diese Konstellationen aus.

Die erste Bedingung wird erst relevant, wenn es sich um mehr als zwei gleichzeitige Lessons handelt. Anders ausgedrückt, müssen also, wenn mehr als zwei Teilgruppen-Lessons zeitgleich stattfinden, diese zum selben Kurs gehören. Diese Bedingung ist in Formel 5 gezeigt.

[1] Das heißt eine Konstellation, dass ein Kurs zum Beispiel 4 Lessons hat und jeweils die Hälfte der Semestergruppe an zwei dieser Lesson teilnimmt, ist im Modell nicht vorgesehen. Es kann jedoch trotzdem sichergestellt werden, dass dies dann immer möglich ist, indem explizit angegeben wird, wann die Lessons stattfinden, oder dass sie nacheinander stattfinden sollen, da so verhindert wird, dass alle Lessons zeitgleich stattfinden. (AvailableLessonTimeConstraint (2.3.9) und ConsecutiveLessonsConstraint (2.3.19))

$$\forall l_1 \forall l_2 \forall l_3 :$$
$$IstTeilgruppenLesson(l_1) \land IstTeilgruppenLesson(l_2) \land IstTeilgruppenLesson(l_3) \land$$
$$l_1 \neq l_2 \land l_2 \neq l_3 \land l_1 \neq l_3 \land$$
$$Zeitgleich(l_1, l_2, l_3) \land GleicheSemestergruppe(l_1, l_2, l_3) \implies$$
$$GleicherKurs(l_1, l_2, l_3)$$

Formel 5: PartSemesterGroupConstraint 1

Auch die zweite Bedingung kann andersherum ausgedrückt werden. Eine mehrstündige Teilgruppen-Lesson darf nur zeitgleich mit einer Teilgruppen-Lesson desselben Kurses stattfinden:

$$\forall l_1 \forall l_2 : IstTeilgruppenLesson(l_1) \land IstTeilgruppenLesson(l_2) \land l_1 \neq l_2 \land Zeitgleich(l_1, l_2) \land$$
$$(Mehrst\ddot{u}ndig(l_1) \lor Mehrst\ddot{u}ndig(l_2)) \land GleicheSemestergruppe(l_1, l_2) \implies$$
$$GleicherKurs(l_1, l_2)$$

Formel 6: PartSemesterGroupConstraint 2

Die Prädikate der Formeln 5 und 6:

- $IstTeilgruppenLesson(l)$: Gibt an, dass es sich um eine Lesson handelt an der nur Teile von Semestergruppen teilnehmen.
- $Zeitgleich(l_1, ...)$: Das Prädikat ist erfüllt, sobald es mindestens einen Timeslot gibt zudem alle Lessons des Prädikats stattfinden.
- $GleicheSemestergruppe(l_1, l_2)$: Es gibt mindestens eine Semestergruppe, die an Lesson l_1 und Lesson l_2 teilnimmt.
- $GleicherKurs(l_1, ...)$: Ist das Prädikat für Lessons die alle zu demselben Kurs gehören.
- $Mehrst\ddot{u}ndig(l)$: l ist eine Lesson, die mehr als einen Timeslot belegt.

Die Implementierungsdetails sind zusammen mit der Implementierung des SemesterGroupTimeConstraints in Abschnitt 3.4.4 SemesterGroupTimeConstraints und PartSemesterGroupTimeConstraint zu finden.

2.3.5 StudyDayConstraint

Jedem Professor steht ein Tag in der Woche zu, an dem für ihn keine Veranstaltungen stattfinden. Dabei sollen jeweils zwei Wochentage gewählt werden, an denen dieser Studientag stattfinden soll. Eine Erstwahl und eine Zweitwahl. Es soll erlaubt sein, dass die Erst- und Zweitwahl der gleiche Tag sind. Das StudyDayConstraint stellt sicher, dass an einem dieser Tage keine Lessons stattfinden, denen der Lehrende zugeteilt ist. Die

Bevorzugung der Erstwahl geschieht als eigenes optionales Constraint PreferFirstStudyDayChoiceConstraint (2.4.1).

Formel 7 zeigt dies in formaler Form: Entweder dürfen alle Lessons die der Lehrende unterrichtet nicht am Erstwahl- oder nicht am Zweitwahltag stattfinden.

$$\forall t : IstLehrender(t) \land HatStudientag(t) \implies$$
$$(\forall l : IstLesson(l) \land Unterrichtet(t, l) \implies \neg FindetAnErstwahlStatt(l, t)) \lor$$
$$(\forall l : IstLesson(l) \land Unterrichtet(t, l) \implies \neg FindetAnZweitwahlStatt(l, t))$$

Formel 7: StudyDayConstraint

Die Prädikate der Formel 7:

- $IstLehrender(t)$: Gibt an, dass es sich um einen Lehrenden handelt.
- $HatStudientag(t)$: Gilt für alle Lehrenden denen ein Studientag zusteht.
- $Unterrichtet(t, l)$: Gibt an, dass der Lehrende t der Lesson l zugeteilt ist.
- $FindetAnErstwahlStatt(l, t)$: Das Prädikat ist wahr für Lessons l die am Erstwahl-Wochentag des Lehrenden t stattfinden.
- $FindetAnZweitwahlStatt(l, t)$: Entsprechend zu $FindetAnErstwahlStatt(l, t)$ für die Zweitwahl des Studientags.

Die Implementierungsdetails sind in Abschnitt 3.4.6 StudyDayConstraints zu finden.

2.3.6 NotAvailableRoomTimeConstraint

Für die Stundenplanerstellung soll es möglich sein, für einzelne Räume Zeiten anzugeben, zu denen diese aus beliebigen Gründen nicht für die Veranstaltungen des Stundenplans zur Verfügung stehen. Denkbar ist zum Beispiel, dass die Räume von anderen Fachbereichen belegt sind. Diese Zeiträume sollen im Datenmodell für den jeweiligen Raum als Liste von Timeslots angegeben werden. Soll zum Beispiel ein Raum eines anderen Fachbereichs oder fachbereichsübergreifend genutzt werden, kann auf diese Weise dessen Nutzungszeit auf einzelne Timeslots eingeschränkt werden zu denen der Fachbereich des Stundenplans den Raum nutzen darf. Diese Liste eines Raumes darf natürlich auch leer sein.

Formel 8 zeigt, dass eine Lesson, die in einem Raum r stattfindet, nicht zu Zeitslots stattfinden darf, zu denen dieser Raum nicht verfügbar ist.

$$\forall l \forall r \forall \tau :$$
$$IstLesson(l) \wedge IstRaum(r) \wedge IstTimeslot(\tau) \wedge$$
$$FindetImRaumStatt(l, r) \wedge NichtVerfügbar(\tau, r) \Longrightarrow$$
$$\neg FindetStattUm(l, \tau)$$

Formel 8: NotAvailableRoomTimeConstraint

Die Prädikate der Formel 8:

- $IstLesson(l)$: Gibt an, dass es sich bei l um eine Lesson handelt.
- $IstRaum(r)$: Gibt an, dass es sich bei r um einen Raum handelt.
- $IstTimeslot(\tau)$: Gibt an, dass es sich bei τ um einen Timeslot handelt.
- $FindetImRaumStatt(l, r)$: Bedeutet, dass die Lesson l im Raum r veranstaltet wird.
- $NichtVerfügbar(\tau, r)$: Gibt an, dass der Timeslot τ in der Liste der nicht verfügbaren Timeslots des Raumes r enthalten ist.
- $FindetStattUm(l, \tau)$: Die Lesson l findet zum Timeslot τ statt. Bei mehrstündigen Lessons bedeutet dies, dass einer der belegten Timeslots dem angegebenem Timeslot entspricht.

Die Implementierungsdetails sind in Abschnitt 3.4.7 RoomNotAvailableConstraints zu finden.

2.3.7 NotAvailableTeacherTimeConstraint

Wie bei Räumen, soll es bei Lehrenden möglich sein, Zeiten anzugeben zu denen diese nicht für Veranstaltungen des Stundenplans zur Verfügung stehen. Dies kann zum Beispiel notwendig sein, weil die Lehrenden an Sitzungen teilnehmen, Sprechstunden haben oder aus privaten Gründen solche Zeiten angeben. Das Datenmodell soll für jeden Lehrenden eine Liste der Timeslots bereitstellen zu denen keine Lessons mit dem jeweiligen Lehrenden stattfinden dürfen. Dieser Umstand ist in Formel 9 aufgezeigt.

$$\forall l \forall t \forall \tau :$$
$$IstLesson(l) \wedge IstLehrender(t) \wedge IstTimeslot(\tau) \wedge$$
$$Lehrt(t, l) \wedge NichtVerfügbar(t, \tau) \Longrightarrow$$
$$\neg FindetStattUm(l, \tau)$$

Formel 9: NotAvailableTeacherTimeConstraint

Die Prädikate der Formel 9:

- $IstLesson(l)$: Gibt an, dass es sich bei l um eine Lesson handelt.
- $IstLehrender(t)$: Gibt an, dass es sich bei t um einen Lehrenden handelt.
- $IstTimeslot(\tau)$: Gibt an, dass es sich bei τ um einen Timeslot handelt.
- $Lehrt(t, l)$: Der Lehrende l ist für die Lesson l als Lehrender eingetragen.
- $NichtVerfügbar(\tau, t)$: Gibt an, dass der Timeslot τ in der Liste der nicht verfügbaren Timeslots des Lehrenden t enthalten ist.
- $FindetStattUm(l, \tau)$: Die Lesson l findet zum Timeslot τ statt. Bei mehrstündigen Lessons bedeutet dies, dass einer der belegten Timeslots dem angegebenem Timeslot entspricht.

Die Implementierungsdetails sind in Abschnitt 3.4.1 Zeit- und Raumvariablen enthalten.

2.3.8 OnlyForenoonConstraint

Wie im Abschnitt 2.2 Besonderheiten des Stundenplans TH-Lübeck, EI, unter Absatz Veranstaltungen nur vormittags, dargelegt, sollen einzelne Veranstaltungen nur vormittags stattfinden. Zunächst muss geklärt sein, welche Timeslots zum Vormittag zählen. Bei dem Stundenplan der TH-Lübeck sind dies jeweils die ersten drei eines jeden Wochentages. Da das gesamte Programm zur Stundenplanerstellung möglichst unabhängig von den Stunden und Unterrichtszeiten einer einzelnen Hochschule sein soll, soll es möglich sein die Vormittags-Timeslots an einer geeigneten Stelle in einer Python Datei anzugeben. Genaue Angaben hierzu im Abschnitt 3.3.1 Timeslot.

Ob eine Veranstaltung nur vormittags stattfinden darf, soll jeweils für einen Kurs angegeben werden. Ist dies der Fall, sollen alle Lessons dieses Kurses nur zu den Timeslots stattfinden, die als Vormittags-Timeslots markiert sind. Es wird kein Stundenplan gefunden werden können, wenn solch ein Kurs eine Lesson beinhaltet, die mehr Timeslots umfasst, als die Anzahl der Vormittags-Timeslots pro Tag.

Die Formel 10 zeigt, dass eine Lesson eines so markierten Kurses, nicht zu einem Zeitslot stattfinden darf, der nicht an einem Vormittag liegt.

$$\begin{gathered}\forall l \forall \tau : IstLesson(l) \land IstTimeslot(\tau) \land \\ NurVormittags(l) \land \neg Vormittags(\tau) \Longrightarrow \\ \neg FindetStattUm(l, \tau)\end{gathered}$$

Formel 10: OnlyForenoonConstraint

Die Prädikate der Formel 10:

- $IstLesson(l)$: Gibt an, dass es sich bei l um eine Lesson handelt.
- $IstTimeslot(\tau)$: Gibt an, dass es sich bei τ um einen Timeslot handelt.

- $NurVormittags(l)$: Für den Kurs der Lesson l ist angegeben, dass Veranstaltungen dieses Kurses nur vormittags stattzufinden haben.
- $Vormittags(\tau)$: Bei τ handelt es sich um einen Timeslot an einem Vormittag.
- $FindetStattUm(l, \tau)$: Die Lesson l findet zum Timeslot τ statt. Bei mehrstündigen Lessons bedeutet dies, dass einer der belegten Timeslots dem angegebenem Timeslot entspricht.

Die Implementierungsdetails sind i Abschnitt 3.4.1 Zeit- und Raumvariablen enthalten.

2.3.9 AvailableLessonTimeConstraint

Eine allgemeinere Form des OnlyForenoonConstraints ist die direkte Angabe von Timeslots zu denen eine Lesson stattfinden darf. Das OnlyForenoonConstraint ist gewissermaßen eine Komfortfunktion, da es auch durch dieses Constraint umsetzbar wäre. So ist es jedoch mit einem Flag an und abschaltbar.

Im Datenmodell soll für jede Lesson eine Liste von Timeslots angegeben werden, zu denen die jeweilige Lesson stattfinden darf. Die Angabe folgt quasi umgekehrt zur Angabe bei den Constraints NotAvailableRoomTimeConstraint und NotAvailableTeacherTimeConstraint, bei denen nicht verfügbare Timeslots angegeben werden. Dies hat den einfachen Grund, dass angenommen wird, dass auf diese Weise jeweils möglichst wenig Timeslots angegeben werden müssen. Ziel ist aber immer die Menge der Timeslots in erlaubte und verbotene Timeslots aufzuteilen.

Ist die angegebene Liste leer, gibt es für den Zeitpunkt der Lesson keine Einschränkungen. Dies ist äquivalent zur Angabe aller Timeslots. Die Liste darf auch einen einzelnen Timeslot enthalten. Ist die Lesson mehrstündig, muss die Liste entsprechend viele direkt aufeinanderfolgende Timeslots enthalten, wenn eine Einschränkung der Veranstaltungszeit erwünscht ist.

Dieses Constraint ermöglicht zum Beispiel das Umsetzen von Konstellationen im Stundenplan die durch keine anderen Constraints ermöglicht werden, weil sie bisher nicht vorgesehen sind. Es bietet die Möglichkeit zur konkreteren Angabe, wann Veranstaltungen stattfinden sollen. Ein möglicher Anwendungsfall wäre, dass ein Lehrender einen genauen Zeitpunkt wünscht, zudem eine seiner Veranstaltungen stattfindet. Stehen alle angegebenen Zeitpunkte im Konflikt mit anderen Constraints, wird jedoch kein Stundenplan gefunden werden können. Die vorhandenen Möglichkeiten, wann eine Lesson stattfinden kann, können daher durch dieses Constraint nur eingeschränkt, nicht erweitert werden. Dies ist der Fall, da alle anderen Constraints weiterhin gelten und für das AvailableLessonTimeConstraint keine Ausnahmen gemacht werden.

Den Umstand, dass die Lessons nur zu den angegebenen Zeitslots stattfinden dürfen, zeigt die Formel 11.

$$\forall l \forall \tau :$$
$$IstLesson(l) \land IstTimeslot(\tau) \land \neg Verfügbarer Timeslot(\tau, l) \implies$$
$$\neg FindetStattUm(l, \tau)$$

Formel 11: AvailableLessonTimeConstraint

Die Prädikate der Formel 11:

- $IstLesson(l)$: Gibt an, dass es sich bei l um eine Lesson handelt.
- $IstTimeslot(\tau)$: Gibt an, dass es sich bei τ um einen Timeslot handelt.
- $Verfügbarer Timeslot(\tau, l)$: Das Prädikat gibt an, dass τ ein Timeslot ist, zu dem die Lesson l stattfinden darf. Dies ist der Fall, wenn er in der Liste der verfügbaren Timeslots der Lesson eingetragen ist, oder diese Liste leer ist.
- $FindetStattUm(l, \tau)$: Die Lesson l findet zum Timeslot τ statt. Bei mehrstündigen Lessons bedeutet dies, dass einer der belegten Timeslots dem angegebenem Timeslot entspricht.

Die Implementierung ist erläutert im Abschnitt 3.4.1 Zeit- und Raumvariablen.

2.3.10 LessonsAtSameTimeConstraint

Obwohl der Stundenplan einwöchig ist, soll es möglich sein, Veranstaltungen zu planen, die im wöchentlichen Wechsel zur selben Zeit stattfinden, oder zwei Veranstaltungen zeitgleich zu planen, wovon die eine nur in der ersten Hälfte und die andere nur in der zweiten Hälfte des Semesters stattfindet. Dies soll ermöglicht werden, auch wenn dann Lessons zeitgleich im Stundenplan eingetragen werden, obwohl sie zum Beispiel vom selben Lehrenden gehalten werden. Siehe Absatz Gleichzeitige Veranstaltungen im Abschnitt über die Besonderheiten des Stundenplans TH-Lübeck, EI (2.2).

Von diesen Gründen abgesehen, kann das Constraint auch dazu verwendet werden für beliebige andere Lessons anzugeben, dass diese zeitgleich stattfinden müssen. Zum Beispiel zwei Lessons eines Praktika-Kurses die sowieso zeitgleich stattfinden könnten, dies ohne Verwendung dieses Constraints jedoch nicht zwingend wäre.

Konkret bedeutet dieses Constraint, dass Lessons angegeben werden können, die in jedem Fall zeitgleich stattfinden sollen, wobei Constraints, die dies normalerweise verhindern würden, ignoriert werden. Wie die Angabe genau erfolgt ist im Absatz Zusätzliche Beziehungen im Abschnitt 2.6.1 Datenbankmodell angegeben. Alle Lessons die mit dem Constraint belegt sind bilden eine Äquivalenzrelation. Die Lessons dürfen unterschiedlich lang sein. Zur Vereinfachung sollen unterschiedlich lange Lessons immer zeitgleich beginnen. Dies ermöglicht eine einfachere Implementierung, da es ausreicht den Beginn der Lessons zu vergleichen. Dies macht den Stundenplan etwas weniger flexibel und schließt

mögliche Lösungen aus. Beispielsweise, dass eine zweistündige Lesson einen Timeslot früher beginnt, als eine einstündige Lesson die zeitgleich stattfinden soll. Die Auswirkung auf die Lösungsmenge ist jedoch sehr gering.

Die prädikatenlogische Formel 12 zeigt, dass Lessons die durch dieses Constraint verknüpft sind, zum selben Timeslot beginnen müssen.

$$\forall l_1 \forall l_2 \exists \tau :$$
$$IstLesson(l_1) \wedge IstLesson(l_2) \wedge IstTimeslot(\tau) \wedge SameTimeConstraint(l_1, l_2) \Longrightarrow$$
$$BeginntUm(l_1, \tau) \wedge BeginntUm(l_2, \tau)$$

Formel 12: LessonsAtSameTimeConstraint

Die Prädikate der Formel 12:

- $IstLesson(l)$: Gibt an, dass es sich bei l um eine Lesson handelt.
- $IstTimeslot(\tau)$: Gibt an, dass es sich bei τ um einen Timeslot handelt.
- $SameTimeConstraint(l_1, l_2)$: Gibt an, dass die beiden Lessons l_1 und l_2 durch das LessonsAtSameTimeConstraint miteinander verknüpft sind.
- $BeginntUm(l, \tau)$: Bedeutet, dass die Lesson l zum Timeslot τ beginnt.

Die Constraints, die bei Angabe des LessonsAtSameTimeConstraints für die betreffende Menge von Lessons zu ignorieren sind:

- RoomTimeConstraint
- TeacherTimeConstraint
- SemesterGroupTimeConstraint

Beim MaxLessonsPerDayCourseConstraint (2.3.18) werden Lessons mit dem LessonsAtSameTimeConstraint ignoriert, da sonst verhindert würde, dass mehrere Lessons desselben Kurses zeitgleich stattfinden. Dies bedeutet jedoch, dass Lessons mit dem LessonsAtSameTimeConstraint am selben Tag (nicht zur selben Zeit) wie eine weitere Lesson desselben Kurses stattfinden kann, welche selber nicht mit dem Constraint belegt ist. Mehr dazu im Abschnitt 2.3.18 MaxLessonsPerDayCourseConstraint.

Außerdem sollen die, auf diese Weise gleichzeitig stattfindenden Lesson bei den Constraints MaxLessonsPerDayTeacherConstraint, MaxLecturesPerDayTeacherConstraint und MaxLessonsPerDaySemesterGroupConstraint nur einfach, das heißt nur die längste der Lessons, gezählt werden.

Alle anderen Constraints behalten ihre Gültigkeit und wenn ein Widerspruch zum LessonsAtSameTimeConstraint besteht, wird kein Stundenplan erstellt werden können.

Wie das Constraint umgesetzt wird ist im Abschnitt 3.4.1 Zeit- und Raumvariablen erläutert.

2.3.11 CourseAllInOneBlockConstraint

Für Kurse soll angegeben werden können, dass alle Lessons des Kurses direkt nacheinander und somit als Block stattfinden. Dies ermöglicht zum Beispiel mehrstündige Veranstaltungen bei denen die einzelnen Timeslots von verschiedenen Lehrenden gehalten werden. Es sind immer alle Lessons eines Kurses betroffen und die Reihenfolge der Lessons ist nicht festgelegt. Der Block darf nur innerhalb eines Tages stattfinden. Dies bedeutet, die Lessons des Kurses dürfen insgesamt nicht länger als die Anzahl Timeslots eines Tages sein.

Im MaxLessonsPerDayCourseConstraint, wird zugesichert, dass nicht mehrere Lessons desselben Kurses, an denen die gesamte Semestergruppe teilnimmt, am selben Tag stattfinden. Da dies im direkten Widerspruch zu diesem Constraint steht, wird dort wird für Lessons von Kursen, für die das CourseAllInOneBlockConstraint gilt, eine Ausnahme gemacht.

Die Formel 13 zeigt, in welchem zeitlichen Zusammenhang der Anfang der ersten Lesson eines Kurses und das Ende der letzten Lesson eines Kurses, der mit diesem Constraint belegt ist, stehen müssen. Genutzt wird dazu die zählende Nummerierung der Timeslots, sodass der Abstand zwischen den Nummern des ersten und des letzten Timeslots, die der Kurs belegt, der Gesamtanzahl an Timeslots aller Lessons des Kurses minus 1 entspricht.

$$\begin{gathered}\forall k \forall \tau_{min} \forall \tau_{max} \forall d : \\ IstAlsBlockKurs(k) \wedge IstBeginnVon(\tau_{min}, k) \wedge IstEndeVon(\tau_{max}, k) \wedge \\ IstGesamtdauerVon(d, k) \Longrightarrow \\ (\tau_{min} + d - 1 = \tau_{max}) \wedge AmSelbenTag(\tau_{min}, \tau_{max})\end{gathered}$$

Formel 13: CourseAsBlockConstraint

Die Prädikate der Formel 13:

- $IstAlsBlockKurs(k)$: k ist ein Kurs und ist mit dem CourseAsBlockConstraint belegt.
- $IstBeginnVon(\tau_{min}, k)$: τ_{min} ist die Nummer des Timeslots zu dem die erste Lesson des Kurses k beginnt.[2] Mit der ersten Lesson ist gemeint, dass diese vor allen anderen Lessons des Kurses stattfindet.
- $IstEndeVon(\tau_{max}, k)$: τ_{max} ist die Nummer des Timeslots zu dem die letzte Lesson des Kurses k endet. Mit der letzten Lesson ist gemeint, dass diese nach allen anderen Lessons des Kurses stattfindet.
- $IstGesamtdauerVon(d, k)$: Besagt, dass d die Summe aller Timeslots ist, die alle Lessons des Kurses belegen.

[2] Zu allen Timeslots des Stundenplans gehört eine Nummer. Die Nummern sind dabei aufzählend und beginnen mit eins für den ersten Timeslot am ersten Wochentag und enden mit der Anzahl aller Timeslots des Stundenplans für den letzten Timeslot des letzten Wochentags.

- $AmSelbenTag(\tau_{min}, \tau_{max})$: Die zu den Nummern τ_{min} und τ_{max} gehörenden Timeslots, finden am selben Tag statt.

Die Umsetzung ist im Abschnitt 3.4.8 CourseAllInOneBlockConstraints erläutert.

2.3.12 BlockInSameRoomConstraint

Wenn mehrere Lessons eines Kurses als Block, also direkt nacheinander, stattfinden, sollen diese alle im selben Raum stattfinden. Es wäre ungewöhnlich, wenn während so einer längeren Veranstaltung der Raum gewechselt werden müsste. Dieses Constraint soll für alle Blöcke gelten, die durch das CourseAllInOneBlockConstraint (2.3.11) gebildet werden. Diese beiden Constraints gelten also immer zusammen. Die Lessons können immer im selben Raum stattfinden, weil sie zum selben Kurs gehören und die möglichen Räume pro Kurs angegeben sind. Dieses Constraint schränkt die Lösungsmenge ein, da ein einzelner Raum für einen längeren Zeitraum belegt ist und dies nicht auf mehrere Räume aufgeteilt werden kann.

Bei einzelnen mehrstündigen Lessons gilt dieses Constraint bereits implizit, da für eine einzelne Lesson, egal wie viele Timeslots sie belegt, nur ein Raum angegeben ist, in dem sie stattfindet. Blöcke die durch das ConsecutiveLessonsConstraint zustande kommen, müssen dieses Constraint nicht erfüllen, da die Lessons jenes Constraints in der Regel zu verschiedenen Kursen gehören.

Da von dem CourseAllInOneBlockConstraint immer alle Lessons eines einzelnen Kurses betroffen sind, genügt es sicherzustellen, dass alle Lessons des Kurses im selben Raum stattfinden. Dies ist in Formel 14 gezeigt:

$$\begin{gathered}\forall k \forall l_1 \forall l_2 \exists r : \\ IstAlsBlockKurs(k) \land IstLessonVon(l_1, k) \land IstLessonVon(l_2, k) \land \\ IstMöglicherRaumVon(r, k) \land FindetStattIn(l_1, r) \implies \\ FindetStattIn(l_2, r)\end{gathered}$$

Formel 14: BlockInSameRoomConstraint

Die Prädikate der Formel 14:

- $IstAlsBlockKurs(k)$: k ist ein Kurs und alle seine Lessons sollen als Block stattfinden.
- $IstLessonVon(l, k)$: l ist eine Lesson des Kurses k.
- $IstMöglicherRaumVon(r, k)$: r ist ein Raum indem der Kurs k stattfinden kann.
- $FindetStattIn(l, r)$: Die Lesson l findet im Raum r statt.

Die Implementierung ist zusammen mit der des CourseAllInOneBlockConstraints in Abschnitt 3.4.8 CourseAllInOneBlockConstraints erläutert.

2.3.13 OneCoursePerDayTeacherConstraint

Bei der Erhebung der Anforderungen, wurde festgestellt, dass es bisher Praxis ist, für bestimmte Lehrende – meist Laboringenieure – nie mehrere Veranstaltungen verschiedener spezieller Kurse am selben Tag stattfinden zu lassen. Diese (Labor-)Veranstaltungen benötigen einen höheren Aufwand zur Vorbereitung und die Lehrenden sollen nicht mehrere solcher Veranstaltungen pro Tag vorbereiten müssen. Für die Stundenplanerstellung wird dies umgesetzt, indem solche Kurse markiert werden. Alle Lessons von so markierten Kursen, die ein Lehrender an einem Tag hält, müssen dann zum selben Kurs gehören.

Anders ausgedrückt, unterrichtet ein Lehrender Lessons verschiedener solcher Kurse, dürfen diese nicht am selben Tag stattfinden. Diese Formulierung ist in formaler Form in der Formel 15 gezeigt. Lessons von Kursen die nicht markiert sind, sollen für den Lehrenden zusätzlich stattfinden können und werden daher für dieses Constraint nicht beachtet.

$$\forall t \forall k_1 \forall k_2 \forall l_1 \forall l_2 :$$
$$IstLehrender(t) \wedge IstConstraintKurs(k_1) \wedge IstConstraintKurs(k_2) \ \wedge k_1 \neq k_2 \wedge$$
$$IstLessonVon(l_1, k_1) \wedge IstLessonVon(l_2, k_2) \wedge$$
$$Unterrichtet(t, l_1) \wedge Unterrichtet(t, l_2) \Longrightarrow$$
$$\neg GleicherTag(l_1, l_2)$$

Formel 15: OneCoursePerDayTeacherConstraint

Die Prädikate der Formel 15:

- $IstLehrender(t)$: t ist ein Lehrender.
- $IstConstraintKurs(k)$: k ist ein Kurs mit dem OneCoursePerDayTeacherConstraint.
- $IstLessonVon(l, k)$: l ist eine Lesson des Kurses k.
- $Unterrichtet(t, l)$: Der Lehrende t ist bei der Lesson l als Lehrender eingetragen.
- $GleicherTag(l_1, l_2)$: Die Lessons l_1 und l_2 finden am selben Tag statt.

Die Implementierung ist in Abschnitt 3.4.15 OneCoursePerDayTeacherConstraints erläutert.

2.3.14 MaxLessonsPerDayTeacherConstraint

Für jeden einzelnen Lehrenden soll angegeben werden können, wie viele 90 Minuten Blöcke (Timeslots) pro Tag für ihn maximal mit Veranstaltungen belegt werden dürfen. Die Anzahl verschiedener Lessons innerhalb des Tages ist dabei unerheblich, es zählt die Summe der Lessonlängen. Die Lessonlänge ist durch die Anzahl an Timeslots definiert, die diese belegt. Bei der Datengenerierung wird für die maximale Anzahl pro Tag ein

Default von 5 verwendet. Dieser Wert soll für jeden einzelnen Lehrenden separat angegeben werden können. Dieses Constraint dient dazu, den Stundenplan für die Lehrenden angenehmer zu machen und so unnötig volle Wochentage zu vermeiden. Es ist meist wünschenswert die Veranstaltungen eines Lehrenden über die ganze Woche gleichmäßig zu verteilen. Zu niedrige Werte erhöhen jedoch die Stundenplanfülle, da das Verhältnis zwischen Veranstaltungen und Platz, den diese belegen können, ungünstiger wird.

Die angegebene maximale Anzahl von belegten Timeslots pro Tag darf nicht kleiner sein als die maximale Lessonlänge einer Lesson, die der Lehrende hält (In dem Fall könnte kein Stundenplan erstellt werden). Lessons die durch das LessonsAtSameTimeConstraint gleichzeitig stattfinden, sollen nur einfach gezählt werden. In diesem Fall soll von den gleichzeitigen Lessons die längste gezählt werden.

In Formel 16 ist der einfache Zusammenhang gezeigt, dass die Anzahl tatsächlich belegter Timeslots an jeden Wochentag immer kleiner gleich der angegebenen maximalen Anzahl sein muss.

$$\forall t \forall w \forall m \forall s :$$
$$IstLehrender(t) \land IstWochentag(w) \land$$
$$IstMaximaleAnzahlProTag(m,t) \land IstTimeslotSumme(s,t,w) \implies$$
$$s \leq m$$

Formel 16: MaxLessonsPerDayTeacherConstraint

Die Prädikate der Formel 16:

- $IstLehrender(t)$: t ist ein Lehrender.
- $IstWochentag(w)$: w ist ein Wochentag des Stundenplans.
- $IstMaximaleAnzahlProTag(m,t)$: m ist die maximal erlaubte Anzahl an zu belegenden Timeslots des Lehrenden t pro Tag.
- $IstTimeslotSumme(s,t,w)$: s ist die Summe alle Lessonlängen der Veranstaltungen des Lehrenden t, die am Wochentag w stattfinden.

Die Details der Implementierung sind in Abschnitt 3.4.10 MaxLessonsPerDayTeacherConstraints zu finden.

2.3.15 MaxLecturesPerDayTeacherConstraint

Lectures, also Vorlesungen, sind bestimmte Lessons, die als solche markiert sind. Da angenommen wird, dass sie für die Lehrenden anstrengender sind und gegebenenfalls mehr Vorbereitung bedürfen, soll die Anzahl von Vorlesungen, die ein Lehrender an einem Tag geben muss, noch einmal separat zu der Einschränkung aller Lessons (MaxLessonsPerDayTeacherConstraint), eingeschränkt werden. Es zählt jedoch nicht die Anzahl unterschiedlicher Vorlesungen, sondern deren Längen. Dies wird als Anzahl von Timeslots

angegeben. Das Constraint gleicht dem MaxLessonsPerDayTeacherConstraint, abgesehen davon, dass nur die Lessons mit der Markierung als Lecture gezählt werden sollen. Auch hier gibt es für jeden Lehrenden eine eigene Angabe, wie viele Timeslots eines Tages maximal mit Vorlesungslessons belegt sein dürfen. Der Default Wert ist 3. Von Vorlesungen die durch das LessonsAtSameTimeConstraint gleichzeitig eingetragen sind, sollen jeweils nur die längsten gezählt werden.

In Formel 17 ist der Zusammenhang gezeigt, dass die Anzahl tatsächlich mit Vorlesungen belegter Timeslots an jeden Wochentag kleiner gleich der angegebenen maximalen Anzahl sein muss.

$$\forall t \forall w \forall m \forall s :$$
$$IstLehrender(t) \wedge IstWochentag(w) \wedge$$
$$IstMaximaleAnzahlProTag(m,t) \wedge IstVorlesungsTimeslotSumme(s,t,w) \Longrightarrow$$
$$s \leq m$$

Formel 17: MaxLecturesPerDayTeacherConstraint

Die Prädikate der Formel 17:

- $IstLehrender(t)$: t ist ein Lehrender.
- $IstWochentag(w)$: w ist ein Wochentag des Stundenplans.
- $IstMaximaleAnzahlProTag(m,t)$: m ist die maximal erlaubte Anzahl an mit Vorlesungen zu belegenden Timeslots des Lehrenden t pro Tag.
- $IstVorlesungsTimeslotSumme(s,t,w)$: s ist die Summe alle Lessonlängen der Vorlesungen des Lehrenden t, an Wochentag w.

Die Details der Implementierung sind in Abschnitt 3.4.14 MaxLecturesPerDayTeacherConstraints zu finden.

2.3.16 MaxLecturesAsBlockTeacherConstraint

Aus denselben Gründen wie beim MaxLecturesPerDayTeacherConstraint, soll eingeschränkt werden, wie viele Timeslots hintereinander Lehrende Vorlesungen halten muss. Diese Blöcke von Vorlesungen hintereinander gelten immer nur innerhalb eines Tages. In diesem Kontext sind mit Block nicht zwangsläufig mehrere Vorlesungen gemeint. Für eine einzelne Vorlesung, die sich über drei Timeslots erstreckt, ist die Blocklänge demnach 3. Diese kann selbstverständlich nicht durch das Constraint verkürzt werden, zählt aber dazu, wenn zum Beispiel im Anschluss eine weitere Vorlesung mit zwei Timeslots stattfinden würde. In diesem Fall wäre die Blocklänge für dieses Constraint 5, während im allgemeinen Fall von einem Doppelblock gesprochen würde, da zwei Lessons beteiligt sind.

Der Default Wert für die maximale Blocklänge soll 2 sein. Für Lehrende, die Vorlesungslessons haben, welche länger als 2 Timeslots sind, muss der Wert in jedem Fall höher sein.

Alternativ wäre es wünschenswert, für diese Lessons Ausnahmen zu machen, da dann andere Tage mit anderen Vorlesungen nicht ebenfalls von der erhöhten maximalen Anzahl betroffen wären. Aufgrund der aufwändigeren Implementierung wurde dieser Ansatz nicht gewählt. Eine Implementierung als optionales Constraint würde diesen Umstand ebenfalls lösen. Es ist aber gewünscht, dass die maximale Blocklänge in jedem Fall eingehalten wird[3].

In Lücken zwischen solchen Vorlesungsblöcken können jedoch andere Veranstaltungen des Lehrenden stattfinden. So wäre es beispielsweise möglich, dass ein Lehrender zuerst zwei Timeslots Vorlesung hat, dann einen Timeslot mit einer anderen Veranstaltung und dann wieder zwei Timeslots Vorlesung. Vorausgesetzt die maximale Vorlesungsanzahl pro Tag ist für diesen Lehrenden auf mindestens 4 gesetzt. So eine Konstellation wäre sicherlich nicht wünschenswert, da es keine größere Pause zwischen den langen Vorlesungen gibt. Eine Implementierung von speziellen Lücken zwischen zu vielen Vorlesungen scheint zu aufwendig. Eine allgemeinere Lösung mit einer maximalen Blockgröße nicht nur für Vorlesungen, sondern separat auch für Lessons, wie es bei der maximalen Anzahl pro Tag gelöst ist, wäre denkbar. Dies würde jedoch das Finden möglicher Stundenpläne stark einschränken, da es auch viele Blöcke ganz ohne Vorlesungen betreffen würde. So kommen beispielsweise verhältnismäßig lange Laborpraktika vor, die diese maximale Blocklänge aller Lessons ohnehin nach oben ziehen würden. Die Stundenplanfülle würde weiter erhöht werden. Außerdem wäre so eine Einschränkung von den Lehrenden vermutlich oftmals gar nicht gewollt.

In Formel 18 ist gezeigt, dass die Vorlesungsblockgrößen für jeden Lehrenden immer kleiner gleich der maximalen Blockgröße zu sein hat.

$$\forall t \forall m \forall vb : IstLehrender(t) \land$$
$$IstMaximaleBlockGröße(m,t) \land IstVorlesungsblockGröße(vb,t) \implies$$
$$vb \leq m$$

Formel 18: MaxLecturesAsBlockTeacherConstraint

Die Prädikate der Formel 18:

- $IstLehrender(t)$: t ist ein Lehrender.
- $IstMaximaleBlockGröße(m,t)$: m ist die maximal erlaubte Blockgröße mit Vorlesungen des Lehrenden t.

[3]Generell besteht das Problem bei zu vielen optionalen Constraints, deren Erfüllung allen anderen vorgezogen werden sollen, dass die relative Gewichtung aller optionaler Constraints zueinander aus dem Gleichgewicht gerät. Insbesondere, wenn von den sehr wichtigen Constraints einige deutlich öfter gebrochen werden können als andere. Daher scheint es ratsam bei solchen, eine Implementierung als harte Anforderung vorzuziehen, wenn dies möglich ist.

- $IstVorlesungsblockGröße(vb, t)$: vb ist die Länge in Timeslots eines Vorlesungsblocks des Lehrenden t.

Die Details der Implementierung sind in Abschnitt 3.4.13 MaxLecturesAsBlockTeacherConstraints zu finden.

2.3.17 MaxLessonsPerDaySemesterGroupConstraint

Diese Anforderung ist sehr ähnlich zu dem MaxLessonsPerDayTeacherConstraint, bezieht sich jedoch auf einzelne Semestergruppen anstatt auf Lehrende. Für jede Semestergruppe gibt es eine Angabe, wie viele Timeslots maximal an einem Tag mit Veranstaltungen belegt sein dürfen. Der Default Wert ist 5. Auch die Gründe für dieses Constraint sind gleich. Tage an denen für eine Semestergruppe von morgens bis abends Veranstaltungen stattfinden, dürften als sehr anstrengend empfunden werden. Solche Tage sind oft auch unnötig, wenn genug freie Stunden an anderen Wochentagen vorhanden sind.

Formel 19 zeigt, dass die tatsächlich belegten Lessons eines Wochentages für jede Semestergruppe kleiner gleich der maximalen Anzahl der jeweiligen Gruppe sein muss.

$$\begin{gathered}\forall sg \forall w \forall m \forall s : \\ IstSemesterGruppe(sg) \land IstWochentag(w) \land \\ IstMaximaleAnzahlProTag(m, sg) \land IstTimeslotSumme(s, sg, w) \implies \\ s \leq m\end{gathered}$$

Formel 19: MaxLessonsPerDaySemesterGroupConstraint

Die Prädikate der Formel 19:

- $IstSemesterGruppe(sg)$: sg ist eine Semestergruppe.
- $IstWochentag(w)$: w ist ein Wochentag des Stundenplans.
- $IstMaximaleAnzahlProTag(m, sg)$: m ist die maximal erlaubte Anzahl von mit Veranstaltungen belegten Timeslots pro Tag für die Semestergruppe sg.
- $IstTimeslotSumme(s, sg, w)$: s ist die Summe alle Lessonlängen der Veranstaltungen der Semestergruppe sg, an Wochentag w.

Die Details der Implementierung sind in Abschnitt 3.4.11 MaxLessonsPerDaySemesterGroupConstraints zu finden.

2.3.18 MaxLessonsPerDayCourseConstraint

Bei Kursen, die mehrere Lessons beinhalten, ist es normalerweise so, dass an einem Tag nicht mehrere Lessons dieses Kurses stattfinden, wenn diese nicht direkt aufeinanderfolgende Timeslots belegen. Da Lessons mehrstündig sein können, soll dies neben dem

CourseAllInOneBlockConstraint die normale Möglichkeit sein, mehrere Timeslots an einem Tag mit Veranstaltungen eines Kurses zu belegen. Dies gilt nur für Veranstaltungen an denen ganze Semestergruppen teilnehmen. Dieses Constraint soll daher verhindern, dass mehr als eine Lesson eines Kurses am selben Tag stattfindet. Das CourseAllInOneBlockConstraint bildet eine von mehreren Ausnahmen.

Um das direkte Aufeinanderfolgen aller Lessons eines Kurses zu erzwingen, gibt es das CourseAllInOneBlockConstraint. Für Kurse mit diesem Constraint, soll das MaxLessonsPerDayCourseConstraint daher nicht gelten.

Des Weiteren sollen Lessons an denen nur Teilgruppen teilnehmen (PartSemesterGroupConstraint) ohne Einschränkungen auch mehrfach und mit einer einzelnen Lesson desselben Kurses, an der die ganze Semestergruppe teilnimmt, am selben Tag stattfinden dürfen. Es ist zum Beispiel nicht unüblich, dass Praktika am selben Tag stattfinden, wie eine Vorlesung desselben Kurses[4].

Außerdem erzwingt das LessonsAtSameTimeConstraint das gleichzeitige Stattfinden mehrere Lessons, unter anderem auch desselben Kurses. Um das MaxLessonsPerDayCourseConstraint möglichst einfach zu halten, werden Lessons, die Teil eines LessonsAtSameTimeConstraints sind, völlig ignoriert. Diese Vereinfachung hat den Nachteil, dass so mehrere Lessons eines Kurses zusammen an einem Tag stattfinden, obwohl dies nicht gewünscht ist, genau dann, wenn mindestens eine der Lessons Teil eines LessonsAtSameTimeConstraints ist und mindestens eine andere nicht.

Auch das ConsecutiveLessonsConstraint kann das Stattfinden mehrere Lessons am selben Tag erzwingen. Für dieses Constraint wird hier jedoch keine Ausnahme gemacht, sodass Lessons die dadurch verknüpft sind, tatsächlich am selben Tag stattfinden dürfen müssen. Siehe dazu 2.3.19 ConsecutiveLessonsConstraint. Theoretisch könnte hierfür eine weitere Ausnahme gemacht werden. Es verkompliziert jedoch dieses Constraint und dessen Implementierung erheblich. Würde eine Ausnahme hinzugefügt, sollte diese so gestaltet werden, dass keine Lesson außerhalb des ConsecutiveLessonsConstraints (aber desselben Kurses) am selben Tag stattfinden kann, wie die Lessons ebenjenen Constraints. Dies bringt jedoch eine nicht unerhebliche Komplexität mit sich. Insbesondere Spezialfälle und Konflikte mit Lessons die gleichzeitig sowohl Teil eines ConsecutiveLessonsConstraints, eines LessonsAtSameTimeConstraints und gegebenenfalls noch eines CourseAllInOneBlockConstraints oder eines PartSemesterGroupConstraints sind, haben sich als sehr problematisch und schwer zu Überblicken herausgestellt. Es liegt daher nahe, dass sich die hohe Fehleranfälligkeit von solchen Ausnahmen nicht lohnt, um das Stundenplanmodell ein klein wenig flexibler zu machen.

Die Formel 20 zeigt, dass von den relevanten Lessons eines Kurses, der nicht als Block stattfinden soll, nur höchstens eine pro Tag stattfinden darf. Relevante Lessons sind die, an denen ganze Semestergruppen teilnehmen und die nicht erzwungenermaßen zeitgleich mit anderen Lessons stattfinden sollen. Sie dürfen also nicht Teil des PartSemesterGroup-

[4]Da Praktika meist in anderen Räumen stattfinden als die Vorlesung und die Benennung im Stundenplan durch den Kursnamen erfolgt, sollten Vorlesungen und Praktika im Normalfall sowieso durch zwei verschiedene Kurse modelliert werden.

Constraints oder eines LessonsAtSameTimeConstraints sein.

$$\forall k \forall w \forall a :$$
$$IstKurs(k) \land \neg AlsBlock(k) \land IstWochentag(w) \land$$
$$AnzahlRelevanterLessons(a, k, w) \Longrightarrow$$
$$a \leq 1$$

Formel 20: MaxLessonsPerDayCourseConstraint

Die Prädikate der Formel 20:

- $IstKurs(k)$: Es handelt sich bei k um einen Kurs.
- $AlsBlock(k)$: Der Kurs k unterliegt dem CourseAsBlockConstraint, dessen Lessons müssen daher alle am selben Tag nacheinander stattfinden.
- $IstWochentag(w)$: w ist einer der Wochentage des Stundenplans.
- $AnzahlRelevanterLessons(a, k, w)$: a ist die Anzahl der Lessons eines Kurses k, die am Wochentag w stattfinden. An den Lessons müssen ganze Semestergruppen teilnehmen und sie dürfen nicht Inhalt eines LessonsAtSameTimeConstraints sein.

Alternative Möglichkeiten dieses Constraints, die mehr Möglichkeiten zur Stundenplanmodellierung bieten, aber dieses Constraint weniger stark wie mit den oben genannten Ausnahmen verkomplizieren würde, wären zum einen, für jede Lesson einzeln anzugeben, ob diese zusammen mit anderen Lessons desselben Kurses am selben Tag stattfinden dürfen oder sollen. Es wäre auch möglich Gruppen von Lessons anzugeben, welche am selben Tag stattfinden sollen. Lessons die Teil der genannten Constraints sind, die Probleme bereiten, müssten dann auf diese Weise passend konfiguriert werden. Dies würde das Stundenplanmodell selber noch komplexer machen. Je komplexere und speziellere Möglichkeiten es gibt, das Stundenplanmodell vorzugeben, desto mehr mögliche Konflikte zwischen einzelnen Vorgaben kann es jedoch auch geben.

Die Details der Implementierung sind in Abschnitt 3.4.12 MaxLessonsPerDayCourseConstraints zu finden.

2.3.19 ConsecutiveLessonsConstraint

Dieses Constraint soll es ermöglichen, Lessons anzugeben, die direkt nach einer bestimmten anderen Lesson stattfinden sollen. Die Lessons müssen dann folglich auch am selben Tag stattfinden. So wird es zum Beispiel ermöglicht, die Reihenfolge der Lessons eines CourseAllInOneBlockConstraint Kurses vorzugeben. Hauptsächlich geht es jedoch darum, Lessons verschiedener Kurse oder Lessons an denen Teilgruppen teilnehmen, zu verknüpfen, da diese immer zusammen an einem Tag stattfinden dürfen. Im Gegensatz zu Lessons desselben Kurses, die normalerweise nicht am selben Tag stattfinden dürfen. Anwendungen sind zum Beispiel die Fälle wo eine Vorlesung direkt von einem Praktikum

gefolgt werden soll, oder wo mehrere Praktikumsgruppen direkt hintereinander stattfinden sollen. Das Constraint ist jedoch so allgemein gehalten, dass auch andere Anwendungsfälle ermöglicht werden, die es erfordern, dass zwei Lessons direkt nacheinander stattfinden. Um insbesondere das MaxLessonsPerDayCourseConstraint nicht erheblich umständlicher werden zu lassen, sollen die Lessons, die durch dieses Constraint nacheinander stattfinden, wie oben erwähnt, auch tatsächlich am selben Tag stattfinden dürfen. Durch diese Einschränkung müssen in anderen Constraints keine umständlichen Ausnahmen für dieses Constraint hinzugefügt werden. Es hat sich gezeigt, dass dies erhebliche Probleme mit sich bringen würde.

Die erstgenannte und zuerst stattfindende Lesson wird im Kontext dieses Constraints „Startlesson“, die Lessons die im Anschluss an die Startlesson stattfinden sollen, „Folgelessons“, genannt. Es sollen mehrere Folgelessons angegeben werden können. Es soll zum Beispiel angegeben werden können, dass die Folgelessons B und C direkt im Anschluss an die Startlesson A stattfinden. Im Falle mehrere Folgelessons, finden diese (B und C im vorigen Beispiel) implizit gleichzeitig statt. Folgelessons können wiederum Startlessons anderer Lessons sein.

Die Angabe dieses Constraints erfolgt über eine Liste für jede Lesson, welche die Folgelessons enthält. Standardmäßig ist die Liste der Folgelessons leer.

In Formel 21 wird gezeigt, dass der Beginn einer Folgelesson immer einen Timeslot nach dem letzten Timeslot der zugehörigen Startlesson liegen muss. Dazu wird der Umstand genutzt, dass alle Timeslots zählend nummeriert sind und diese Nummerierung somit eine Repräsentation der Timeslots darstellt, mit der wie mit Zeitangaben gerechnet werden kann. Des Weiteren müssen die Startlesson und die Folgelesson am selben Tag stattfinden[5].

$$
\begin{gathered}
\forall s \forall f \forall ende_s \forall start_f \forall tag_s \forall tag_f : \\
IstLesson(s) \wedge IstLesson(f) \wedge \\
IstFolgelessonVon(f, s) \wedge EndeVon(ende_s, s) \wedge StartVon(start_f, f) \wedge \\
TagVon(tag_f, f) \wedge TagVon(tag_s, s) \Longrightarrow \\
(start_f = ende_s + 1) \wedge (tag_f = tag_s)
\end{gathered}
$$

Formel 21: MaxLessonsPerDayCourseConstraint

Die Prädikate der Formel 21:

- $IstLesson(l)$: Gibt an, dass es sich bei l um eine Lesson handelt.
- $IstFolgelessonVon(f, s)$: Die Lesson f ist in der Liste der Folgelessons der Lesson s enthalten.

[5] Dies ist deshalb extra zu erwähnen, da im Stundenplanmodell der erste Timeslot eines (nicht ersten) Wochentages, der nächste Timeslot zum letzten Timeslot des Vortags ist.

- $StartVon(start, l)$: $start$ ist die zugehörige Nummer, des Timeslots zudem die Lesson l beginnt.[6]
- $EndeVon(ende, s)$: $ende$ ist die zugehörige Nummer, des letzten Timeslots den die Lesson l belegt.
- $TagVon(tag, l)$: Die Lesson l findet am Wochentag tag statt.

Es sollen für Lessons dieses Constraints keinerlei andere Constraints ignoriert werden. Aus diesem Grund müssen die involvierten Lessons am selben Tag stattfinden dürfen. Dies ist der Fall wenn diese Lessons:

- Zu verschiedenen Kursen gehören,
- An mindestens allen bis auf einer der Lessons nur ein Teil der Semestergruppe teilnimmt, oder
- Zu einem Kurs gehören, für den das CourseAllInOneBlockConstraint gilt.

Ferner müssen die Folgelessons (wenn es denn mehrere sind) einer Startlesson auch gleichzeitig stattfinden dürfen. Und ebenso müssen die Startlesson und Folgelesson nacheinander stattfinden dürfen, was insbesondere nicht der Fall wäre, wenn die Lesson durch das LessonsAtSameTimeConstraint verknüpft wären.

Das ConsecutiveLessonsConstraint wurde erdacht um die folgenden drei Szenarien zu ermöglichen. Andere Anwendungsfälle sind jedoch denkbar.

- Eine Vorlesung und ein Praktikum/Übung sollen direkt nacheinander stattfinden. Die Vorlesung und das Praktikum müssen dabei als zwei verschiedene Kurse modelliert werden.
- Bestimmte Praktikumlessons desselben Kurses, also Lessons an denen nicht die gesamte Semestergruppe teilnimmt, sollen nacheinander stattfinden.
- Bei einem Kurs mit dem CourseAllInOneBlockConstraint soll die Reihenfolge der Lessons vorgegeben werden.

Die Details der Implementierung sind in Abschnitt 3.4.9 ConsecutiveLessonsConstraints zu finden.

2.4 Optionale Anforderungen

Bei den optionalen Anforderungen geht es darum, bestimmte Merkmale des Stundenplans entweder zu bevorzugen oder möglichst auszuschließen. Sie unterscheiden sich dabei erheblich von den anderen Anforderungen. Sie sind so angelegt, dass bei größeren Stundenplänen meist nicht alle von ihnen erfüllt werden können. Eine Übersicht über die optionalen Constraints ist in Tabelle 2 zu sehen. In der rechten Spalte ist jeweils der Abschnitt aufgeführt, in dem die Implementierung erklärt ist. In einzelnen Unterabschnitten erfolgt jeweils eine Erläuterung der einzelnen optionalen Anforderungen.

[6] Zu allen Timeslots des Stundenplans gehört eine Nummer. Die Nummern sind dabei aufzählend

Constraint Name	Inhalt	Implementierung
PreferFirstStudyDayChoiceConstraint	2.4.1	3.5.1
AvoidLateTimeslotsConstraint	2.4.2	3.5.2
AvoidEarlyTimeslotsConstraint	2.4.3	3.5.2
AvoidGapBetweenLessonsSemesterGroupConstraint	2.4.4	3.5.3
AvoidGapBetweenDaysTeacherConstraint	2.4.5	3.5.4
FreeDaySemesterGroupConstraint	2.4.6	3.5.5

Tabelle 2: Übersicht der optionalen Anforderungen

Jedes einzelne Constraint beschreibt einzelne Eigenschaften im Stundenplan, zum Beispiel eine Lücke zwischen zwei Lessons aus Sicht einer Semestergruppe. Dies bedeutet, dass es für jedes der optionalen Constraints viele Fälle im Stundenplan gibt, bei denen das Constraint erfüllt oder verletzt werden kann. Die Constraints sind gewichtet. Die Anforderungen mit höherer Gewichtung sollen bei der Stundenplansuche, wenn möglich denen mit niedrigerer Gewichtung vorgezogen werden, in den Fällen bei denen nicht alle erfüllt werden können. Die Gewichtung erfolgt in Form von natürlichen Zahlen. Die Erfüllung von Constraints, die zum Beispiel fünfmal so stark gewichtet sind, wie ein anderes, wird bei der Lösungssuche bevorzugt. Bei diesem Beispiel würde ein Stundenplan einem anderen vorgezogen werden, auch wenn dass niedriger gewichtete Constraint bis zu viermal verletzt wird, um an einer Stelle, die Erfüllung des höher gewichteten Constraints zu ermöglichen.

Die Gewichtung sollte so gewählt werden, dass Constraints, bei denen auch nur eine einzelne Verletzung, eine hohe Auswirkung auf die Stundenplangüte hat, möglichst hoch gewichtet sind. Auch das Verhältnis der Häufigkeit, wie oft ein Constraint erfüllt oder verletzt werden kann, spielt eine Rolle. Haben zwei Constraints zwar die gleiche Gewichtung, das eine kann aber nur höchstens 10 mal und das andere 1.000 mal verletzt werden, so wird der Anteil des zweiten Constraints in der Bewertung der Stundenplangüte auch entsprechend höher sein.

Da es eine Gewichtung gibt, unterschiedet sich auch die Implementierung erheblich. Im Abschnitt 3.5 SoftConstraints, Implementierung wird gezeigt, wie die einzelnen Anforderungen implementiert sind. Die Angabe der Gewichtung ermöglicht es außerdem, einzelne optionale Constraints auszuschalten, indem für sie der Wert 0 als Gewicht gewählt wird.

Des Weiteren ist die Beendigung der Lösungssuche von der Erfüllung der optionalen Anforderungen abhängig. Ist eine Lösung gefunden worden, bei der alle optionalen Anforderungen vollständig erfüllt sind, oder es bewiesen ist, dass keine bessere Lösung existiert, endet die Suche sofort. Die Suche endet außerdem nach einer vorgegebenen maximalen Suchzeit. In diesem Fall, mit einer Lösung, die nicht optimal ist, für die nicht gezeigt werden konnte, ob sie optimal ist oder gegebenenfalls auch ohne Lösung, falls auch die Basis- und erweiterten Anforderungen nicht erfüllt werden konnten.

und beginnen mit eins, für den ersten Timeslot am ersten Wochentag und enden mit der Summe aller Timeslots des Stundenplans, für den letzten Timeslot des letzten Wochentags.

Wie funktioniert die Bewertung einer Stundenplanlösung durch die Gewichtungen der optionalen Constraints? Für einen ganzen Stundenplan werden für alle Verletzungen der optionalen Anforderungen ihre jeweiligen Gewichtungen addiert. Diese Summe ist das sogenannte Objective. Es ist der Wert, den es bei Constraint Programming zu optimieren gilt. In unserem Fall, da unerwünschte Eigenschaften gezählt werden, ist der Wert zu minimieren. Der Wert des Objectives wird durch die Optimierungsfunktion oder $objective function$ bestimmt. Nennen wir die Optimierungsfunktion o. Bei Optimierung mit Constraints besteht die Optimierungsfunktion aus Variablen die in den Constraints verwendet werden. Eine Variable könnte zum Beispiel den Betrag einer Fläche darstellen und wenn diese Fläche zum Beispiel minimiert werden soll, besteht die Funktion nur aus dieser Variablen.

Da in unserem Fall Vorgaben an den Stundenplan möglichst erfüllt sein sollen, muss für die Optimierungsfunktion gezählt werden, wie oft die Constraints erfüllt beziehungsweise nicht erfüllt sind. Im ersten Fall müsste der Wert von o maximiert, im zweiten Fall minimiert werden.

Für die Stundenplanbewertung sollen verletzte Constraints gezählt und der Wert von o - das Objective - daher minimiert werden. Für die Erklärung der einzelnen Constraints wird folgende Form der Optimierungsfunktion verwendet. Für jedes optionale Constraint und jede Möglichkeit im Stundenplan, bei der ein optionales Constraint erfüllt oder verletzt werden kann, gibt es eine Variable c_n. Das n gibt an zu welchem Constraint die Variable gehört. Da es für jedes Constraint so viele Variablen, wie Möglichkeiten zur Verletzung gibt, werden diese ebenfalls nummeriert: $c_{n,k}$. Dies ist in formaler Form in Formel 1 gezeigt. Ist an einer Stelle das Constraint verletzt, soll der Wert von $c_{n,k}$ 1 sein. Ist das Constraint an der Stelle erfüllt, ist der Wert 0 (Formel 2). Damit bildet die Funktion v die Menge der $c_{n,k}$ genannt $\mathbb{A}$, auf die Menge von 0 und 1 ab (Formel 3).

$$\mathbb{A} = \left\{ c_{n,k} \;\middle|\; \text{Constraint n, Fall k} \right\} \tag{1}$$

$$v(c_{n,k}) \quad \begin{cases} 0 & \text{wenn Constraint n bei Fall k nicht verletzt} \\ 1 & \text{wenn Constraint n bei Fall k verletzt} \end{cases} \tag{2}$$

$$v: \; \mathbb{A} \to \{0, 1\} \tag{3}$$

In der Optimierungsfunktion o wird dann jedes $v(c_{n,k})$ mit dem Gewicht g_n des jeweiligen Constraints multipliziert. Wie die Optimierungsfunktion dann aussieht ist in den Formeln 4 bis 6 zu sehen. k_n ist dabei die Anzahl an Fällen, in denen das Constraint n verletzt sein kann.

$$o: \left\{ \mathbb{A} \to \{0, 1\} \right\} \longrightarrow \mathbb{N}_0 \tag{4}$$

$$o(v) = g_1 \sum_{i=1}^{k_1} c_{1,i} + ... + g_n \sum_{i=1}^{k_n} c_{n,i} \tag{5}$$

$$\iff \sum_{p=1}^{n} g_p \cdot \sum_{i=1}^{k_p} c_{p,i} \tag{6}$$

Bei den Erklärungen zu den einzelnen optionalen Constraints, wird unter anderem jeweils erläutert unter welchen Umständen der Wert des $c_{n,k}$ gleich 1 oder 0 ist. Es wird also gezeigt, wie die Funktionswerte der Funktion v errechnet werden. Diese Werte werden dort immer als Variable c bezeichnet. Der Teil der Optimierungsfunktion eines Constraints n ist dann $g_n \cdot (v(c_{n,1}) + ... + v(c_{n,k}))$. Bei der eigentlichen Implementierung der optionalen Constraints werden die Summanden $v(c_{n,1}) + ... + v(c_{n,k})$ meist zu einer Variable zusammengefasst, die angibt wie oft das Constraint verletzt wird. Um die Constraints formal darzustellen, ist es jedoch nützlich sich eine einzelne Variable pro Verletzungsmöglichkeit des Constraints vorzustellen[7].

Im Sinne der Optimierung wird von einer Bestrafung gesprochen, wenn ein Constraint nicht erfüllt ist. Die Bestrafung entspricht einer Erhöhung des Wertes von o um den Wert des Gewichtes des jeweiligen Constraints.

2.4.1 PreferFirstStudyDayChoiceConstraint

Die harte Anforderung StudyDayConstraint stellt bereits sicher, dass eine der beiden Wahlmöglichkeiten des Studientags eines Lehrenden immer umgesetzt wird. Dieses Constraint setzt nun um, dass die Erstwahl der Zweitwahl bevorzugt wird. Dies bedeutet, dass durch die Objective Function für jeden Lehrenden, der nur seine Zweitwahl (nicht aber seine Erstwahl) als Studientag bekommen hat, der Wert der Gewichtung dieses Constraint zum Objective zu addieren ist. Es wird für jeden Lehrenden mit einem Studientag angewandt, der maximale Einfluss durch dieses Constraint lässt sich folgend berechnen:

$$lehrende\_mit\_studientag \cdot gewicht\_des\_constraints \tag{7}$$

Der Term der Optimierungsfunktion o für dieses Constraint ist in Gleichung 8 zu sehen. Dabei ist g das Gewicht dieses Constraints und k die Anzahl Lehrender mit einem Studientag. Jedes c ist also einem Lehrenden mit Studientag zugeordnet.

$$o_c = g \cdot (c_1 + c_2 + ... + c_k) \tag{8}$$

In Formel 22 ist gezeigt wie entschieden wird, ob die einzelnen c Variablen den Wert 1 oder 0 zugewiesen bekommen. c ist daher ein Summand der Optimierungsfunktion.

[7] Diese einzelnen Variablen existieren tatsächlich auch bei den Implementationen einiger Constraints, jedoch müssen sie nicht extra erstellt werden, da sie bereits zur Umsetzung anderer Constraint verwendet werden und nur in einem anderen Kontext wiederverwendet werden.

Die letzten beiden Zeilen der Formel, zeigen die Fallunterscheidung. Existiert eine Lesson, die am Erstwahltag stattfindet, ist das Constraint, dass keine Veranstaltungen am Erstwahltag stattfinden sollen, verletzt.

$$\forall t \forall ew \forall c :$$
$$IstLehrender(t) \land IstErstwahlTag(ew, t) \land$$
$$IstOVVariable(c, t) \land$$
$$(\exists l : IstLesson(l) \land Unterrichtet(t, l) \land FindetStattAm(l, ew) \implies c = 1) \land$$
$$(\neg \exists l : IstLesson(l) \land Unterrichtet(t, l) \land FindetStattAm(l, ew) \implies c = 0)$$

Formel 22: PreferStudyDayChoiceConstraint

Die Prädikate der Formel 22:

- $IstLehrender(t)$: Gibt an, dass es sich bei t um einen Lehrenden handelt.
- $HatStudientag(t)$: Gibt an, dass dem Lehrenden t ein Studientag zusteht.
- $IstErstwahlTag(ew, t)$: ew ist der Wochentag, den der Lehrende t als Erstwahl für seinen Studientag ausgewählt hat.
- $IstOVVariable(c, t)$: c ist die Variable in der Optimierungsfunktion, die für den Lehrenden t und die Erfüllung seiner Erstwahl für den Studientag steht.
- $IstLesson(l)$: Es handelt sich bei l um eine Lesson.
- $Unterrichtet(t, l)$: Die Lesson l wird vom Lehrenden t gehalten.
- $FindetStattAm(l, ew)$: Die Lesson l findet am Wochentag ew statt.

Die Details der Implementierung ist in Abschnitt 3.5.1 PreferFirstStudyDayChoiceConstraint zu finden.

2.4.2 AvoidLateTimeslotsConstraint

Veranstaltungen zu späterer Stunde sind bei den meisten Studierenden und Lehrenden eher unbeliebt. Sie ziehen außerdem einen Studierenden- oder Arbeitstag oft auseinander, da durch Veranstaltungen die an beiden Enden des Tages stattfinden (morgens und abends) quasi der ganze Tag belegt ist. Daher sollen Veranstaltungen im letzten und im vorletzten Timeslot jedes Tages vermieden werden. Beim Stundenplan der TH-Lübeck sind dies der 5. und 6. Timeslot. Sie sollen verschieden stark gewichtet werden können. Dieses Constraint ist also in zwei Constraints aufgeteilt, einmal für den fünften und einmal für den sechsten Timeslot eines Tages, im Folgendem AvoidSixthTimeslotConstraint und AvoidFifthTimeslotConstraint genannt. Eine Lesson kann auch beide Constraints verletzen, wenn sie mindestens die Länge zwei hat und zum 5. und 6. Timeslot eines Tages stattfindet.

Diese beiden Constraints, werden bei üblichen Hochschulstundenplänen mit Sicherheit nicht immer eingehalten werden können. Dazu gibt es für gewöhnlich zu viele Veranstaltungen und die Timeslots sind ja Bestandteil des Stundenplans und daher für gewöhnlich auch notwendig. Daher sollte dieses Constraint relativ niedrig gewichtet werden, zumal es für jede einzelne Lesson angewandt wird und es daher auch viele Möglichkeiten gibt, für die sich das Objective durch dieses Constraint erhöhen kann. Die Constraints stellen jedoch sicher, dass nur Veranstaltungen spät am Tag stattfinden wenn dies unbedingt notwendig ist. Es wäre sinnvoll die Gewichtung für den 6. Timeslot höher zu wählen. Dies würde zum einen für einen kompakteren Stundenplan sorgen (weniger Lücken im Verlauf des Tages) und zum anderen sind Veranstaltungen im 6. Timeslot sicherlich noch unbeliebter als Veranstaltungen im 5. Timeslot.

Dieses Constraint stellt eine Abhängigkeit an den Stundenplan der TH-Lübeck dar, welcher 6 Timeslots pro Tag besitzt. Bei Stundenplänen mit mehr oder weniger als 6 Timeslots kann es sinnlos oder sogar unpassend sein, Veranstaltungen im 5. und 6. Timeslot zu vermeiden. Es wäre möglich das Constraint so abzuändern, dass es sich immer auf den letzten beziehungsweise vorletzten Timeslot eines Tages bezieht. Da die Festlegung auf den 5. und 6. Timeslot jedoch die Implementierung zunächst einfacher machte und versucht wurde, diese so performant wie möglich zu gestalten und es bei der Arbeit konkret um den Stundenplan der TH-Lübeck gehen soll, soll es bei einer Festlegung auf diese beiden Zahlen bleiben. Die Implementierung wurde jedoch im Laufe dieser Arbeit so angepasst, dass sie für beliebige Timeslots eines Tages funktioniert.

Im Vergleich zum PreferFirstStudyDayChoiceConstraint, kann die maximale Auswirkung dieses Constraints auf das Objective nicht auf einfache Weise eingeschätzt werden. Sie ist in jedem Fall kleiner-gleich der Gewichtung, multipliziert mit der Anzahl an Lessons im Stundenplan. Da es aber immer viele freie Timeslots gibt, die von diesem Constraint nicht bestraft werden, wird immer nur ein kleiner Bruchteil der Lessons tatsächlich zum 5. und ein noch kleinerer Teil zum 6. Timeslot stattfinden.

Der Term der Optimierungsfunktion o für das AvoidSixthTimeslotConstraint beziehungsweise AvoidFifthTimeslotConstraint ist in Gleichung 9 zu sehen. Es gibt einen Summanden für jede Lesson, da jede Lesson potentiell zum 5. oder 6. Timeslot stattfinden kann. k ist daher die Anzahl aller Lessons. Um das Constraint einfach zu halten, werden zum Beispiel Lessons die nur vormittags stattfinden sollen nicht ausgeschlossen. Dies wäre lediglich eine mögliche Optimierung bei der Implementierung.

$$o_c = g \cdot (c_1 + c_2 + ... + c_k) \tag{9}$$

In den Formeln 23 und 24 ist die Fallunterscheidung zu sehen, durch die entschieden wird, welchen Wert die Variable c der Optimierungsfunktion annimmt. c steht dabei für eine Lesson des Stundenplans.

$$\forall l \forall c :$$
$$IstLesson(l) \land IstOVVariable(c, l) \land$$
$$(BelegtTimeslot(l, 6) \Longrightarrow c = 1) \land$$
$$(\neg BelegtTimeslot(l, 6) \Longrightarrow c = 0)$$

Formel 23: AvoidSixthTimeslotFormel

und

$$\forall l \forall c :$$
$$IstLesson(l) \land IstOVVariable(c, l) \land$$
$$(BelegtTimeslot(l, 5) \Longrightarrow c = 1) \land$$
$$(\neg BelegtTimeslot(l, 5) \Longrightarrow c = 0)$$

Formel 24: AvoidFifthTimeslotFormel

Die Prädikate der Formel 23 und 24:

- $IstLesson(l)$: Gibt an, dass es sich bei l um eine Lesson handelt.
- $IstOVVariable(c, l)$: c ist die Variabel in der Optimierungsfunktion, die für die Lesson l und die Erfüllung des AvoidFifthTimeslotConstraint bzw. AvoidSixthTimeslotConstraint in Bezug zu dieser Lesson steht.
- $BelegtTimeslot(l, n)$: Die Lesson l belegt einen Timeslot mit der Nummer n. Wobei n angibt als wievielter Timeslot der Timeslot an seinem Tag stattfindet[8]. Das Prädikat ist auch erfüllt, wenn die Lesson mehrstündig ist (und dann nur mit einem Teil der Lesson den entsprechenden Timeslot belegt).

Die Details der Implementierung sind in Abschnitt 3.5.2 CountLessonsAtHour zu finden.

2.4.3 AvoidEarlyTimeslotsConstraint

Genau wie Veranstaltungen zu späten Stunden, können auch Veranstaltungen zu früh am Morgen unbeliebt sein. Ob es gewollt ist diese zu vermeiden, hängt sicherlich von der Einstellung der Beteiligten bei der Stundenplanerstellung dazu ab. Jedes weiche Constraint kann deaktiviert werden, indem dessen Gewichtung auf null gesetzt wird.

Das AvoidEarlyTimeslotsConstraint setzt die Vermeidung von Veranstaltungen im ersten Timeslot eines jeden Wochentages um. Vermutlich ist dies den Verantwortlichen, von allen weichen Anforderungen am unwichtigsten, dann empfiehlt es sich, hierfür die niedrigste Gewichtung zu wählen.

[8]Die hier genutzte Nummer entspricht nicht der zählenden Nummerierung die für alle Timeslots einzigartig ist. Stattdessen gibt es zu jedem Timeslot zusätzlich eine Nummer, welche angibt um den wievielten Timeslot des Tages, zu dem der Timeslot gehört, es sich handelt.

Die formale Definition des Constraints entspricht denen der AvoidLateTimeslotsConstraints und ist in Formel 25 gezeigt.

$$\forall l \forall c :$$
$$IstLesson(l) \wedge IstOVVariable(c, l) \wedge$$
$$(BelegtTimeslot(l, 1) \Longrightarrow c = 1) \wedge$$
$$(\neg BelegtTimeslot(l, 1) \Longrightarrow c = 0)$$

Formel 25: AvoidFirstTimeslotFormel

Die Prädikate der Formel 25:

- $IstLesson(l)$: Gibt an, dass es sich bei l um eine Lesson handelt.
- $IstOVVariable(c, l)$: c ist die Variable in der Optimierungsfunktion, die für die Lesson l und die Erfüllung des AvoidEarlyTimeslotsConstraint in Bezug zu dieser Lesson steht.
- $BelegtTimeslot(l, n)$: Die Lesson l belegt den Timeslot eines Tages mit der Nummer n. Wobei n angibt als wievielter Timeslot der Timeslot an seinem Tag stattfindet. Das Prädikat ist auch erfüllt, wenn die Lesson mehrstündig ist (und dann selbstverständlich nur mit einem Teil der Lesson den entsprechenden Timeslot belegt).

Die Details der Implementierung ist in Abschnitt 3.5.2 CountLessonsAtHour zu finden.

2.4.4 AvoidGapBetweenLessonsSemesterGroupConstraint

Besonders für Studierende, welche ja im Vergleich zu den meisten Lehrenden keine eigenen Büros auf dem Campusgelände besitzen, sind Lücken im Stundenplan innerhalb eines Tages oft unerwünscht. Abgesehen von der zu überbrückenden Zeit, verlängern sie auch die Dauer eines Studierendentages. Eine Lücke beschreibt einen oder mehrere Timeslots, zu denen keine Veranstaltungen stattfinden, davor und danach jedoch schon. Durch dieses Constraint, sollen solche Lücken vermieden werden.

Lücken sind immer aus der Sicht einer Semestergruppe zu betrachten, sodass jeweils nur Veranstaltungen zu betrachten sind, an denen die jeweilige Semestergruppe teilnimmt. Bei einem Stundenplan mit 6 Timeslots pro Tag, sind 4 verschiedene Lücken denkbar. Lücken die ein, zwei, drei oder vier Timeslots lang sind. Die Vermeidung der einzelnen Lückentypen, sollen jeweils verschieden stark gewichtet werden können. Würden alle Lücken gleich gewichtet werden, wäre der maximale Einfluss auf das Objective immer kleiner als

$$semestergruppen \cdot wochentage \cdot 2 \cdot gewicht\_des\_constraints, \tag{10}$$

da bei 6 Timeslots am Tag maximal zwei Lücken pro Tag auftreten können. Wichtig zu erwähnen ist aber, dass die Anzahl an Variablen für die Optimierungsfunktion deutlich über der Anzahl tatsächlich stattfindbarer Lücken liegt. Es gibt eine Variable c für jeden

Timeslot – beziehungsweise Gruppe von Timeslots – zu denen eine Lücke auftreten kann. Pro Tag und Semestergruppe gibt es zum Beispiel 4 Timeslots zu denen eine Lücke der Größe 1 auftreten kann (Bei einer Tageslänge von 6 Timeslots). Es ist dabei unerheblich, dass nie zu allen dieser Timeslots gleichzeitig Lücken auftreten können.

Der Term o_c der Objective Function für dieses Constraint teil sich in vier Teile auf, welche jeweils für eine Lückenlänge stehen.

$$o_c = l_1 + l_2 + l_3 + l_4 \tag{11}$$

Jeder Teil besteht aus Werten für jede mögliche Lücke, für jede Semestergruppe, jeweils multipliziert mit der Gewichtung der Lücke. Einen Timeslot große Lücken, sind vier pro Tag möglich, insgesamt also 20. Der Term l_1 enthält also für jede Semestergruppe 20 Variablen c, die jeweils für einen Timeslot stehen, an dem eine Lücke auftreten kann und alle mit dem Gewicht der Lücke multipliziert werden. Bei Term l_4 sind es nur noch 5 Variablen je Semestergruppe, da es an jedem Tag (mit 6 Timeslots) nur eine Möglichkeit gibt, zu der eine Lücke der Größe 4 stattfinden kann. In Formel 26 wird für Lücken der Größe 1 formal beschrieben, unter welchen Umständen die Variablen c entweder den Wert 1 oder 0 hat. Die Variable c steht immer für eine Semestergruppe und einen Timeslot an dem eine Lücke sein kann.

$$\begin{gathered}\forall sg \forall \tau \forall c : \\ IstSemestergruppe(sg) \wedge IstTimeslot(\tau_{+1}) \wedge \\ IstOVSummand(c, \tau, sg) \wedge \\ (Lücke(\tau, sg) \Rightarrow c = 1) \wedge \\ (\neg Lücke(\tau, sg) \Rightarrow c = 0)\end{gathered}$$

Formel 26: AvoidGapFormel

Die Prädikate der Formel 26:

- $IstSemestergruppe(sg)$: sg ist eine Semestergruppe.
- $IstTimeslot(\tau)$: τ ist ein Timeslot des Stundenplans.
- $IstOVSummand(c, \tau, sg)$: c ist die Variable in der Optimierungsfunktion, die für das Vorkommen einer Lücke der Größe eins, für die Semestergruppe sg, zum Timeslot τ und damit der Erfüllung des AvoidGapBetweenLessonsSemesterGroupConstraints, steht.

Gezeigt wird lediglich, dass bei einer Lücke zum Zeitslot τ der Wert der Variablen c dem Wert 1 entspricht. Wie ermittelt wird, ob eine Lücke besteht, also der Inhalt des Prädikats $Lücke(\tau, sg)$ ist in Formel 27 gezeigt. Das Prädikat soll wahr sein, wenn für die Semestergruppe sg zum Timeslot τ eine Lücke im Stundenplan ist. Es werden sich die Timeslots vor und nach dem Timeslot τ angeschaut. Existiert jeweils eine Lesson

an der die Semestergruppe teilnimmt und die vor und nach dem Timeslot stattfinden, jedoch keine die zum Timeslot τ stattfindet, liegt eine Lücke vor. Die Timeslots davor und danach sollen immer Timeslots desselben Tages sein.

$$L\ddot{u}cke(\tau, sg) \coloneqq \exists l_{-1} \neg \exists l \exists l_{+1} \forall \tau_{-1} \forall \tau_{+1} :$$
$$LessonsVon(l_{-1}, l, l_{+1}, sg) \wedge Timeslots(\tau_{-1}, \tau, \tau_{+1}) \wedge$$
$$Aufeinanderfolgend(\tau_{-1}, \tau) \wedge Aufeinanderfolgend(\tau, \tau_{+1}) \wedge$$
$$FindetStattUm(l_{-1}, \tau_{-1}) \wedge FindetStattUm(l, \tau) \wedge FindetStattUm(l_{+1}, \tau_{+1})$$

Formel 27: Lücke Prädikat

Die Prädikate der Formel 26:

- $LessonsVon(l_1, l_2, ..., l_n, sg)$: Gibt an, dass es sich bei l_1 bis l_n um Lessons handelt, an denen die Semestergruppe sg teilnimmt.
- $Timeslots(\tau_1, \tau_2, ..., \tau_n)$: Gibt an, dass es sich bei τ_1 bis τ_2 um Timeslots des Stundenplans handelt.
- $Aufeinanderfolgend(\tau_1, \tau_2)$: Die Timeslots τ_1 und τ_n folgen direkt aufeinander und liegen am selben Wochentag.
- $FindetStattUm(l, \tau)$: Die Lesson l findet zum Timeslot τ statt.

Unschön an diesem Constraint ist, dass zum einen Lücken durch Veranstaltungen gefüllt werden, an denen nur ein Teil der Semestergruppe teilnimmt. Für alle anderen Studierenden der Semestergruppe, die nicht an dieser Veranstaltung teilnehmen, besteht in diesen Fällen trotzdem eine Lücke. Um dies zu optimieren wäre es möglich Fälle zu bestrafen, wenn Veranstaltungen an denen nur ein Teil der Semestergruppe teilnimmt einzeln und nicht zusammen mit anderen solchen Veranstaltungen derselben Semestergruppe stattfinden. Dies würde dazu führen, dass wenn immer möglich, mehrere dieser Veranstaltungen gleichzeitig stattfinden. Alternativ oder zusätzlich wäre es möglich, Veranstaltungen nur mit Teilen der Semestergruppe, bevorzugt am Anfang oder Ende eines Studierendentages der Semestergruppe stattfinden zu lassen. Dann entstünden in diesen Fällen keine Lücken für den Rest der Gruppe.

Dies könnte eventuell dadurch erreicht werden, indem solche Veranstaltungen keine Lücken füllen können, im Sinne dieses Constraint also nicht als Veranstaltung gezählt würden. Vermutlich würden durch den Suchalgorithmus diese Veranstaltungen dann automatisch bevorzugt an Anfang und Ende der Studierendentage gelegt werden. Eine Untersuchung und Implementierung hierzu hat jedoch den Umfang dieser Arbeit überstiegen.

Anmerkung:
Auf den ersten Blick, scheint dieses Constraint eine Optimierung darzustellen, welche recht speziell und gewissermaßen ein Zusatzelement („Nice to have“) ist, welches bei einer automatischen Stundenplanerstellung, bequem zusätzlich eingebaut werden kann. Es

ist jedoch ziemlich essentiell um eine Stundenplanqualität zu vermeiden, die der eines jeden manuell erstellten Stundenplan weit unterliegt. Üblicherweise würde man bei der manuellen Erstellung die Veranstaltungen zunächst automatisch so legen, dass sie direkt aneinanderliegen. Und wenn dies nicht möglich ist, würde ein Mensch die Lücken automatisch so klein wie möglich wählen. Bei der automatischen Erstellung kann dies lediglich optimiert werden, da viel mehr Kombinationen erfasst werden. Es stellt aber kein völlig neues Element dar.

Indirekt vermeiden allerdings auch bereits das AvoidLateTimeslotsConstraint und das AvoidEarlyTimeslotsConstraint Lücken, insbesondere Größere, weil sich der bevorzugte Zeitraum für die Veranstaltungen auf drei Timeslots pro Tag verdichtet.

Die Details der Implementierung sind in Abschnitt 3.5.3 CountGapsBetweenLessonsSemesterGroup zu finden.

2.4.5 AvoidGapBetweenDaysTeacherConstraint

Bei der Evaluierung der benötigten Anforderungen, kam der Wunsch einiger Professoren auf, zwischen Tagen mit Veranstaltungen möglichst keine Tage Pause zu haben. Also Tage ohne Veranstaltungen für sie. Grund war zum Beispiel, dass die Professoren weiter entfernt wohnen und nur unter der Woche in der Nähe der Hochschule wohnen. Da es nicht alle Lehrenden betrifft, soll es für jeden einzelnen Lehrenden an- bzw. abschaltbar sein. Es sollen im Kontext der Optimierungsfunktion Tage ohne Veranstaltungen bestraft werden, welche sich zwischen Tagen mit Veranstaltungen befinden. Dies können folglich Dienstag, Mittwoch oder Donnerstag sein. Es wurde zusätzlich erwogen, ob der Studientag des Lehrenden nicht als Lücke gezählt werden soll. Dies scheint zunächst intuitiv. Jedoch ist die Auswirkung, den Studientag mitzuzählen, relativ unbedeutend und betrifft nur Lehrende (für die dieses Constraint aktiviert ist), deren eine Studientagswahl, einer der inneren Wochentage, die andere aber ein Montag oder Freitag ist. Im Sinne einer möglichst einfachen Implementierung ist diese Vereinfachung daher zu bevorzugen. Die Auswirkungen werden nun kurz erläutert.

Zunächst kann durch einen Studientag eine Lücke der Größe eins an einem Dienstag, Mittwoch oder Donnerstag entstehen. Diese ist dann zwangsläufig und erhöht zwar den Wert der Optimierungsfunktion, dies hat jedoch keinen negativen Einfluss auf die Stundenplansuche, nur der Objective Value für den optimalen Stundenplan ist dann etwas höher (und niemals 0). Wichtig ist es jedoch, dass die Gewichtung aller Tageslücken der Größe eins, höher ist, als die Gewichtung des PreferFirstStudyDayChoiceConstraints. Andernfalls würde die Vermeidung der Lücke, vor der Erfüllung der Erstwahl, bevorzugt, was sicherlich nicht im Sinne des Lehrenden ist.

Bei Lücken der anderen Größen, erhöht dieses Constraint ebenfalls nur den Wert der Optmierungsfunktion, für den Fall, dass die Vermeidung größere Lücken höher Gewichtet wird. Zum Beispiel wird eine Lücke von sonst zwei Tagen am Dienstag und Mittwoch und einem Studientag am Donnerstag dann als Lücke von drei Tagen gezählt.

Allerdings ist hinzuzufügen, dass bei Lehrenden auf die die oben genannten Bedingungen für diese Auswirkungen zutreffen, eine Verzerrung des Gewichts für die Bevorzugung der Erstwahl des Studientages im Vergleich zu anderen weichen Anforderungen auftritt, da die Veränderung des Objectives welche mit Erfüllung oder Nichterfüllung der Erstwahl größer oder kleiner als die eigentliche Gewichtung des PreferFirstStudyDayChoiceConstraints ausfällt.

Bei einer Arbeitswoche von fünf Tagen, sind sechs verschiedene Lücken denkbar. Drei mit einem Tag Pause, zwei mit zwei Tagen und die Lücke von Dienstag bis Donnerstag. Für jede soll eine unterschiedliche Gewichtung angegeben werden können.

Der Term o_c der Objective Function für dieses Constraint teilt sich in vier Teile auf, welche jeweils für eine Lückenlänge stehen.

$$o_c = l_1 + l_2 + l_3 \tag{12}$$

Jeder Teil besteht aus Variablen c für jede mögliche Lücke, jeweils für jeden Lehrenden für den dieses Constraint gilt und jeweils multipliziert mit dem Gewicht der jeweiligen Lücke. Die Variable c hat dann den Wert 1, wenn eine entsprechende Lücke vorhanden ist und den Wert 0 bei keiner solchen Lücke. In Formel 28 wird formal beschrieben, unter welchen Umständen diese Wertbelegung erfolgt. Als Beispiel dient die Lücke an einem Dienstag.

$$\begin{gathered}\forall t \forall c : \\ IstLehrender(t) \wedge \\ IstOVVariable(c, Dienstag, t) \wedge \\ (DienstagsLücke(t) \implies c = 1) \wedge \\ (\neg DienstagsLücke(t) \implies c = 0)\end{gathered}$$

Formel 28: AvoidTuesdayGapFormel

Die Funktionsweise des Prädikats $DienstagsLücke(t)$ ist in Formel 29 gezeigt.

$$\begin{gathered}DienstagsLücke(t) \coloneqq \exists l_1 \neg \exists l_2 \exists l_3 : \\ LessonsVon(l_1, l_2, l_3, t) \wedge \\ AmMontag(l_1) \wedge AmDienstag(l_2) \wedge AmMittwoch(l_3)\end{gathered}$$

Formel 29: DienstagsLücke Prädikat

Die Prädikate der Formeln 28 und 29:

- $IstLehrender(t)$: Bei t handelt es sich um einen Lehrenden.
- $IstOVVariable(c, Dienstag, t)$: c ist die Variable in der Optimierungsfunktion, die für eine Lücke am Dienstag für den Lehrenden t steht.

- $LessonsVon(l_1, l_2, ..., l_n, t)$: Gibt an, dass es sich bei l_1 bis l_n um Lessons handelt die der Lehrende t gibt.
- $AmMontag(l)$: Die Lesson l findet am Montag statt. Entsprechend gelten $AmDienstag(l)$ und $AmMittwoch(l)$.

Die Lösung für das grundlegende Problem, welche dieses Constraint darstellt, ist nicht optimal. Eigentlich wollen die Professoren vermutlich direkt aufeinanderfolgende Arbeitstage, welche aber nicht die gesamte Arbeitswoche belegen. Gewissermaßen ein möglichst langes Wochenende entsprechend der Möglichkeiten die durch die Anzahl ihrer Veranstaltungen gegeben sind. Insbesondere der Umstand, dass eine einzelne Veranstaltung an einem Tag, so eine Lücke schließt, kann sich kontraproduktiv auswirken, da die Veranstaltungen unter Umständen unnötig auseinandergezogen werden. Dies betrifft besonders Lehrende mit wenigen Veranstaltungen. So würden, bedingt durch den Suchablauf, eventuell bei einem Lehrenden mit drei Veranstaltungen, diese an drei aufeinanderfolgenden Tagen stattfinden, obwohl dies an einem Tag möglich wäre. Eine Vermeidung von Tagen mit nur einer oder zwei Veranstaltungen wäre möglich und würde eventuell insgesamt bei allen Lehrenden zu einem ausgeglichenerem Stundenplan führen, bei dem die Veranstaltungen gleichmäßiger auf die Arbeitstage verteilt sind. Für den Rahmen dieser Arbeit soll es jedoch bei der oben vorgestellten, einfachen Lösung bleiben.

Die Details der Implementierung sind in Abschnitt 3.5.4 CountGapsBetweenDaysTeacher zu finden.

2.4.6 FreeDaySemesterGroupConstraint

Bei der Ermittlung der Anforderungen, kam auf, dass einige Semestergruppen, insbesondere von kleineren Masterstudiengängen, manchmal einen Wochentag ohne Veranstaltungen wünschen. Dies würde sich zum Beispiel als harte Anforderung umsetzen lassen, indem durch das AvailableLessonTimeConstraint (2.3.9), für alle Lessons der Semestergruppe nur Timeslots vorgegeben werden, welche nicht an dem Tag liegen, der frei bleiben soll. Durch das Constraint an dieser Stelle, soll ein alternativer Ansatz gezeigt werden, der die Anforderung nicht garantiert und dadurch nicht beliebig große Auswirkungen auf den restlichen Stundenplan zulässt. Gegebenenfalls wäre es durch den freien Tag der Semestergruppe nicht möglich, einen Stundenplan zu finden oder es könnten unverhältnismäßig viele optionale Constraints nicht erfüllt werden. Durch die Angabe als optionales Constraint lässt sich dessen Wichtigkeit durch die Gewichtung in Verhältnis zu den anderen optionalen Anforderungen setzen.

Für jede Lesson, welche an dem Tag stattfindet, soll der Wert des Objectives, um die Gewichtung des FreeDaySemesterGroupConstraints erhöht werden. Es werden nur die Anzahl der Lessons betrachtet, nicht deren Länge. Jede Semestergruppe darf höchstens einen freien Tag wünschen. Die Anzahl der Lessons die am gewünschten freien Tag stattfinden, wird auf diese Weise möglichst reduziert. Durch das Verhältnis der Gewichtungen aller weichen Anforderungen, kann spezifiziert werden, wie viele Kompromisse gegebenenfalls eingegangen werden sollen um dies zu erfüllen.

Eine alternative Möglichkeit wäre es, die Timeslots zu zählen, die an dem Tag für die Semestergruppe belegt sind.

Alternativ wäre es auch möglich, den Wert der Objective Function nur einmal zu erhöhen, sobald eine oder mehrere Lessons am gewünschten freien Tag stattfinden. Wenn es dann nicht möglich wäre, dass der Tag tatsächlich völlig frei bleibt, würden beliebig viele Veranstaltungen an diesem Tag stattfinden können, da es keine weitere Auswirkungen auf das Objective gibt. Allerdings kann angenommen werden, dass es den Studierenden schon helfen kann, wenn am gewünschten Tag nur eine oder möglichst wenige Veranstaltungen stattfinden, um ihre Termine oder Verpflichtungen trotzdem wahrnehmen zu können, wegen denen sie den freien Tag gewünscht haben. So bildet die erstgenannte Lösung ein Kompromiss zwischen allen Beteiligten des Stundenplans.

Für jede Lesson einer Semestergruppe, soll entweder das Gewicht des Constraints oder 0, dem Objective hinzugefügt werden, je nachdem ob sie am gewünschten freien Tag stattfindet. Für jede Semestergruppe bei der ein freier Tag gewünscht ist, gibt es daher für jede Lesson, an der die Gruppe teilnimmt, eine Variable c die Bestandteil der Optimierungsfunktion ist. Die Variablen müssen dann jeweils mit dem Gewicht dieses Constraints multipliziert werden. Wie die Belegung zustande kommt, ist in Formel 30 gezeigt.

$$\begin{gathered}\forall sg \forall d \forall l \forall c : \\ IstSemestergruppe(sg) \ \wedge IstWunschtag(d, sg) \wedge IstLessonVon(l, sg) \wedge \\ IstOVVariable(c, sg, l) \wedge \\ (FindetStatt(l, d) \implies c = 1) \wedge \\ (\neg FindetStatt(l, d) \implies c = 0)\end{gathered}$$

Formel 30: FreeDaySemesterGroupFormel

Die Prädikate der Formel 30:

- $IstSemestergruppe(sg)$: Bei sg handelt es sich um eine Semestergruppe die sich einen freien Wochentag gewünscht hat.
- $IstWunschtag(d, sg)$: Die Semestergruppe sg wünscht einen freien Tag am Wochentag d.
- $IstLessonVon(l, sg)$: l ist eine Lesson und die Semestergruppe sg nimmt daran teil.
- $IstOVVariable(c, sg, l)$: c ist die Variable in der Optimierungsfunktion, die für das Stattfinden der Lesson l für die Semestergruppe sg an deren freien Wunschtag steht.
- $FindetStatt(l, d)$: Die Lesson l findet am Wochentag d statt.

Die Details der Implementierung sind in Abschnitt 3.5.5 CountLessonsOnFreeDaySemesterGroup zu finden.

2.5 Alternative Möglichkeiten den Stundenplan zu optimieren

Es sind noch andere Möglichkeiten zur Optimierung des Stundenplans für die Beteiligten denkbar. Zum Beispiel wäre es denkbar, ein Constraint einzuführen, dass die Stunden für die Semestergruppen möglichst gleichmäßig auf die ganze Woche verteilt. Indirekt wird dies jedoch schon durch die Constraints erreicht, die zu viele, mit Lessons belegte Timeslots pro Tag verhindern (MaxLessonsPerDaySemesterGroupConstraint). Auch die optionalen Constraints, die Lessons in den späten oder frühen Timeslots eines Tages verhindern, sorgen indirekt für eine gleichmäßige Verteilung über die Woche hinweg.

Es wäre außerdem möglich, die Beschränkung für maximale Timeslots pro Tag optional, also als Teil der weichen Anforderungen zu gestalten. Dies hätte den Vorteil, dass niedrigere Zahlen gewählt werden könnten, da in Kauf genommen werden kann, dass das Constraint nicht immer eingehalten werden kann. Dies würde automatisch zu einer gleichmäßigen Verteilung über die Woche führen, wenn dies möglich ist. Nachteil wäre, dass Tage mit sehr vielen belegten Timeslots möglich wären.

2.6 Umsetzung als Datenbankmodell

Um für die Suche nach einem Stundenplan ausgelesen werden zu können, sollen sämtliche Stundenplandaten an einem Ort gesammelt werden. Dort müssen alle Objekte des Stundenplans, deren Eigenschaften und gegebenenfalls Vorgaben für Constraints, enthalten sein.

Die Speicherung soll in Form einer SQLite[9] Datenbank stattfinden. Dies ist eine sehr leichtgewichtige Datenbank, welche die Speicherung in einer einzigen Datei erlaubt. Des Weiteren gibt es Schnittstellen für alle gängigen Programmiersprachen und die Daten können durch Objekt-Relationales-Mapping durch das zu implementierende Python Programm ausgelesen werden.

2.6.1 Datenbankmodell

In diesem Abschnitt wird gezeigt, wie jedes Objekt der Stundenplandaten und deren Verknüpfungen untereinander, im Datenbankmodell umgesetzt ist. Die Benennung erfolgt im snake_case, kleingeschrieben, mit Unterstrichen. Ein doppelter Unterstrich bei Tabellenbezeichnungen weist auf eine Assoziationstabelle als Implementierung einer m zu n Beziehung anderer Entitäten hin. So Verknüpft zum Beispiel die Tabelle *lesson___teacher*, die beiden Entitäten *lesson* und *teacher* und stellt die Information bereit, welche Lehrenden einer Lesson zugeordnet sind.

Die Primärschlüssel der Stundenplanobjekte Timeslot und Raum, werden zusätzlich zur Datenbank auch für die Implementierung der Stundenplansuche benötigt. Bei den Primärschlüsseln der Timeslots ist es außerdem notwendig, dass diese mit 1 für den ersten Timeslot der Woche beginnen und von da an zählend aufsteigen.

In SQLite Datenbanken werden boolsche Werte als Zahl (Wahr = 1, Falsch = 0) angegeben.

[9] sqlite.org

Timeslot
Jeder Timeslot des Stundenplans, wird als Entität in einer eigenen Tabelle dargestellt. Die Struktur dieser Tabelle ist in Abbildung 1 zu sehen. Die Bedeutung der einzelnen Spalten ist anschließend aufgeführt.

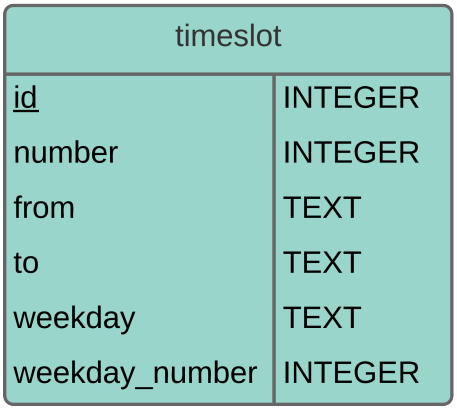

Abbildung 1: Die Tabelle `timeslot`

Die Bedeutung der einzelnen Spalten der Timeslottabelle:

- *id*: Der Primärschlüssel des Timeslots. Muss mit 1 für den ersten Timeslot der Woche beginnen und dann zählend aufsteigend.
- *number*: Die Nummer des Timeslots an einem Tag, jeder erste Timeslots einer der Wochentage hat die Nummer 1.
- *from*: Gibt die Uhrzeit, zudem der Timeslot beginnt, als Text an. Nötig für die spätere Ausgabe des Stundenplans.
- *to*: Gibt die Uhrzeit, zudem der Timeslot endet, als Text an. Nötig für die spätere Ausgabe des Stundenplans.
- *weekday*: Gibt den Wochentag, an dem sich der Timeslot befindet, als Abkürzung an. (MO, TU, WE, TH, FR)
- *weekday_number*: Die Nummer des Wochentags, an dem sich der Timeslot befindet. Beginnend mit 1 für Montag.

In Tabelle 3 sieht man als Beispiel, den Inhalt der `timeslot` Tabelle für den Stundenplan der TH-Lübeck, für die Tage Montag und Dienstag.

Raum
Jeder Raum ist als Tupel in der Tabelle *room* enthalten. Der Aufbau der Tabelle ist in Abbildung 2 zu sehen. Vier Beispieldatensätze sind in Tabelle 4 enthalten.

Die Bedeutung der einzelnen Spalten der Raumtabelle:

- *id*: Der Primärschlüssel des Raumes.
- *name*: Der Name in Form einer Abkürzung des Raumes. Wird für die spätere Ausgabe des Stundenplans benötigt.

id	number	from	to	weekday	weekday_number
1	1	08:15	09:45	MO	1
2	2	10:00	11:30	MO	1
3	3	12:00	13:30	MO	1
4	4	14:30	16:00	MO	1
5	5	16:15	17:45	MO	1
6	6	18:00	19:30	MO	1
7	1	08:15	09:45	TU	2
8	2	10:00	11:30	TU	2
9	3	12:00	13:30	TU	2
10	4	14:30	16:00	TU	2
11	5	16:15	17:45	TU	2
12	6	18:00	19:30	TU	2

Tabelle 3: Beispielansicht der `timeslot` Tabelle

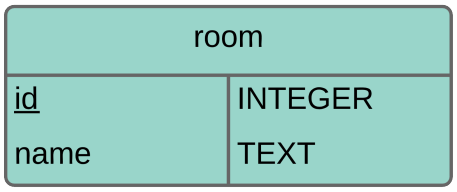

Abbildung 2: Die Tabelle `room`

id	name
1	AM 2
2	1-1.10
3	2-0.10
4	18-1.02

Tabelle 4: Beispielansicht der `room` Tabelle

Semestergruppe

Jede Semestergruppe ist als Tupel in der Tabelle *semester_group* enthalten. Der Aufbau der Tabelle ist in Abbildung 3 zu sehen. Einige Beispieldatensätze sind in Tabelle 5 abgegeben.

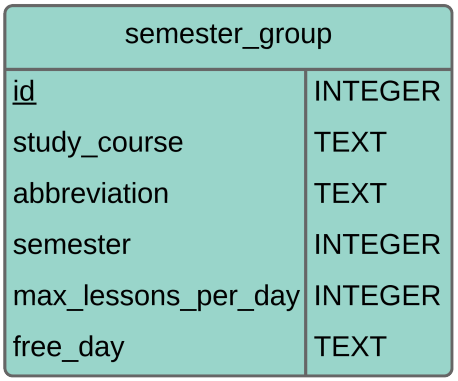

Abbildung 3: Die Tabelle `semester_group`

Die Bedeutung der einzelnen Spalten der Semestergruppentabelle:

- *id*: Der Primärschlüssel der Semestergruppe.
- *study_course*: Der Name des Studiengangs in dem die Semestergruppe studiert. Wird nur für die Ausgabe des Stundenplans benötigt.
- *abbreviation*: Abkürzung der Semestergruppe. Wird nur für die Ausgabe benötigt.
- *semester*: Das Semester der Semestergruppe. Wird nur für die Ausgabe benötigt.
- *max_lessons_per_day*: Gibt an, wie viele Timeslots an einem Tag, maximal mit Lessons dieser Semestergruppe belegt sein dürfen. Angabe für das MaxLessonsPerDaySemesterGroupConstraint (2.3.17).
- *free_day*: Angabe für den gewünschten freien Tag für das optionale FreeDaySemesterGroupConstraint (2.4.6). Angabe als englisches Wochentagskürzel, wie es auch bei den Timeslots angegeben ist. `null`, wenn kein freier Tag gewünscht ist.

id	sutdy_course	abbreviation	semester	max_lessons_per_day	free_day
1	Informatik/Softwaretechnik	INF 1	1	5	*null*
2	Informatik/Softwaretechnik	INF 3	3	5	*null*
3	Elektrotechnik - ESA	ESA 1	1	4	*null*
4	Elektrotechnik - EKS	EKS 1	1	5	WE

Tabelle 5: Beispielansicht der `semester_group` Tabelle

Lehrende

Jeder Lehrende ist als Tupel in der Tabelle *teacher* enthalten. Der Aufbau der Tabelle ist in Abbildung 4 gezeigt. Die Angaben zu Name und Abkürzung sind nur für die Ausgabe des Stundenplans relevant. Sie sind daher zur besseren Übersicht nicht in der Tabelle 6, mit Beispieldaten einer *teacher* Tabelle, enthalten.

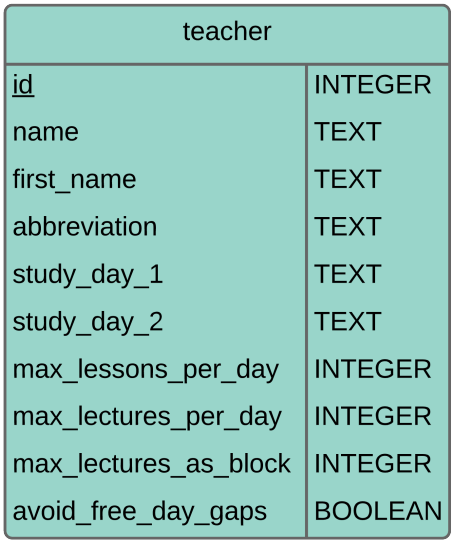

Abbildung 4: Die Tabelle `teacher`

Die Bedeutung der einzelnen Spalten der Lehrendentabelle:

- *id*: Der Primärschlüssel des Lehrenden.
- *name*: Der Nachname des Lehrenden. Wird lediglich für die Ausgabe der Lehrenden-Stundenpläne benötigt.
- *first_name*: Der Vorname des Lehrenden. Wird lediglich für die Ausgabe der Lehrenden-Stundenpläne benötigt.
- *abbreviation*: Die Abkürzung des Lehrenden. Sollte eindeutig sein. Wird lediglich für die Ausgabe der Stundenpläne benötigt.
- *study_day_1*: Gibt die Erstwahl für den Studientag des Lehrenden als Wochentagskürzel an, StudyDayConstraint (2.3.5). Steht dem Lehrenden kein Studientag zu, wird ein *null* Wert eingefügt. Mögliche Belegungen sind: MO, TU, WE, TH, FR, *null*
- *study_day_2*: Gibt die Zweitwahl für den Studientag des Lehrenden an. Siehe Spalte *study_day_1*.
- *max_lessons_per_day*: Gibt an, wie viele Timeslots mit Veranstaltungen für den Lehrenden, pro Tag maximal belegt sein dürfen. Angabe für das MaxLessonsPerDayTeacherConstraint (2.3.14).
- *max_lectures_per_day*: Gibt an, wie viele Timeslots mit Veranstaltungen, die als Vorlesung markiert sind, für den Lehrenden pro Tag maximal belegt sein dürfen. Angabe für das MaxLecturesPerDayTeacherConstraint (2.3.15).
- *max_lectures_as_block*: Gibt an wie viele Timeslots hintereinander, für den Lehrenden, maximal mit Vorlesungen belegt sein dürfen. Angabe für das MaxLecturesAsBlockTeacherConstraint (2.3.16).
- *avoid_free_day_gaps*: Gibt an, ob für den Lehrenden als optionales AvoidGapBetweenDaysTeacherConstraint (2.4.5), freie Tage zwischen nicht freien Tagen vermieden werden sollen.

id	*	**study_day_1**	**study_day_2**	**max_lessons_per_day**	**max_lectures_per_day**	**max_lectures_as_block**	**avoid_free_day_gaps**
1		MO	FR	5	3	2	0
2		FR	TH	4	3	2	1
3		TU	TU	5	3	2	0
4		*null*	*null*	6	3	3	0

Tabelle 6: Beispielansicht der `teacher` Tabelle. Die Spalten mit den Informationen zu Namen, wurden zur besseren Übersicht weggelassen.

Kurs

Jeder Kurs, wird als Tupel in der Tabelle *course* dargestellt. Die Struktur der Tabelle ist in Abbildung 5 zu sehen. Die Information des Kursnamen, dessen Abkürzung und Kurstyp, werden nur für die spätere Ausgabe des Stundenplans benötigt und haben für die

eigentliche Stundenplansuche und die Anforderungen keine Bedeutung. Die Informationen zu den Lessons, die zu einem Kurs gehören, Lehrende welche die Lessons unterrichten, Semestergruppen die an einem Kurs teilnehmen und Räume in denen ein Kurs stattfinden kann, sind durch Assoziationstabellen beziehungsweise Fremdschlüssel der anderen Entitäten angegeben. Siehe dazu Absatz Enitäts-Beziehungen. Beispielinhalte der Kurstabelle sind in Tabelle 7 aufgeführt.

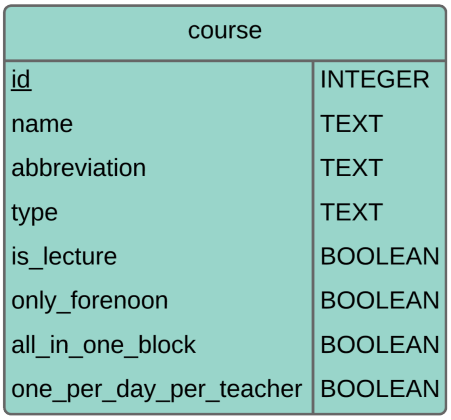

Abbildung 5: Die Tabelle `course`

Die Bedeutung der einzelnen Spalten der Kurstabelle:

- *id*: Der Primärschlüssel der Lesson.
- *name*: Enthält den Namen des Kurses. Dieser wird derzeit nicht in dem, als Excel Datei exportierten, Stundenplan verwendet, da dort nur die Abkürzungen enthalten sind. Sie sind jedoch in der Konsolenausgabe des Stundenplans aufgeführt.
- *abbreviation*: Die Abkürzung des Kurses. Gibt den Kurs in der zu exportierenden Excel Datei an.
- *type*: Der Kurstyp. Wird ausschließlich für die Ausgabe benötigt. Dort sind die Lessons einer Semestergruppe noch einmal in Kurstypen gruppiert. Übliche Kurstypen in der Stundenplanvorlage, sind Vorlesung, Übung, Praktika, WPF und Projekt.
- *is_lecture*: Gibt an, ob es sich um eine Vorlesung handelt. Unabhängig vom Kurstyp. Diese Angabe wird für die Constraints MaxLecturesPerDayTeacherConstraint (2.3.15) und MaxLecturesAsBlockTeacherConstraint (2.3.16) benötigt.
- *only_forenoon*: Angabe, ob Lessons des Kurses nur Vormittags stattfinden sollen. OnlyForenoonConstraint (2.3.8)
- *all_in_one_block*: Gibt an, ob alles Lessons des Kurses als Block stattfinden sollen. CourseAllInOneBlockConstraint (2.3.11)
- *one_per_day_per_teacher*: Angabe für das OneCoursePerDayTeacherConstraint (2.3.13)

id	*	abbre-viation	type	is_ lecture	only_ forenoon	all_in_ one_block	one_per _day_per _teacher
1		MA I	Vorlesung	1	1	0	0
2		MA I Ü	Übung	0	0	0	0
3		Prog I Pr	Praktikum	0	1	0	0
4		TI	WPF	1	0	0	0
5		BS	Vorlesung	1	0	1	0

Tabelle 7: Beispielansicht der course Tabelle
*Die Spalte mit den Kursnamen wurde zur besseren Übersicht weggelassen.

Lesson
Jede Lesson wird durch ein Tupel in der Tabelle *lesson* repräsentiert. Die Struktur der Tabelle ist in Abbildung 6 zu sehen. In Tabelle 8 sind Beispieldaten von vier Lessons zu sehen.

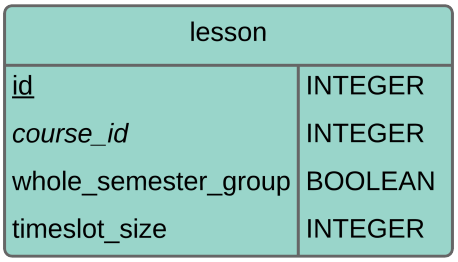

Abbildung 6: Die Tabelle *lesson*

Die Bedeutung der einzelnen Spalten der Lessontabelle:

- *id*: Der Primärschlüssel der Lesson.
- *course_id*: Der Fremdschlüssel des Kurses, zu dem die Lesson gehört.
- *whole_semester_group*: Gibt an, ob es sich um eine Lesson handelt, an der gesamte Semestergruppen teilnehmen. Siehe 2.3.4 PartSemesterGroupConstraint.
- *timeslot_size*: Gibt an, über wie viele aufeinanderfolgende Timeslots sich die Lesson erstreckt.

id	course_id	whole _semester_group	timslot _size
1	1	1	1
2	2	1	2
3	3	0	1
4	3	0	1

Tabelle 8: Beispielansicht der `lesson` Tabelle

Enitäts-Beziehungen

Eine Übersicht über die notwendigen und optionalen Beziehungen zwischen den Stundenplanobjekten, zeigt das ER-Diagramm mit Martin-Notation (Krähenfußnotation) in Abbildung 7. Die einzige 1:n Beziehung, besteht zwischen der Kurs- und der Lesson-Tabelle. Alle anderen Beziehungen sind m:n Beziehungen, für die eine Assoziationstabelle nötig ist. Der Name der Assoziationstabelle steht jeweils an der Kante zwischen den Entitäten. Im Folgendem, wird kurz auf jede Beziehung eingegangen und etwaige Besonderheiten erläutert.

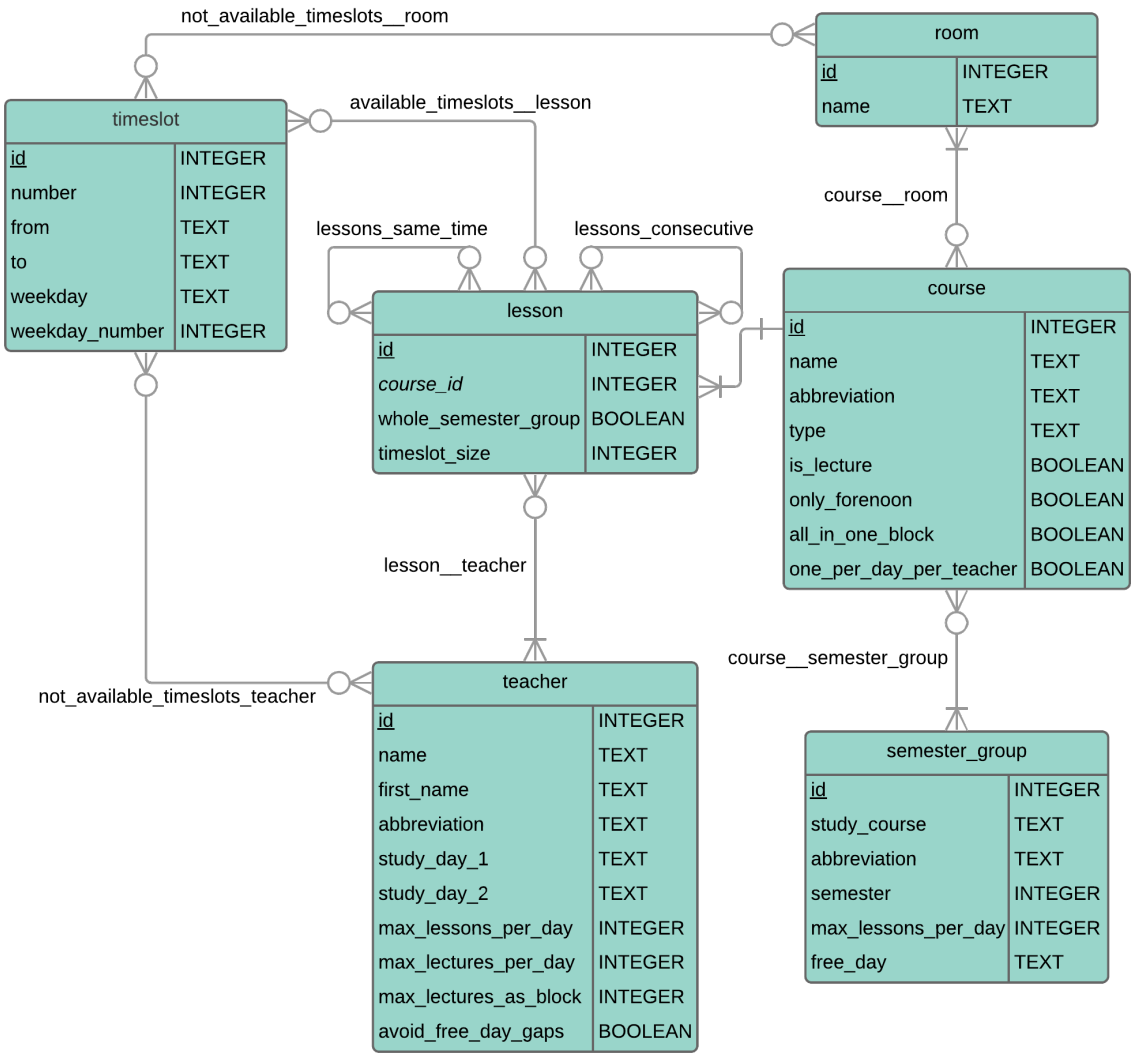

Abbildung 7: ER-Diagramm der Stundenplandatenbank

Grundlegende Beziehungen Diese Beziehungen sind für jeden Stundenplan unabdingbar. Sie zeichnen sich im Diagramm dadurch aus, dass für mindestens eine der Entitäten die Beziehung nicht-optional ist, also jeder Entität der einen Seite immer mindestens eine Entität der anderen Seite zugeordnet sein muss.[10]. Die Assoziationstabellen

[10]Sich am gegenüberliegenden Ende der Kante also kein Kreis befindet, welcher in der Krähenfußnotation darstellt, dass die Entität nicht zwingend an einer Beziehung des jeweiligen Typs beteiligt sein muss. Zum Beispiel benötigt jede Lesson mindestens einen Lehrenden.

enthalten immer zwei Spalten mit den Fremdschlüsseln der jeweiligen verknüpften Entitäten.

- Beziehung zwischen Kurs und Lesson (1:1..n): Jede Lesson enthält die Kurs id des Kurses zudem sie gehört als Fremdschlüssel. Jede Lesson gehört zu genau einem Kurs und jeder Kurs muss mindestens eine Lesson enthalten.
- course___semester_group (0..n:1..m): Gibt an welche Semestergruppen an einem Kurs teilnehmen. Jeder Kurs benötigt mindestens eine teilnehmende Semestergruppe.
- course___room (0..n:1..m): Gibt an, in welchen Räumen ein Kurs (dessen Lessons) stattfinden können. Jeder Kurs benötigt mindestens ein Raum in dem er stattfinden kann.
- lesson___teacher (0..n:1..m): Gibt die Lehrenden an, die eine Lesson unterrichten. Jede Lesson benötigt mindestens einen Lehrenden der sie unterrichtet.

Zusätzliche Beziehungen Diese Beziehungen sind jeweils für ein Constraint notwendig. Diese Constraints sind immer der Art, dass sie nicht grundlegend für alle Stundenplanobjekte gelten, sondern durch die Inhalte der Datenbank gesteuert werden. Die Kardinalitäten sind daher immer 0..n. Erklärt wird jeweils die Bedeutung der Assoziationstabelle.

- not_available_timeslots___teacher: Liefert die Datengrundlage für das NotAvailableTeacherTimeConstraint (2.3.7). Alle Lessons des jeweiligen Lehrenden werden nicht zu einem Timeslot stattfinden, mit dem der Lehrende über diese Beziehung verknüpft ist.
- not_available_timeslots___room: Liefert die Datengrundlage für das NotAvailableRoomTimeConstraint (2.3.6). Alle Lessons, die im jeweiligen Raum stattfinden, werden nicht zu einem Timeslot stattfinden, mit der der Raum über diese Beziehung verknüpft ist.
- available_timeslots___lesson: Datengrundlage für das AvailableLessonTimeConstraint (2.3.9). Wenn eine Lesson über diese Beziehung mit mindestens einem Timeslot verknüpft ist, kann diese nur noch zu Timeslots stattfinden, mit denen sie so verknüpft ist.
- lessons_same_time: Datengrundlage für das LessonsAtSameTimeConstraint (2.3.10). Verknüpft mindestens zwei Lessons zu einer Menge von Lessons, die zeitgleich anfangen müssen. Die Fremdschlüssel müssen dabei zwingend auf folgende Weise eingetragen sein. Eine beliebige Lesson (beziehungsweise deren ID) der Menge zeitgleich stattfindender Lessons, steht immer in der ersten Spalte der Assoziationstabelle. In allen Tupeln mit dieser Lesson, steht eine der anderen Lessons der Menge, in der zweiten Spalte. Also Beziehungen der Form A->B, A->C A->D geben an, dass die Lessons-Menge A,B,C,D gleichzeitig anfangen müssen. Verkettete Angaben der Menge (z.B.: A->B, B->C, C->D) werden nicht unterstützt.

- lessons_consecutive: Datengrundlage für das ConsecutiveLessonsConstraint (2.3.19). In der ersten Spalte, wird jeweils die Anfangslesson angegeben, in der zweiten Spalte die Folgelesson. Jede Lesson kann sowohl mehrfach Anfangs- als auch Folgelesson sein.

3 Implementierung

In diesem Kapitel wird auf den Aufbau des Programms zur Stundenplangenerierung eingegangen. Es wird insbesondere erläutert wie jede einzelne Anforderung aus den Abschnitten 2.3 Harte Anforderungen und 2.4 Optionale Anforderungen, in Form von Constraints mittels des `CpModels` aus der OR-Tools Bibliothek umgesetzt wird.

Zunächst folgt ein Überblick über die Programmarchitektur und die verwendete Bibliothek OR-Tools.

3.1 Bezeichner und Bibliotheken

Die Bezeichner im erstellten Python Code halten sich an die lowerCamelCase Notation. Ausnahme bilden Variablen die direkt Attribute und Beziehungen aus dem Datenbankmodell widerspiegeln. Für diese werden die gleichen Bezeichner wie in der Datenbank und somit snake_case verwendet. So ist an jeder Stelle ersichtlich, ob es sich um Daten direkt aus dem Stundenplanmodell oder später erstelle Variablen handelt.

Für das Objekt-Relationale-Mapping zwischen der Datenbank mit den Stundenplaninhalten aus der SQLite Datenbank und den Python Klassen wird die Python Bibliothek SQLAlchemy[11] verwendet.

Zur Ausgabe des erstellten Stundenplans als Excel Datei wird die Bibliothek XlsxWriter[12] verwendet. Sie erlaubt das automatisierte Erstellen von Excel Dateien und Arbeitsblättern.

Zur Installation der genannten Tools siehe 3.7.1 Installation.

3.1.1 Die OR-Tools Bibliothek

> *Google Optimization Tools (a.k.a., OR-Tools) ist ein open-source, schnelles und portables Softwarepaket für das Lösen kombinatorische Optimierungsprobleme.*

(Übersetzt aus der Einleitung der GitHub Ressource von Googles OR-Tools [Goob])

Die Optimierungsprobleme können dabei sehr vielfältig sein. Neben den klassischen Problemen wie Scheduling, Aufgaben- oder Terminplanung oder Routenplanung, können zum Beispiel auch Puzzles wie das N-Damen-Problem gelöst werden. Das Vorgehen sieht dabei immer so aus, dass das Problem in Form eines Modells beschrieben werden muss. Dies erfolgt durch Variablen und Constraints. Durch einen Solver kann dann versucht werden Erfüllungen der Constraints und somit einen Zustand zu finden, der eine Lösung darstellt.

Neben dem Constraint-Programming Solver, der hier hier für die Stundenplansuche verwendet werden wird, bietet OR-Tools noch Solver für Linear-Programming, Wrapper für Integer- und Mixed-Integer-Programming, sowie Tools für Behälter-, Routen- und

[11] www.sqlalchemy.org
[12] www.github.com/jmcnamara/XlsxWriter

Graphenprobleme. Zusätzlich zur Sprache C++ in der OR-Tools geschrieben ist, existieren Wrapper für C#, Java und Python.

Für die Stundenplansuche wird lediglich der Constraint-Programming Teil der Bibliothek verwendet. Nützliche einführende Informationen für die Implementierung, sind vor allem auf Googles eigener Website [Gooc] und der zugehörigen Python Referenz [Good] zu finden. Die GitHub Seite der Bibliothek bietet außerdem etwas weiterführendere Details [Gooe], sowie viele Beispielimplementierungen bekannter Probleme aus denen Strategien zur Implementierung der Constraints abgeleitet werden können.

Auf Googles Website zu OR-Tools, findet sich auch ein einführendes Kapitel zu Constraint-Programming im Allgemeinen. [Gooa]

Um ein Problem beschreiben zu können, gibt es eine Klasse `CpModel`. Einer Instanz dieser Klasse werden dann Integer und Boolean Variablen (genannt `IntVar` und `BoolVar`) hinzugefügt. Diesem Modell von Variablen können dann Constraints (Bedingungen) hinzugefügt werden, die Beschreiben, unter welchen Umständen und mit welchen Werten die Variablen belegt werden sollen. Auf diese Weise muss das gesamte Stundenplanmodell und alle gewünschten Anforderungen in dieses `CpModel` übersetzt werden.

OR-Tools bietet einen eigenen CP-SAT genannten Solver um das angegebene Modell zu lösen. Auf diese Weise wird nach einer Belegung aller Variablen gesucht, die mit den angegebenen Constraints kompatibel ist. Aus einigen der Variablen kann außerdem der Wert der Optimierungsfunktion, das Objective, errechnet werden. Die Optimierung ist jedoch optional, es kann auch nach einer beliebigen, validen Lösung gesucht werden.

Mit einer Instanz der Klasse `CpSolver` kann daher entweder eine beliebige Lösung, alle Lösungen, oder eine möglichst optimale Lösung (unter Angabe der Optimierungsfunktion) gesucht werden. Es kann dabei spezifiziert werden, wie lange gesucht werden soll. Außerdem können alle gefundene Lösungen zur Suchzeit ausgegeben werden.

3.2 Programmaufbau/Architektur

Das erstellte Programm zur Stundenplansuche lässt sich in folgende Teile aufteilen:

- **Stundenplandaten**: Implementierung der Stundenplanobjekte und Tabellen der Datenbank als Klassen und das Laden der Daten durch einen Object-Relational-Mapper.
- **Suche**: Der größte Teil umfasst die Dateien mit der Erstellung des `CpModels`, Funktionen zur Übersetzung der Stundenplandaten in das `CpModel` und die Ausführung der Suche. Dazu gehört außerdem ein Teil, der während der Suche gefundene Lösungen auf der Konsole ausgibt.
- **Datengenerierung**: Um Testdaten erstellen und in die SQLite Datenbank schreiben zu können, gibt es einen Teil, der vorgegebene Stundenplandaten generiert und in eine SQLite Datenbank Datei schreibt. Aus dieser werden bei der Suche die Daten ausgelesen.
- **Datenausgabe**: Umfasst die Ausgabe des besten, gefundenen Stundenplans in Form einer Excel Datei.

- **Validierung**: Um die Implementierung während der Entwicklung auf Fehler hin zu untersuchen, wurden Funktionen erstellt, die eine gefundene Stundenplanlösung mit den Vorgaben aus den Stundenplandaten vergleichen und die Einhaltung der Anforderungen überprüft.

Eine Übersicht der Python Dateien jedes Teils ist in Abbildung 8 zu sehen. Die Aufteilung ist jedoch nur zur Veranschaulichung. Auf eine Einteilung in Python Module wurde verzichtet. Mit *Stundenplanobjekte* sind die Klassen der Programmdaten bezeichnet, siehe Abschnitt 3.2.1 Stundenplandaten.

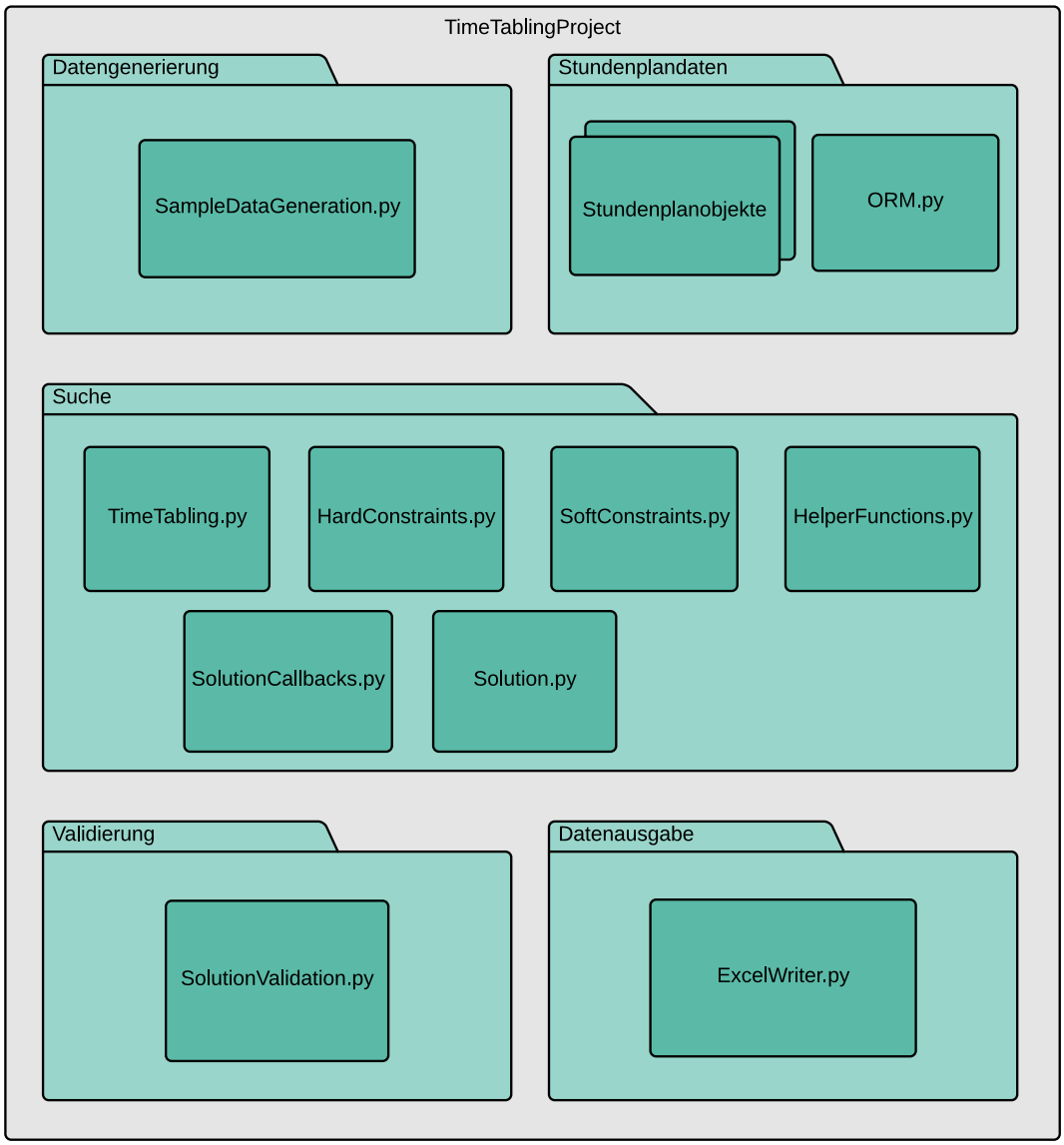

Abbildung 8: Übersicht der Python Dateien

Auf einen Programmteil, der die Stundenplandaten aus der Datenbank, welche der Fachbereich Elektrotechnik und Informatik verwendet, um den Stundenplan zu planen, in eine für das Programm benötigte Form zu übersetzen, wurde verzichtet. Es konnten für diese Arbeit leider keine Daten aus dieser Datenbank zur Verfügung gestellt werden.

3.2.1 Programmbestandteile

Es folgt eine kurze Erläuterung der einzelnen Programmteile die zuvor aufgelistet wurden.

Stundenplandaten

Die Stundenplanobjekte werden durch entsprechende Klassen dargestellt. Dies sind die Klassen `Timeslot`, `Room`, `Teacher`, `SemesterGroup`, `Course` und `Lesson`. Ihre Attribute entsprechen zunächst den Attributen und Beziehungen aus der Datenbank. Die aus den Stundenplandaten erstellten Instanzen, dienen jedoch auch dazu, Variablen des `CpModels` zwischenspeichern zu können. Diese Klassen befinden sich jeweils in einer Python Datei mit der Bezeichnung ***"Klassenname".py***. Sie sind einzeln in Abschnitt 3.3 Programmdaten aufgeführt.

Die Datei ***ORM.py*** enthält den Aufruf des Objekt-Relationalen-Mappers der SQLAlchemy Bibliothek und stellt die instanziierten Objekte der Stundenplandaten bereit. Zudem werden zwei Anpassungen an den eingelesenen `Lesson` Objekten durchgeführt (Details dazu im Abschnitt 3.3.5 Lesson).

Datengenerierung

Die Datei ***SampleDataGeneration.py*** enthält eine Hauptfunktion deren Zweck es ist, einen Beispieldatensatz zu erstellen und in die SQLite Datenbank zu schreiben. Es sind Hilfsfunktionen enthalten um die Daten anzulegen, sowie einzelne Funktionen die jeweils einen Datensatz anlegen. Es sind drei verschiedene Datensätze enthalten.

- Ein sehr kleiner Stundenplan um grundlegene Funktion des Programms testen zu können.
- Einen Datensatz der möglichst jedes Constraint und jede Besonderheit beziehungsweise Ausnahme mindestens einmal enthält und so alle Teile der Implementierung abzudecken.
- Einen großen Datensatz, der näher an einem tatsächlichen Hochschulstundenplan angelehnt ist.

Sie lassen sich jeweils durch Aufruf der Funktionen `generateSmallDataset`, `generateSmallFullTestDataset` und `generateBigDataset` die Daten generieren und in die SQLite Datenbank schreiben. Die Datenbankdatei wird jedoch nicht erstellt. Es gibt kein DDL Skript um die Datenbank selber zu erzeugen. Bei SQLite kann die vorliegende Datenbankdatei jedoch einfach kopiert werden, um eine weitere Instanz der Datenbank zu erhalten. Auf diese weise kann die Datenbankdatei dupliziert werden, um mehr als ein Stundenplanmodell gleichzeitig speichern zu können.

Suche

Für den Programmablauf sind einige Python Dateien vorgesehen. Die sind die Datei ***TimeTabling.py***, welche den Programmeinstieg, Erstellung des `CpModels`, Aufruf der Lösungssuche und der Ausgabe enthält.

In der Datei ***HardConstraints.py*** sind sämtliche Funktionen enthalten, die das Stundenplanmodell und die harten Anforderungen in das `CpModel` überführen.

Zur Übersetzung der weichen Anforderungen, dienen Funktionen in der Datei ***SoftConstraints.py***.

Um gefundene Lösungen zu speichern, weiterzuverarbeiten, auszugeben oder den Suchfortschritt auszugeben, dient die Datei ***SolutionCallbacks.py***. Sie enthält verschiedene Callbacks, die ausgeführt werden wenn eine Lösung gefunden wurde. Außerdem sind Funktionen enthalten die einen gefundenen Stundenplan auf der Konsole ausgeben. Es gibt ein Callback ***TimeTablePrinter***, das für die Ausgabe der gefundenen Lösungen auf der Console verwendet wird. Es bietet auch eine Option um den Fortschritt bei der Optimierung auszugeben. Dieses Callback wird bei der vorgesehenen Programmnutzung verwendet.

Zusätzlich gibt es eine Klasse `Solution` (in gleichnamiger Datei), die dazu dient, gefundene Lösungen in, von den Variablen des `CpModels` unabhängiger Form, speichern zu können.

Einige wenige Hilfsfunktionen die bei der Implementierung der Constraints verwendet werden, befinden sich in der Datei ***HelperFunctions.py***.

Validierung

Ein `SolutionValidator` Callback kann zur Validierung der Lösungen verwendet werden. Die Datei ***SolutionValidation.py*** bietet mit der Funktion `validateSolution` die Möglichkeit, die Einhaltung aller harten Anforderungen des Stundenplans in einem `Solution` Objekt, zu überprüfen. Diese Überprüfung diente jedoch nur dazu mögliche Fehler in der Implementierung zu finden. Bei einem normalen Programmdurchlauf findet sie nicht statt. Dies ist unnötig, da von einer korrekten Umsetzung der Constraints ausgegangen wird.

Datenausgabe

Der Code um aus einer Datenbanklösung eine Excel Datei zu generieren befindet sich in der Datei ***ExcelWriter.py***. Dort nimmt die Funktion `writeTimeTableExcelFile` ein `Solution` Objekt entgegen und erstellt eine Excel Datei mit zwei Arbeitsblättern. Eines enthält den Stundenplan aus Studierendensicht, aufgeteilt in die einzelnen Studiengänge und Semester. Der Stundenplan für die Lehrenden befindet sich im zweiten Arbeitsblatt.

3.3 Programmdaten

Mit Programmdaten ist die Repräsentation der Stundenplanobjekte und deren Pendants in der Datenbank, in Form von Python Klassen gemeint. Jedes der 6 Stundenplanobjekte hat eine eigene Python Klasse. Eine Übersicht über alle Klassen und deren Beziehungen ist in Abbildung 9, als UML-Klassendiagramm zu sehen. Da für das Laden und Speichern aus und in die Datenbank, die SQLAlchemy Bibliothek verwendet wird, erben all diese Klassen von der SQLAlchemy Klasse `Base`. Diese Klasse wird in der eigenen Datei ***Base.py*** erzeugt um von den Datenklassen geerbt werden zu können. Diese Datei enthält lediglich den nötigen Funktionsaufruf um die `Base` Klasse zu erstellen.

Die Klassen enthalten Attribute für jede Spalte der entsprechenden Tabelle im Datenbankmodell. Mit SQLAlchemy können diese Variablen als Typ `Column` zusammen mit

dem Datentyp der Spalte angegeben werden. Der Variablenname entspricht, wenn nicht extra angegeben, dem Spaltennamen. Außerdem werden die in der Datenbank vorhandenen Beziehungen übernommen. Die Navigierbarkeit und Bezeichner der Variablen für diese Beziehungen sind ebenfalls dem Klassendiagramm zu entnehmen.

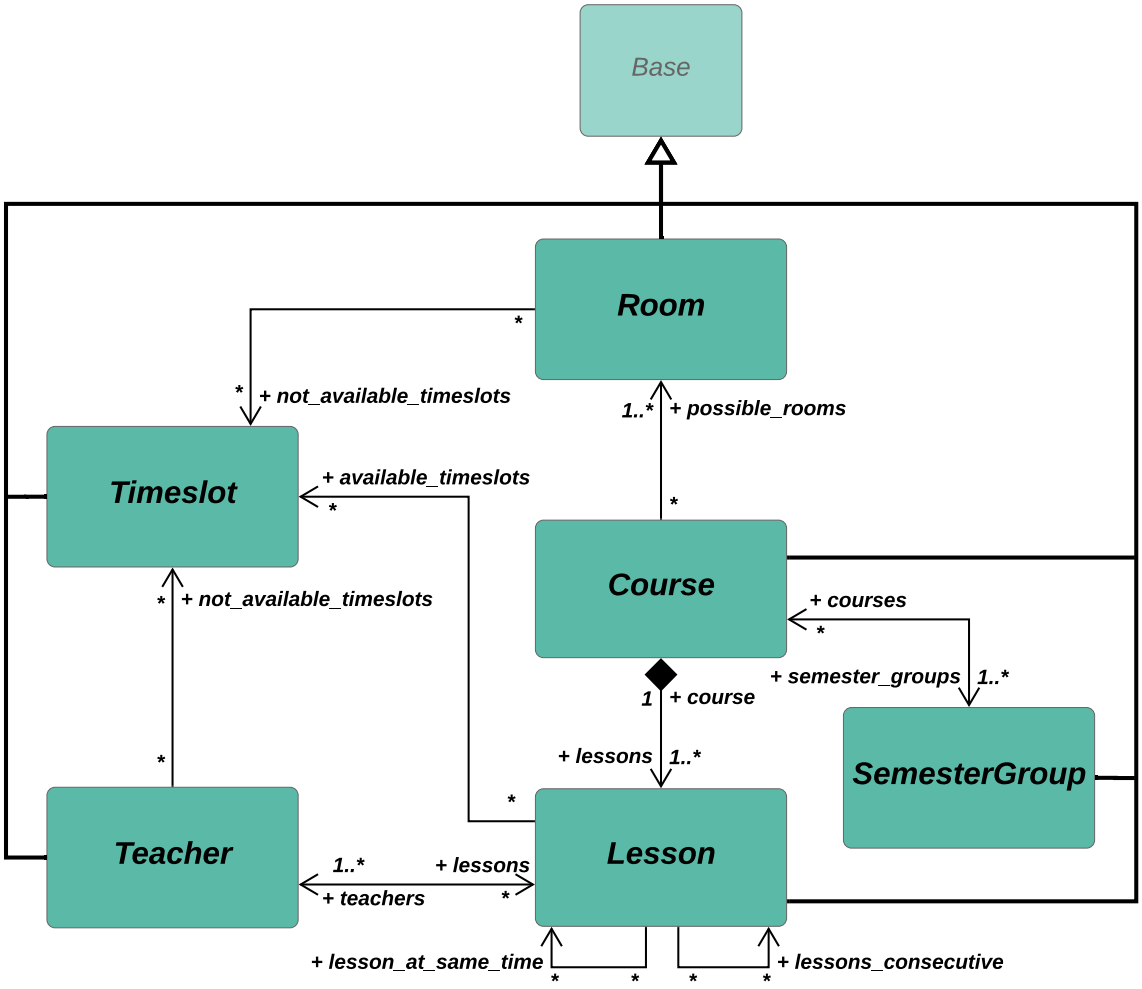

Abbildung 9: Klassendiagramm der Stundenplanobjekte

Im Folgendem werden die einzelnen Klassen, die die Stundenplanobjekte repräsentieren, kurz vorgestellt. Für die Angabe von Typen in den Abbildungen, wird die Form des in Python verwendeten „Type Hinting“, verwendet. Die Angabe des (in anderen Sprachen generischen) Typs einer Datenstruktur folgt dabei in eckigen Klammern. `list[Timeslot]` gibt zum Beispiel an, dass es sich im eine Liste mit `Timeslot` Objekten handelt.

3.3.1 Timeslot

Die Klasse `Timeslot` befindet sich in der Datei ***Timeslot.py***. Die Datei enthält zusätzlich String-Konstanten mit den Namen `MONDAY` bis `FRIDAY`. Mit den enthaltenen Zeichenketten (MO, TU, WE, TH, FR) können an verschiedenen Stellen, Wochentage angegeben werden. Verwendet werden diese zum Beispiel für die Variable `weekday` in jedem Timeslot Objekt, oder für die Angabe von gewünschten Studientagen bei Lehrenden.

Die eigentliche Klasse `Timeslot` enthält Variablen für alle Spalten der Timeslot Ta-

belle in der Datenbank. Die ID des Timeslots als `int`, die Angabe des Wochentages als eine des String-Konstanten, den Wochentag als Nummer, beginnend mit 1 für Montag, die Nummer des Timeslots an dem Tag an dem er stattfinden und die Uhrzeiten zu dem der Timeslot anfängt und aufhört. Letztere werden nur für die Ausgabe des Stundenplans verwendet. Die IDs der Timeslots werden jedoch direkt im `CpModel` verwendet um Timeslots anzugeben. Eine Übersicht über die Klasse ist in Abbildung 10 zu sehen.

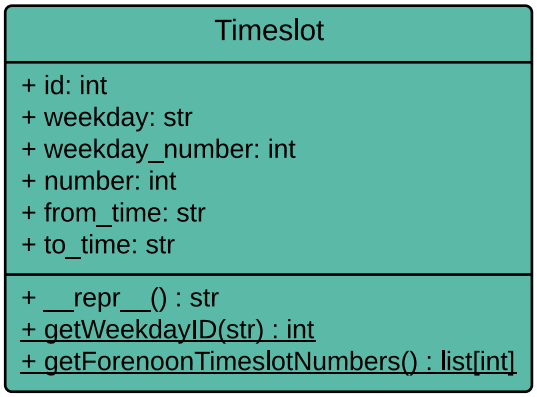

Abbildung 10: Aufbau der Klasse `Timeslot`

Neben einer Methode für die textuelle Repräsentation des Timeslots gibt es noch zwei statische Methoden:

- `getWeekdayID(weekday: str) -> int`: Die Methode liefert zu einer Wochentags-Konstanten die Nummer des Wochentages.
- `getForenoonTimeslotNumbers() -> list[int]`: Um abfragen zu können, welche Timeslots als Vormittags-Timeslots zu zählen sind, liefert die Methode eine Liste mit den Nummern dieser Timeslots. Es handelt sich dabei nicht um IDs sondern die Nummer der Timeslots an einem Tag. Die Liste $[1, 2, 3]$ gibt also an, dass die ersten drei Timeslots jeden Tages als Vormittags-Timeslots zu zählen sind. Diese Angabe wird für das OnlyForenoonConstraint (2.3.8) benötigt.

3.3.2 Room

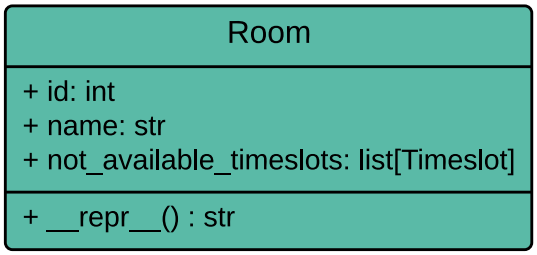

Abbildung 11: Aufbau der Klasse `Room`

Zur `Room` Klasse gehört lediglich die ID des Raumes und ein Name, der für die Stundenplanausgabe benötigt wird. Abbildung 11 zeigt eine Übersicht. Dazu kommt eine

Liste mit Timeslots, zu denen der Raum nicht verfügbar ist, NotAvailableRoomTimeConstraint (2.3.6). Um dafür die Beziehung mit den Timeslots herzustellen, wird die Angabe der Assoziationstabelle benötigt. Diese wird in derselben Datei als SQLAlchemy `Table` angegeben.

Die IDs der Räume werden im `CpModel` verwendet um einen Raum anzugeben.

3.3.3 Teacher

Die Klasse `Teacher` ist aufgrund der vielen Cosntraints, die sich direkt auf Lehrende beziehen, eine der Klassen mit dem meisten Inhalt. Eine Übersicht der Klasse zeigt die Abbildung 12. Die meisten Attribute entsprechen den jeweiligen Spalten der Datenbanktabelle *teacher*. Da die Variablen genau den Daten aus der Datenbank entsprechen, sind die Bezeichner zu den Spaltennamen identisch.

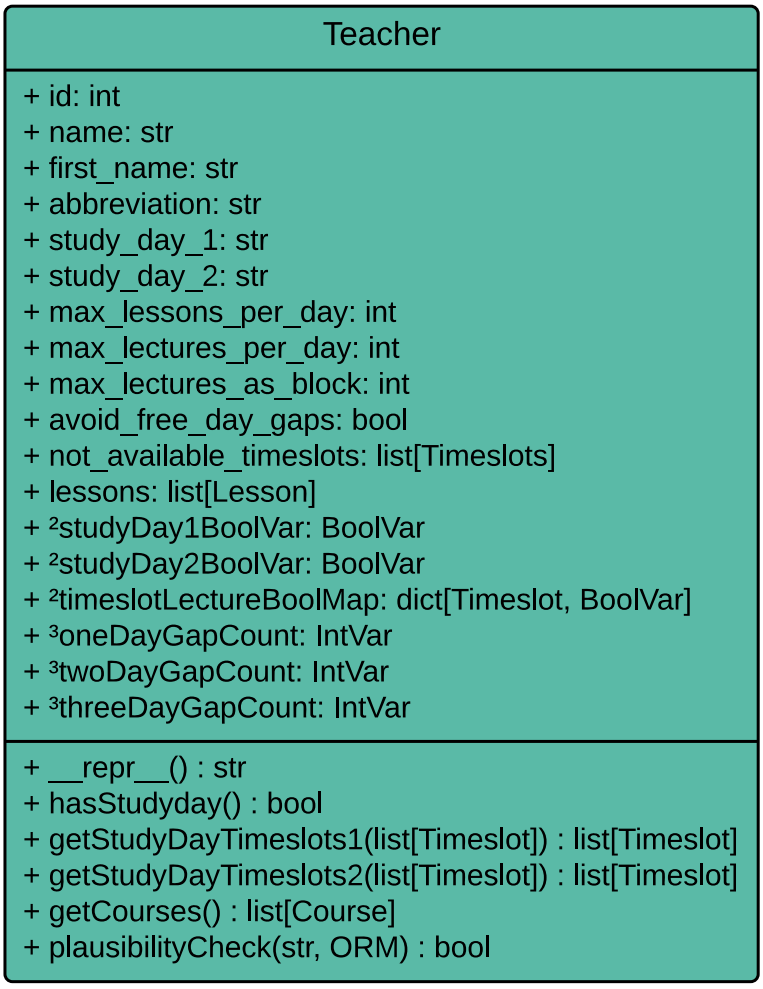

Abbildung 12: Aufbau der Klasse `Teacher`

- id, int: Primärschlüssel in der Datenbank, für die Stundenplansuche nicht benötigt.
- name, str: Nachname, nur für die Ausgabe des Stundenplans.
- first_name, str: Vorname, nur für die Ausgabe des Stundenplans.

- abbreviation, str: Abkürzung, möglichst eindeutig. Wird für die Ausgabe des Stundenplans für Studierende benötigt.
- study_day_1, str: Erstwahl des Studientages. Eine der Wochentags String Konstanten aus der ***Timeslot.py*** Datei.
- study_day_2, str: Zweitwahl des Studientages.
- max_lessons_per_day, int: Maximale Anzahl belegter Timeslots pro Wochentag. Datengrundlage für das MaxLessonsPerDayTeacherConstraint.
- max_lectures_per_day, int: Maximale Anzahl mit Vorlesungen belegter Timeslots pro Wochentag. Datengrundlage für das MaxLecturesPerDayTeacherConstraint.
- max_lectures_as_block, int: Maximale Anzahl mit Vorlesungen belegter Timeslots in Folge. Datengrundlage für das MaxLecturesAsBlockTeacherConstraints.
- avoid_free_day_gaps, bool: Gibt an, ob freie Zwischentage vermieden werden sollen. Grundlage für das AvoidGapBetweenDaysTeacherConstraint.

Zu diesen einfachen Attributen, kommt noch die Liste mit Timeslots zu denen der Lehrende nicht verfügbar ist (*not_available_timeslots*). Außerdem wird jedem *teacher* Objekt eine Liste *lessons* hinzugefügt, die alle Lessons enthält, die der Lehrende unterrichtet. Das Hinzufügen geschieht jedoch durch die Klasse `Lesson`, also von der anderen Seite der Relation aus.

Während der Erstellung des `CpModels` werden einige Variablen an einige `Teacher` Objekte angehängt. Dies sind Variablen des `CpModels` selber, die mehrfach verwendet werden und daher über die `Teacher` Objekte, zugreifbar sein sollen. Variablen, die durch Funktionen in der Datei ***HardConstraints.py*** hinzugefügt werden, sind in Abbildung 4 durch eine hochgestellte [2] markiert. Dies sind zum einen Variablen, die angeben, ob ein Studientag eingehalten wird. Sie werden nur bei Lehrenden angehängt denen auch ein Studientag zusteht. Zum anderen wird jedem Lehrenden ein Dictionary hinzugefügt, dass für jeden Timeslot des Stundenplans eine `BoolVar` enthält, die angibt ob zu dem Timeslot eine, als Vorlesung markierte, Lesson für den Lehrenden stattfindet.

Variablen der weichen Anforderungen, die durch Funktionen der Datei ***SoftConstraints.py*** hinzugefügt werden, sind durch eine hochgestellte [3] markiert. Bei Lehrenden für die das AvoidGapBetweenDaysTeacherConstraint (2.4.5) gilt, werden Variablen angefügt, welche Lücken verschiedener Größe zählen.

Die `Teacher` Klasse enthält einige Methoden, die bei der Implementierung von Constraints verwendet werden.

- `hasStudyday() -> bool`: Gibt `True` zurück, wenn der Lehrende einen Studientag hat. Dies wird ermittelt, indem festgestellt wird, dass beide *study_day_* Variablen einen Wert enthalten.

- `getStudyDayTimeslots1(timeslots) -> list[Timeslot]`: Gibt eine Liste mit allen Timeslots zurück, die zu dem Tag gehören, den der Lehrende mit seinem Erstwunsch als Studientag wünscht. Dafür benötigt die Methode eine Liste aller Timeslots. `getStudyDayTimeslots2(timeslots) -> list[Timeslot]` liefert entsprechend die Liste für den Zweitwunsch.

- `getCourses() -> list[Course]`: Liefert eine Liste aller Kurse an denen der Lehrende beteiligt ist. Es sind alle Kurse enthalten die mindestens eine Lesson beinhalten, die der Lehrende unterrichtet.

- `plausibilityCheck(orm, preString) -> bool`: Diese Methode führt für die Datengrundlage des Lehrenden, einen Plausibilitätscheck durch. Dies war zunächst gedacht um bei unübersichtlichen Teststundenplänen, Fehler zu finden, wenn diese ein nicht erfüllbares Modell ergaben. Gefunden werden Widersprüche wie zum Beispiel mehr Timeslots die der Lehrende unterrichten soll, als Timeslots zu denen der Lehrende verfügbar ist. Auch Constraints wie das MaxLecturesAsBlockTeacherConstraint werden überprüft. Es kann zum Beispiel vorkommen, dass angegeben ist, dass der Lehrende nur maximal zwei Timeslots in Folge Vorlesungen halten möchte, ihm jedoch eine Vorlesungslesson der Länge drei zugeteilt ist. Dies gibt zwangsläufig ein unerfüllbares Modell. Es kann jedoch auch false positives geben, da eine genaue Abschätzung teilweise schwierig ist. Zum Beispiel bei vielen Lessons die durch das LessonsAtSameTimeConstraint gleichzeitig im Stundenplan eingetragen sind. Andersherum beweist ein Bestehen des Tests nicht zwangsläufig ein erfüllbares Modell. Die Methode ist jedoch nützlich um Fehler in den Stundenplandaten zu finden.

- `__repr__() -> str`: Liefert eine textuelle Representation des Lehrenden mit ID, Abkürzung und den Studientagswünschen.

3.3.4 SemesterGroup

Die Klasse `SemesterGroup` ist in Abbildung 13 dargestellt. Die nicht markierten Attribute, entsprechen den Attributen aus der Datenbank. Die mit einer hochgestellten [3] markierten Variablen, werden für die Implementierung der weichen Anforderungen benötigt und durch Funktionen der Datei ***SoftConstraints.py*** hinzugefügt.

Attribute aus der Datenbank:

- id, int: Primärschlüssel in der Datenbank. Für die Suche nicht verwendet.

- study_course, str: Name des Studiengangs. Wird nur für die Ausgabe des Stundenplans verwendet.

- abbreviation, str: Abkürzung der Semestergruppe. Zum Beispiel mit Semester oder Vertiefung. Wird in der Ausgabe verwendet.

- semester, int: Nummer des Semesters der Gruppe. Zum Beispiel 1 für eine Erstsemestergruppe.

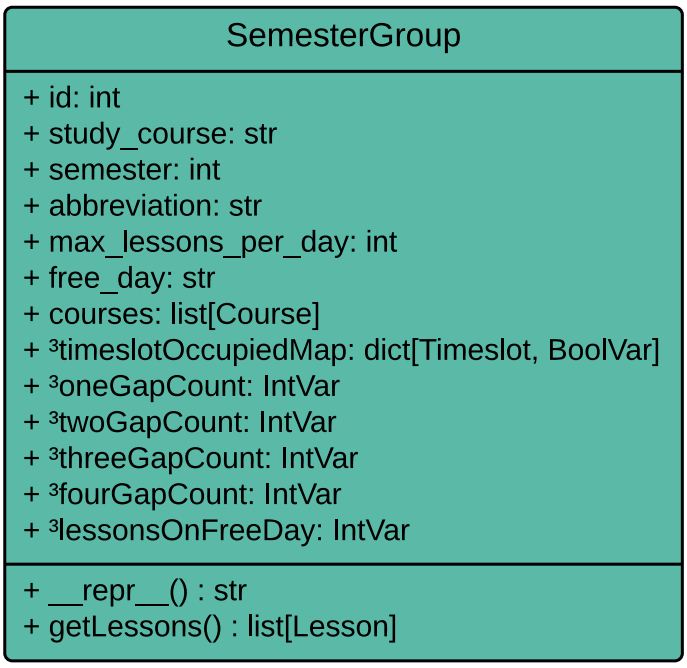

Abbildung 13: Aufbau der Klasse `SemesterGroup`

- max_lessons_per_day, int: Maximale Anzahl belegter Timeslots pro Tag. Grundlage für das MaxLessonsPerDaySemesterGroupConstraint.
- free_day, str: Optionale Angabe eines freien Wochentages. Grundlage für das FreeDaySemesterGroupConstraint. Es werden die Konstanten aus der Timeslot Datei verwendet. Zum Beispiel WE für einen gewünschten freien Mittwoch.
- courses, list[Course]: Liste aller Kurse, an denen die Gruppe teilnimmt. In der Datenbank durch eine Assoziationstabelle angegeben. Das Hinzufügen dieser Liste geschieht in der `Course` Klasse.

Die für jede Semestergruppe später hinzugefügte Variable *`timeslotOccupiedMap`* enthält ein Dictionary dass für jeden Timeslot eine `BoolVar` Varible des `CpModels` enthält, die angibt, ob für die Semestergruppe zu dem Timeslot eine Veranstaltung stattfindet. Die anderen Variablen zählen Lücken zwischen Veranstaltungen für das AvoidGapBetweenLessonsSemesterGroupConstraint (2.4.4), beziehungsweise Lessons an einem gewünschten freien Tag, falls das FreeDaySemesterGroupConstraint (2.4.6) für die Gruppe aktiviert ist.

Es gibt außerdem eine Methode für eine textuelle Repräsentation der Semestergruppe. Die Methode `getLessons() -> list[Lesson]` liefert eine Liste mit allen `Lesson` Objekten aller Kurse an denen die Semestergruppe teilnimmt.

3.3.5 Lesson

Den Inhalt der Klasse `Lesson`, zeigt die Abbildung 14. Es sind die Variablen des Datenmodells enthalten, die direkt aus der Datenbank geladen werden. Nach dem Laden aller Objekte aus der Datenbank, führt die Datei ***ORM.py***, zwei Veränderung an jedem

`Lesson` Objekt aus. Die betreffenden Variablen sind mit einer hochgestellten [1] markiert. Des Weiteren fügen Funktionen der Datei ***HardConstraints.py***, noch zusätzliche Variablen hinzu. Dies geschieht während die harten Anforderungen dem `CpModel` hinzugefügt werden. Diese Variablen sind mit einer hochgestellten [2] versehen.

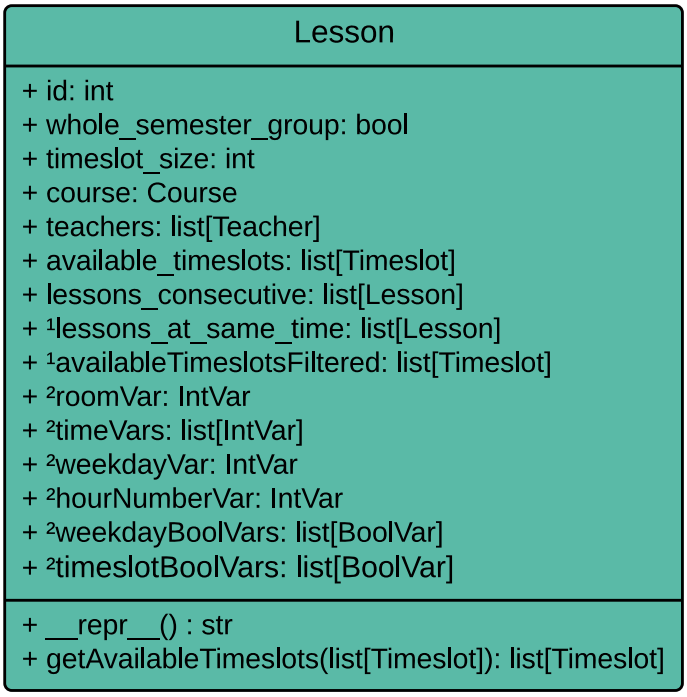

Abbildung 14: Aufbau der Klasse `Lesson`

Attribute aus der Datenbank:

- *`id, int`*: Primärschlüssel in der Datenbank. Für die Suche nicht verwendet.
- *`whole_semester_group, bool`*: Gibt an, ob an dieser Lesson nur ganze Semestergruppen teilnehmen. Benötigt für das (PartSemesterGroupConstraint (2.3.4).
- *`timeslot_size, int`*: Gibt an, wie viele Timeslots die Lesson belegt.
- *`course, Course`*: Das Kurs Objekt zu dem die Lesson gehört.
- *`teachers, list[Teacher]`*: Eine Liste der Lehrenden, welche die Lesson unterrichten.
- *`available_timeslots, list[Timeslot]`*: Liste von Timeslots zu denen die Lesson stattfinden darf. Siehe 2.3.9 AvailableLessonTimeConstraint.
- *`lessons_consecutive, list[Lesson]`*: Angabe von anderen Lessons, die immer im Anschluss an die Lesson stattfinden sollen. (ConsecutiveLessonsConstraint (2.3.19))
- *`lessons_at_same_time, list[Lesson]`*: Angabe von anderen Lessons, die immer gleichzeitig im Stundenplan eingetragen werden sollen. Benötigt für das LessonsAtSameTimeConstraint (2.3.10).

[1]*lessons_at_same_time*:
In der Datenbank erfolgt die Angabe der gleichzeitig einzutragenden Lessons, immer nur für eine der beteiligten Lessons. Sollen die Lessons A, B und C zeitgleich im Stundenplan stattfinden, werden zum Beispiel, bei der Lesson A, die Lessons B und C eingetragen. Nach dem Laden der Daten, fügt die Datei ***ORM.py*** die Daten noch bei den anderen Lessons in die Liste *lessons_at_same_time* ein. Also für obiges Beispiel, die Lessons A und C, bei der Lesson B und die Lessons A und B, bei der Lesson C. Dies erleichtert die Implementierung des LessonsAtSameTimeConstraints.

`getAvailableTimeslots()` und [1]*availableTimeslotsFiltered*:
Neben der Liste der Timeslots, zu denen eine Lesson stattfinden darf, kann die Menge der möglichen Timeslots gegebenenfalls noch weiter eingeschränkt werden. Darf die Lesson zum Beispiel nur vormittags stattfinden, kommen auch nur Timeslots in Frage, die vormittags stattfinden. Haben die Lehrenden, welche die Lesson geben, Timeslots, zu denen sie nicht verfügbar sind, schränkt dies die Menge noch weiter ein. Die Methode `getAvailableTimeslots(list[Timeslot) -> list[Timeslot]` tut genau dies. Es wird die Schnittmenge, der als verfügbar angegebenen Timeslots für die Lesson (für den Fall, dass diese Liste nicht leer ist), der verfügbaren Timeslots jedes Lehrenden der Lesson und gegebenenfalls der Vormittags-Timeslots gebildet. Als Parameter nimmt die Methode eine Liste aller Timeslots des Stundenplans entgegen.

Nach dem Laden aller Daten, wird diese Methode in der Datei ***ORM.py***, für jede Lesson aufgerufen und das Ergebnis in der Variable *availableTimeslotsFiltered* abgespeichert. Damit steht die Liste bei der Implementierung der harten Anforderungen zur Verfügung, ohne dass bei jedem Zugriff, erneut die Methode `getAvailableTimeslots()` aufgerufen werden muss. Mit Hilfe dieser Liste, können die Constraints AvailableLessonTimeConstraint, OnlyForenoonConstraint und NotAvailableTeacherTimeConstraint auf sehr einfache Weise, zusammen implementiert werden.

Variablen der harten Anforderungen ([2]):
Die in der Abbildung 14 mit einer hochgestellten [2] markierten Variablen werden benötigt, um die harten Anforderungen umzusetzen und werden im Verlauf dieser Umsetzung erzeugt. Daher sind diese Variablen immer vom Typ `IntVar`, `BoolVar` oder eine Menge solcher Variablen. Mit Variablen dieser Typen, wird das `CpModel` der OR-Tools Bibliothek aufgebaut. Die hier angegebenen Variablen, enthalten teilweise redundante Informationen. Einige von ihnen sind als Hilfsvariablen anzusehen, die eine bestimmte Sicht, auf die durch sie angegebenen Informationen liefern. Wie genau die Variablen belegt und verwendet werden, wird in Abschnitt 3.4 HardConstraints, Implementierung, erläutert. An dieser Stelle folgt jedoch ein kurzer Überblick.

- *roomVar, IntVar*: Diese Variable dient dazu, für eine Lesson anzugeben, in welchem Raum sie stattfindet. Da eine `IntVar` Variable ganze Zahlen als Wert annehmen kann, wird ein Raum durch seine ID repräsentiert. Der Wertebereich wird dazu auf die IDs der Räume eingeschränkt, in denen die Lesson stattfinden darf.

- *timeVars, list[IntVar]*: Eine einzelne *timeVar* Variable soll die ID eines Timeslots annehmen können. Auf diese Weise gibt sie an, wann die Lesson stattfindet. Es wird eine Liste verwendet, da eine Lesson auch mehrere (aufeinanderfolgende) Timeslots belegen kann. Es gibt eine Variable für jeden Timeslot den die Lesson belegt. Ist die Länge der Lesson zum Beispiel 2 (d.h. *timeslot_size = 2)*, beinhaltet die Liste *timeVars*, 2 Elemente.

- *weekdayVar, IntVar*: Gibt an, an welchem Wochentag die Lesson stattfindet. Die Wochentage sind dabei nummeriert. Sie ist eine Hilfsvariable, der Wochentag lässt sich aus jeder der *timeVars* Variablen ermitteln.

- *hourNumberVar, IntVar*: Gibt an, zu welcher Stunde, beziehungsweise zu welchem Timeslot eines Tages, die Lesson stattfindet (anfängt). Eine Belegung mit dem Wert 2, gibt zum Beispiel an, dass die Lesson mit dem zweiten Timeslot eines Tages beginnt. Sie ist eine Hilfsvariable, die Information ist indirekt bereits in den *timeVars* Variablen enthalten.

- *weekdayBoolVars, list[BoolVar]*: Die Liste enthält je eine `BoolVar` Variable für jeden Wochentag des Stundenplans. Die Angabe des Wochentages erfolgt auf andere Weise, als bei der *weekdayVar*. Damit ist auch dies als Hilfsvariable anzusehen. Die Variable, die den Wochentag repräsentiert, an dem die Lesson stattfindet, hat den Wert $True$ beziehungsweise 1.

- *timeslotBoolVars, list[BoolVar]*: Ähnlich wie *weekdayBoolVars*, jedoch gibt es eine Variable für jeden Timeslot. Variablen die Timeslots repräsentieren, zu denen die Lesson stattfindet, nehmen den Wert $True$ beziehungsweise 1 an.

3.3.6 Course

Ein Überblick über die Klasse `Course`, ist in Abbildung 15 zu sehen.

Attribute aus der Datenbank:

- *id, int*: Primärschlüssel in der Datenbank. Für die Suche nicht verwendet.

- *name, str*: Name des Kurses. Für die Ausgabe benötigt.

- *abbreviation, str*: Abkürzung des Kursnamens. Für die Ausgabe benötigt.

- *type, str*: Art des Kurses. Zum Beispiel „Vorlesung“ oder „Praktika“. Nur für die Stundenplanausgabe als Excel-Datei benötigt.

- *only_forenoon, bool*: Gibt an ob die Lessons der Kurses nur vormittags stattfinden dürfen.

- *all_in_one_block, bool*: Gibt an, ob alle Lessons des Kurses als Block stattfinden müssen.

- *is_lecture, bool*: Gibt an, ob die Lessons des Kurses als Vorlesung zu zählen sind.

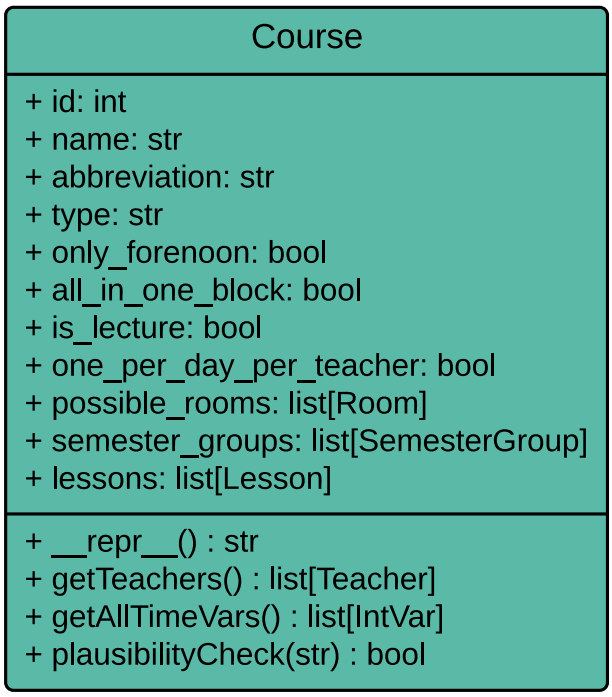

Abbildung 15: Aufbau der Klasse `Course`

- *`one_per_day_per_teacher, bool`*: Angabe für das OneCoursePerDayTeacherConstraint.
- *`possible_rooms, list[Room]`*: Liste mit allen Räumen, in denen Lessons des Kurses stattfinden können.
- *`semester_groups, list[SemesterGroup]`*: Liste mit allen Semestergruppen die an dem Kurs teilnehmen.
- *`lessons, list[Lesson]`*: Liste mit den Lessons des Kurses.

Methoden der Klasse:

- `__repr__() -> str`: Gibt eine textuelle Repräsentation des Kurses zurück.
- `getTeachers() -> list[Teacher]`: Liefert eine Liste mit allen Lehrenden, die an mindestens einer Lesson des Kurses beteiligt sind.
- `getAllTimeVars() -> list[IntVar]`: Jeder Lesson wird bei dem Hinzufügen der harten Anforderungen, eine Liste mit `IntVar` Variablen hinzugefügt, die die IDs der/des Timeslots angeben, zu dem die Lesson stattfindet. Diese Funktion liefert eine Liste mit der Vereinigung all dieser Listen der Lessons des Kurses.
- `plausibilityCheck(str) -> bool`: Führt einen Plausibilitätstest bezüglich der Längen der Lessons des Kurses durch. So kann zum Beispiel aufgedeckt werden, wenn der Kurs eine Lesson enthält die 4 Timeslots lang ist, obwohl der Kurs nur vormittags stattfinden soll und es nur 3 Timeslots pro Vormittag gibt. Ähnlich zur gleichnamigen Methode der `Teacher` Klasse, kann diese Methode helfen um Fehler

in nicht erfüllbaren Stundenplandaten zu finden. Das heißt Stundenplandaten, die zu einem nicht erfüllbarem `CpModel` führen.

3.4 HardConstraints, Implementierung

Für die Erstellung eines Stundenplans, müssen im Grunde lediglich zwei Aufgaben erledigt werden. Jeder Unterrichtsstunde im Stundenplan muss ein Zeitpunkt und ein Ort zugeordnet werden. Alle weiteren Bedingungen und Optimierungsmöglichkeiten beziehen sich ausschließlich darauf welche Zeitpunkte und Orte ausgewählt werden.

Daher bietet es sich an, für jedes `Lesson` Objekt – welche die Repräsentation einer Unterrichtsstunde darstellt – eine Variable für den Zeitpunkt und eine Variable für den Ort zu erstellen. Dies wird in Abschnitt 3.4.1 Zeit- und Raumvariablen gezeigt. Aufgabe der OR-Tools Bibliothek ist es, für diese Variablen Werte zu finden, sodass alle Anforderungen, die an die Belegung dieser Variablen gestellt werden, erfüllt sind. Alle Anforderungen beziehen sich entweder direkt, oder indirekt, durch die Verwendung von Hilfsvariablen, auf diese Variablen. Des Weiteren ist mit diesem Ansatz ohne weiteren Aufwand sichergestellt, dass jeder Lesson auch tatsächlich ein Ort und ein Zeitpunkt zugewiesen wird. Andere Anforderungen sind jedoch nicht implizit gegeben und müssen durch zusätzliche Constraints dem `CpModel` der OR-Tools Bibliothek hinzugefügt werden. So zum Beispiel die Anforderung, dass zwei Lessons nicht zum selben Zeitpunkt, im selben Raum stattfinden können. Dabei wird immer auf das Datenmodell und die Beziehungen der einzelnen Stundenplanobjekte untereinander zugegriffen, um bestimmte Constraints nur für bestimmte Objekte des Stundenplans hinzuzufügen.

Überblick über die Erstellung des `CpModels`

Es wird kurz erläutert, wie mithilfe des `CpModels` der OR-Tools Bibliothek, Probleme beschrieben werden können.

Ein Problem wird immer durch Variablen, die bestimmte Werte annehmen dürfen, und zusätzliche Bedingungen (Constraints) für die Belegung dieser Variablen, beschrieben. Im Constraint-Programming Teil der OR-Tools Bibliothek, gibt es einen grundlegenden Typ von Variablen, den Typ `IntVar`.

`IntVar`

Diese Variablen können Ganze Zahlen ($\mathbb{Z}$) als Werte annehmen. Der Wertebereich (Domain) kann weiter und beliebig klein beziehungsweise genau eingeschränkt werden. Die Domain muss jedoch mindestens ein Element enthalten. Der Wertebereich muss dabei nicht zusammenhängend sein. Eine Variable des Typs, wird erzeugt indem auf der Instanz des `CpModels` die Methode `NewIntVar(lb, ub, name)` aufgerufen wird. Die Methode liefert das erstellte `IntVar` Objekt zurück. *lb* gibt die untere Grenze des Wertebereich, *ub* die obere Grenze an. Alternativ kann auch erst ein Domain Objekt erstellt werden, welches mehr Möglichkeiten anbietet, um den Wertebereich zu definieren. Dann wird die Methode `NewIntVarFromDomain(domain, name)` verwendet. Mit dem dritten Parameter, *name*, kann der Variablen im Modell ein Name zugewiesen werden. Es können jedoch beliebig viele Variablen denselben Namen erhalten.

`BoolVar`

`IntVar` Variablen deren Domain nur die Elemente 0 und 1 enthalten, werden in dieser Arbeit als `BoolVar` Variablen bezeichnet. `BoolVar` ist jedoch (in der Python Implementierung) keine eigene Klasse der OR-Tools Bibliothek.

Die Klasse `IntVar` enthält die Methode `Not()`. Diese Methode darf nur auf Variablen mit dem Wertebereich $W \subseteq \{0, 1\}$, aufgerufen werden. Sie liefert eine Instanz der Klasse `_NotBooleanVariable`. Diese Klasse erbt nicht von `IntVar`, jedoch genau wie `IntVar` von der Klasse `LinearExpr`. Damit können diese Negationen von `BoolVar` Variablen zwar an den meisten Stellen der Bibliothek verwendet werden, teilweise sind jedoch nur `IntVar` Objekte erlaubt. Wo das `_NotBooleanVariable` Objekt verwendet werden kann, steht es immer für die negierte Belegung zu der `BoolVar` Variable, zu der es gehört. Also 0 bei dem Wert 1 und 1 bei dem Wert 0.

Hinzufügen von Constraints

Um die Belegung der erstellten Variablen weiter einzuschränken und an Bedingungen zu knüpfen, können dem `CpModel` Constraints hinzugefügt werden. Die Constraints beziehen sich dabei immer auf mindestens eine zuvor erstellte Variable. Es werden nun die in dieser Arbeit am häufigsten verwendeten Constraints vorgestellt.

- `Add(BoundedLinearExpression)`: Durch diese Methode können dem Modell lineare Gleichungen bzw. Ungleichungen hinzugefügt werden. Bei der Python Implementierung besteht die Besonderheit, dass die Gleichungen in direkter Form als Argument angegeben werden können. Es ist einer der Vergleichsoperatoren $==, != , <, >, <=, >=$ erlaubt. Auf jeder Seite des Vergleichsoperators müssen lineare Ausdrücke angegeben werden. Da auf der Menge der ganzen Zahlen gearbeitet wird, sind hier die Operatoren $+, -$ in unärer und binärer Form sowie, $*$ in binärer Form erlaubt. Als Operanden können `IntVar` Variablen oder Ganzzahl-Literale angegeben werden, solange der Ausdruck linear bleibt. Wurden zum Beispiel die Variablen $\boldsymbol{x}$ und $\boldsymbol{y}$ erstellt, kann folgender Aufruf ausgeführt werden: $model.Add(y == 4 * x + 5)$ Die direkte Schreibweise die bei der Python Implementierung möglich ist, macht das Hinzufügen sehr komfortabel und lesbar. So ist auch die Verwendung der `sum(...)` Funktion möglich. Sie erlaubt es die Summe einer ganzen Liste von Variablen zu bilden.

- `AddAllDifferent(Iterable)`: Fügt ein Constraint hinzu, sodass die übergebenen Variablen alle unterschiedliche Werte annehmen müssen.

- `AddBoolOr(Iterable)`: Die übergebenen Variablen müssen `BoolVar` Variablen oder Literale aus der Menge $\{0, 1\}$ sein. Auch die negierte Form als `_NotBooleanVariable` ist möglich. Fügt dem Modell die Bedingung hinzu, dass mindestens einer der übergebenen Variablen/Literale den Wert 1 haben muss. Es gibt entsprechende Methoden für *And* und XOR.

Jede der vorgestellten Methoden, liefert ein `Constraint` Objekt zurück. Diese Klasse bietet die Methode `OnlyEnforceIf(BoolVar)` an. Damit lässt sich die Erfüllung des

Constraints an die Belegung einer `BoolVar` Variablen knüpfen. Auch die Negation einer `BoolVar` Variable kann übergeben werden.

`BoolVar` Variablen mit der Erfüllung von Ausdrücken belegen

Die Methode `OnlyEnforceIf(BoolVar)` kann andersherum auch dafür verwendet werden, um eine `BoolVar` Variable, abhängig einer bestimmten Bedingung, mit Werten zu belegen. Dies ist oft notwendig, wenn eine `BoolVar` anzeigen soll, ob ein bestimmter Ausdruck gilt oder nicht. Dies oftmals benötigt, da die Methode `OnlyEnforceIf(BoolVar)` tatsächlich nur `BoolVar` Variablen entgegennehmen kann und keine anderen Ausdrücke.

Angenommen, es gibt eine `IntVar` Variable ***x*** und eine `BoolVar` Variable ***xIsPositive***. Nun soll die Variable ***xIsPositive***, genau dann den Wert 1 annehmen, wenn ***x*** einen positiven Wert annimmt. Dazu müssen die beiden folgenden Methodenaufrufe ausgeführt werden:

```
1    model.Add( x >  0 ).OnlyEnforceIf(xIsPositive)
2    model.Add( x <= 0 ).OnlyEnforceIf(xIsPositive.Not())
```

Diese Form ist zunächst unintuitiv. Angenommen b steht für den Ausdruck $x > 0$ und a für die Variable $xIsPositive$. Was erreicht werden soll, ist $b \iff a$. Ein Aufruf

```
1    model.Add( b ).OnlyEnforceIf(a)
```

bedeutet zunächst ja $a \implies b$. Also nur eine Seite der Äquivalenz. Da die Methode `OnlyEnforceIf` keine Ausdrücke entgegennehmen kann, kann nicht direkt die Implikation in die andere Richtung ($b \implies a$) hinzugefügt werden. Da eine Implikation aber Äquivalent zu ihrem Umkehrschluss (Kontraposition) ist, kann die Äquivalenz über die zweite Implikation $\neg a \implies \neg b$ erreicht werden. Als OR-Tools Constraint:

```
1    model.Add( b.Not() ).OnlyEnforceIf(a.Not)
```

Wenn ***b*** ein Ausdruck ist, ist es wichtig darauf zu achten, dass bei der zweiten Implikation tatsächlich das exakte Gegenteil angegeben wird. Sodass zum Beispiel im Falle einer `IntVar`, ihr gesamter Wertebereich durch die beiden Implikationen abgedeckt ist.

Diese Form eine Äquivalenz zu erreichen um `BoolVar` mit Werten zu belegen, ist einem Beispiel der GitHub Seite der OR-Tools Bibliothek entnommen. [Goof]

Aufteilung der Implementierung in einzelne Python-Funktionen

Für alle harten Anforderungen, inklusive dem Erstellen der zugrundeliegenden Basis- und der Hilfsvariablen, gibt es Funktionen die in der Python Skriptdatei ***HardConstrains.py*** abgelegt sind. Im Folgendem wird auf die einzelnen Funktionen eingegangen und wie mit ihnen die einzelnen Constraints umgesetzt werden. Viele Funktionen setzen nur ein Constraint um. Da aber in einigen Fällen, auf einfache Weise, mehrere Constraints auf einmal hinzugefügt werden können, fügen manche Funktionen dem `CpModel` auch mehrere Anforderungen hinzu. Dies ist insbesondere bei der Erstellung der grundlegenden Variablen (Abschnitt 3.4.1 Zeit- und Raumvariablen) der Fall, da durch Angabe

des Wertebereichs bereits mehrere Anforderungen umgesetzt werden können. Bei den meisten Funktionen ist es möglich, diese, zum Beispiel zu Testzwecken, schlicht nicht aufzurufen um so einzelne Constraints nicht zu erfüllen. Dies war bei der Entwicklung hilfreich, um jenes Constraint zu ermitteln, welches gerade dafür sorgt, dass ein Stundenplanmodell nicht erfüllbar ist.

In den folgenden Abschnitten, werden alle relevanten Inhalte der Funktionen zur Implementierung der Constraints, durch Code-Ausschnitte gezeigt. Um diese nicht zu unübersichtlich werden zu lassen, und da der Inhalt im begleitenden Text erklärt wird, sind in den Ausschnitten fast sämtliche Kommentare entfernt worden.

3.4.1 Zeit- und Raumvariablen

Wie im vorigen Abschnitt erläutert, wird für jede Lesson eine Zeit- und eine Raumvariable benötigt, die bestimmen, wann und wo die Lesson stattfinden soll. Diese werden durch die Funktion `createTimeAndRoomVars` erstellt. Sämtliche hier beschriebenen Schritte, werden auf alle Lessons angewendet. Die in den Code-Listings enthaltenen Anweisungen, werden daher für jedes einzelne `Lesson` Objekt ausgeführt. Die dort vorkommende Variable *`lesson`*, enthält jeweils ein `Lesson` Objekt, aus der Menge aller Lessons über die iteriert wird.

Raumvariablen

Jede Lesson erhält eine einzelne Variable *`roomVar`*, vom Typ `IntVar`, der OR-Tools Bibliothek, die angibt in welchem Raum die Lesson stattfindet. Ihre Domain enthält die IDs aller Räume in denen die jeweilige Lesson stattfinden kann. Die möglichen Räume sind dabei immer durch den Kurs der Lesson definiert.

Bei der Erstellung der Variable für den Raum einer Lesson, ist nichts besonderes zu beachten, die möglichen Räume sind bereits im zugehörigen `Course` Objekt jeder Lesson als Liste enthalten. Der zugehörige Kurs der Lesson sei im Listing 1 durch die Variable *`course`* gegeben. Von allen `Room` Objekten, wird jeweils die ID ermittelt (Zeile 1). Die Menge dieser IDs wird dann als Domain für die Raumvariable verwendet. Die tatsächliche Belegung der *`roomVar`* eines `Lesson` Objektes, wird dann bei einer gefundenen Lösung angeben, in welchem Raum die Lesson beim gefundenen Stundenplan stattfindet.

```
1 roomDomain = Domain.FromValues(map(lambda r: r.id, course.possible_rooms))
2 lesson.roomVar = model.NewIntVarFromDomain(roomDomain, "roomVar")
```

Codeausschnitt 1: HardConstraints.py - createTimeAndRoomVars: roomVar

Zeitvariablen

Eine Zeitvariable, ebenfalls vom Typ `IntVar`, gibt an, zu welchem Timeslot, also wann im Stundenplan, eine Lesson stattfindet. Ihre Domain enthält dementsprechend die IDs der Timeslots, zu denen die Lesson stattfinden darf.

Um auf diese Variablen, zum Beispiel bei der Implementierung anderer Constraints, zugreifen zu können, werden sie direkt im jeweiligen `Lesson` Objekt gespeichert. Dabei wird die Eigenschaft von Python ausgenutzt, einem Objekt auch nach Instanziierung noch Variablen hinzufügen zu können.

In diesem Abschnitt wird neben der Erstellung nötiger Variablen, die Implementierung folgender Constraints beschrieben. Die Implementierung erfolgt in der Funktion `createTimeAndRoomVars` in der Datei ***HardConstraints.py***.

- AvailableLessonTimeConstraint (2.3.9): Es können ein oder mehrere Timeslots vorgegeben werden, zu denen eine Lesson stattfinden soll.
- OnlyForenoonConstraint (2.3.8): Bestimmte Lessons sollen nur vormittags stattfinden.
- NotAvailableTeacherTimeConstraint (2.3.7): Lessons können nicht zu Zeiten stattfinden, zu denen die Lehrkräfte der Lesson nicht verfügbar sind.
- LessonsAtSameTimeConstraint (2.3.10): Lessons sollen zeitgleich (auch am selben Tag) anfangen und Constraints die dies verhindern würden, sollen ignoriert werden.

Die ersten drei Constraints können jeweils alleine dadurch umgesetzt werden, dass die Wertebereiche der Zeitvariablen einer Lesson eingeschränkt werden. Da die Vorgaben zu erlaubten, beziehungsweise verbotenen Timeslots der drei Constraints statisch sind, können die erlaubten Timeslots bereits vor der eigentlichen Suche ermittelt werden. Die Domains der einzelnen Zeitvariablen enthalten dann nur noch die IDs der erlaubten Timeslots. Das Verringern der Domain ist hier ein effizientes und einfaches Mittel die Vorgaben umzusetzen. Es sind dafür keine weiteren Constraints dem `CpModel` hinzuzufügen.

Das letzte Constraint kann dadurch garantiert werden, indem sich die Lessons, die gleichzeitig stattfinden sollen, Zeitvariablen teilen.

Da eine Lesson mehrere Timeslots belegen kann, erhält jedes `Lesson` Objekt eine Liste mit Zeitvariablen. Der Name dieser Liste ist *`timeVars`*. Die Anzahl der enthaltenen Zeitvariablen entspricht der Länge der Lesson (*`lesson.timeslot_size`*). Jede *`timeVar`* in der Liste gibt, die ID eines `Timeslots` (90 Minuten Block) im Stundenplan an, zu dem die Lesson stattfindet. Bei allen Lessons, die nur einen Zeitslot belegen, enthält die Liste demnach auch nur eine einzelne *`timeVar`*. Bei Lessons welche sich über mehrere Timeslots erstrecken, gibt es dementsprechend eine Variable für jeden Timeslot den die Lesson belegen soll. Die Erstellung mehrere Zeitvariablen für eine mehrstündige Lesson ist im Grunde nicht notwendig um den Zeitpunkt der Lesson zu bestimmen, dazu würde eine einzelne Variable mit dem Lessonstart genügen. Eine Liste mit Variablen für den gesamten Bereich den jede Lesson belegt, hilft jedoch bei der Umsetzung anderer Constraints. Alle, außer der ersten *`timeVar`* können daher als Hilfsvariablen angesehen werden. Da eine mehrstündige Lesson immer aufeinanderfolgende Timeslots belegt, ist es notwendig, dafür zu sorgen, dass die einzelnen Zeitvariablen der Liste, immer die IDs

aufeinanderfolgender Timeslots annehmen. Dies ist leicht umzusetzen, da die IDs immer zählend aufsteigen.

Es muss jedoch auch sichergestellt werden, dass jede mehrstündige Lesson auch innerhalb eines Tages stattfindet. Also eine zweistündige Lesson zum Beispiel nicht im letzten Timeslot eines Tages anfängt und mit dem ersten Timeslot des Folgetages endet.

Einschränkung der Domains der Zeitvariablen

Um die Zeitvariablen einer Lesson so einzuschränken, dass die drei zuerst genannten Constraints erfüllt werden, muss zunächst eine Liste mit allen Timeslots erstellt werden, zu denen die Lesson grundsätzlich stattfinden darf. Diese Liste liefert die Methode `getAvailableTimeslots(...)` der Klasse `Lesson`. Wie in Abschnitt 3.3.5 erläutert, wird nach dem Laden aller Daten, diese Methode für jedes `Lesson` Objekt aufgerufen und das Ergebnis in der Variablen *`availableTimeslotsFiltered`* gespeichert. Mit dieser Liste von Timeslots wird an dieser Stelle weitergearbeitet. Sie ist zunächst für alle Zeitvariablen einer (mehrstündigen) Lesson gleich.

Es muss noch sichergestellt werden, dass mehrstündige Lessons immer innerhalb eines Tages liegen. Diese Anforderung könnte verletzt werden, da die Timeslots des gesamten Stundenplans (also auch tagesübergreifend), als aufeinanderfolgend anzusehen sind[13]. Vermieden wird diese Belegung der Zeitvariablen mit Timeslots verschiedener Tage, indem die Listen der möglichen Timeslots, für alle, bis auf die letzte Zeitvariable, weiter angepasst werden. Für die vorletzte Zeitvariable, werden zum Beispiel alle Timeslots aus der Liste entfernt, die am Ende eines Tages liegen. Für die vor-vorletzte Zeitvariable, dann die beiden letzten Timeslots jedes Tages und so weiter. Dies stellt sicher, dass die mehrstündige Lesson mindestens so früh an einem Tag anfangen muss, dass der letzte Timeslot, den die Lesson belegt, garantiert noch am selben Tag liegt.

Wie oben erwähnt, muss noch sichergestellt werden, dass die Werte der Zeitvariablen einer Lesson, immer direkt aufeinanderfolgen, da eine mehrstündige Lesson natürlich nur direkt hintereinander liegende Timeslots belegen soll. Dies wird mittels einfacher Constraints erreicht, die jeweils diesen linearen Zusammenhang zweier benachbarter Variablen beschreiben. Dies wird am Ende dieses Abschnitts gezeigt. Es hilft jedoch beim Verständnis der folgenden Erklärungen, wenn man sich dies bereits als gegeben vorstellt. Also, dass die Belegung der Zeitvariablen in der Liste mit Zeitvariablen einer Lesson, nur aufsteigend (zählend) stattfinden kann.

Zeitgleiches Stattfinden von Lessons

Lessons die zeitgleich stattfinden sollen[14] (**LessonsAtSameTimeConstraint**), wobei das **TeacherTimeConstraint** und das **RoomTimeConstraint** ignoriert werden können, teilen sich ihre *`timeVar`* Objekte. So ist das LessonsAtSameTimeConstraint bereits erfüllt und diese Implementierung, macht es insbesondere sehr einfach, die Constraints TeacherTimeConstraint und das RoomTimeConstraint zu ignorieren. Dies ist der Fall,

[13] Der erste Timeslot am Dienstag, ist zum Beispiel der Folgetimeslot des letzten Timeslots am Montag.

[14] Es wird auch im Folgendem von „zeitgleich Stattfinden“ gesprochen, auch wenn es bei dem Constraint eigentlich eher darum geht, dass die Lessons nur zeitgleich im Stundenplan eingetragen werden. In der Praxis würden die Lessons dann eventuell nicht zeitgleich stattfinden, zum Beispiel weil sie im wöchentlichen Wechsel abgehalten werden. Siehe dazu 2.3.10 LessonsAtSameTimeConstraint.

da es zum Beispiel aus Sicht eines Lehrenden, für alle seine Lessons nach wie vor keine zwei *timeVar* Variablen gibt, die den gleichen Wert annehmen können. Es ist jedoch zu beachten, dass die Lessons, die zeitgleich anfangen sollen, unterschiedlich lang sein können. In dem Fall teilen sich die `Lesson` Objekte nur so viele Variablen wie auch beide Lessons lang sind. Ist zum Beispiel eine Lesson zwei und eine andere einen Timeslot lang, teilen sich die Lessons die erste Zeitvariable, und nur die mehrstündige Lesson erhält eine zweite Zeitvariable.

Da bei der Implementierung einmal über alle Lessons iteriert wird, sollen jeweils die Zeitvariablen aller Lessons eines LessonsAtSameTimeConstraint Konstrukts (also die Lessons die zeitgleich stattfinden sollen) angelegt werden, sobald bei der Iteration eine der enthaltenen Lessons erreicht ist. Daraus folgt, dass für spätere Lessons in der Iteration, geprüft werden muss, ob die Variablen für diese Lesson noch nicht angelegt wurden.

Wie in Abschnitt 3.3.5 Lesson beschrieben, enthält, nach einer kleinen Überarbeitung der Daten durch das ***ORM.py*** Skript, jede Variable *lessons_at_same_time* jeder(!) Lesson, welche Bestandteil eines LessonsAtSameTimeConstraints ist, die jeweils anderen Lessons, die zeitgleich stattfinden sollen.

Implementierung

Das Anlegen der *timeVars* Variablen gestaltet sich verhältnismäßig aufwendig, da sich mehrere Lessons die gleichen Variablen teilen können. Der Vorgang ist sehr allgemein aufgebaut, das heißt es wird immer angenommen, dass es um mehrere Lessons und Zeitvariablen je Lesson geht, auch wenn die verwendeten Listen in den meisten („normalen“) Fällen nur einelementig sind. Dies ist immer dann der Fall, wenn die Lessons einstündig, beziehungsweise nicht Teil eines LessonsAtSameTimeConstraints sind.

Für jede Lesson wird der Inhalt von Listing 2 nur ausgeführt, wenn die *timeVars* Liste für die Lesson noch nicht vorhanden ist. Zunächst wird eine Liste aller Lessons erstellt, die zeitgleich stattfinden sollen (Zeile 1)[15] und es wird die maximale Länge dieser Lessons ermittelt (Zeile 2). Für jede dieser Lessons wird anschließend die *timeVars* Liste angelegt.

```
1 lessonsAtSameTime = [lesson] + lesson.lessons_at_same_time
2 maxLessonSize = max(map(lambda l: l.timeslot_size, lessonsAtSameTime))
```

Codeausschnitt 2: HardConstraints.py - createTimeAndRoomVars: timeVars, Teil 1

Nun wird für jeden der Timeslots, den eine der Lessons belegen kann, eine *timeVar* Variable erstellt. Dazu wird von null bis zur maximalen Lessonlänge iteriert (Listing 3, Zeile 1). Pro Iteration wird eine Zeitvariable erstellt. Den Inhalt dieser Schleife zeigen die folgenden Listings, beginnend mit Listing 3. Auf diese Weise werden bei der Iteration, jeweils die Lessons herausgefiltert, die länger sind (mehr Timeslots belegen) als die Iterationsvariable *i*. Diese `Lesson` Objekte werden in der Liste *lessonsWithCurrentSize* gesammelt. Dies sind dann genau die Lessons, die sich die anschließend erstellte Zeitvariable teilen sollen.

[15] Die Variable *lessons_at_same_time* eines `Lesson` Objektes, enthält nur die anderen Lessons, die zeitgleich stattfinden sollen, nicht aber das `Lesson` Objekt, zu dem die Variable gehört, selber.

Außerdem werden die Timeslots ermittelt, zu denen die Lessons stattfinden können. Dies ist die Schnittmenge der möglichen Timeslots der einzelnen Lessons. Dazu wird die Hilfsmethode `intersectAll(Iterable[Iterable])` aus der Datei ***HelperFunctions.py*** verwendet. Diese bildet die Schnittmenge aus einer Menge von Iterables die als Iterable übergeben werden. Die Listen der möglichen Timeslots sind für jede Lesson gespeichert (*availableTimeslotsFiltered*). Hierbei handelt es sich um die oben angesprochene Liste, die nach dem Laden der Daten für jede Lesson angelegt wird.

```
1 for i in range(0, maxLessonSize):
2     lessonsWithCurrentSize = list(filter(lambda l: l.timeslot_size > i, lessonsSameTime))
3     availableTimeslots = intersectAll(map(lambda l: l.availableTimeslotsFiltered,
      ↪  lessonsWithCurrentSize))
```

Codeausschnitt 3: HardConstraints.py - createTimeAndRoomVars: timeVars, Teil 2

Wie oben erwähnt, ist es des Weiteren vonnöten, das sich mehrstündige Lessons nicht über mehrere Tage erstrecken. Jeder Timeslot enthält in der Variable *number*, die Nummer des Timeslots je Tag, also beispielsweise eins für die erste Stunde oder sechs für die letzte Stunde. Es soll erreicht werden, dass zum Beispiel der erste Timeslot einer zweistündigen Lesson, nicht in der letzten Stunde liegen kann. Die Konstante *TIMESLOTS_PER_DAY*, enthält die Anzahl der Stunden je Tag, also für den Stundenplan der TH-Lübeck sechs.

Je Iteration der oben genannten Schleife, wird eine Zeitvariable erstellt, beginnend mit der ersten, also dem Anfang der Lesson. Listing 4 zeigt in der ersten Zeile, wie die Nummer des Timeslots berechnet wird, zudem der jeweilige Teil der Lesson spätestens stattfinden darf. Da bei der Iteration mit 0 gestartet wird, die Nummerierung aber mit 1 beginnt, wird zur Zählvariablen *i*, eins addiert.

Beispiel für den Fall einer einzelnen Lesson der Länge 1: Es soll keine Einschränkungen geben, da einstündige Lessons sowieso schon nie an zwei Tagen gleichzeitig stattfinden können. Da also *maxLessonsSize* den Wert 1 hat: $TIMESLOTS\_PER\_DAY - 1 + 0 + 1$ So entspricht die errechnete letzte Nummer auch dem letzten Timeslot jedes Tages.

In Zeile 2 des Codeausschnitts 4, wird dann mit der errechneten maximalen Timeslot-Nummer die zuvor erstellte Liste, noch ein letztes Mal eingeschränkt.

```
1     lastPossibleTimeslotNumber = TIMESLOTS_PER_DAY - maxLessonSize + i + 1
2     timeslots = list(filter(lambda t: t.number <= lastPossibleTimeslotNumber,
      ↪  availableTimeslots))
```

Codeausschnitt 4: HardConstraints.py - createTimeAndRoomVars: timeVars, Teil 3

Das Erstellen der *timeVar* Variable, funktioniert analog zur Erstellung von *roomVar*, nur diesmal mit den IDs der Timeslots aus der *timeslots* Liste als Domain. Diese wird der *timeVars* Liste der entsprechenden Lessons hinzugefügt. Listing 5

```
    timeDomain = Domain.FromValues(map(lambda t: t.id, timeslots))
    timeVar = model.NewIntVarFromDomain(timeDomain, "")
    for lessonAtSameTime in lessonsWithCurrentSize:
        lessonAtSameTime.timeVars.append(timeVar)
```

Codeausschnitt 5: HardConstraints.py - createTimeAndRoomVars: timeVars, Teil 4

Das direkte Aufeinanderfolgen der Timeslots einer Lesson wird durch ein einfaches `LinearConstraint` erreicht. Die in letzter Iteration erstellte *timeVar* wird dafür jeweils gespeichert. Für jede *timeVar* außer der ersten gilt daher:

```
    if lastTimeVar:
        # Timeslots shall be consecutive
        model.Add(timeVar == lastTimeVar + 1)
    lastTimeVar = timeVar
```

Codeausschnitt 6: HardConstraints.py - createTimeAndRoomVars: timeVars, Teil 5

Der Inhalt der gesamten Funktion ist im Anhang unter B.1 zu finden.

3.4.2 Hilfsvariablen

Für viele spätere Constraints, ist die Umsetzung einfacher, wenn andere Sichtweisen in Form von abgeleiteten Variablen auf den Stundenplan vorhanden sind. Um die Erzeugung und Wertzuweisung dieser Hilfsvariablen geht es in diesem Abschnitt.

Die folgend erläuterten Hilfsvariablen *weekdayVar*, *hourNumberVar* und *weekdayBoolVars*, werden jeweils einer Lesson hinzugefügt. Da sich aber Lessons, die durch das **LessonsAtSameTimeConstraint** gleichzeitig stattfinden, auch diese Hilfsvariablen teilen können, werden die erstellen Variablen immer auch allen Lessons der *lessons_at_same_time* Liste des jeweiligen `Lesson` Objektes, hinzugefügt. Dafür muss dann wiederum beim Iterieren über die Lessons, zuvor abgefragt werden, ob die Hilfsvariablen dem aktuellen `Lesson` Objekt nicht schon hinzugefügt wurden.

LessonTime Hilfsvariablen, *weekdayVar* und *hourNumberVar*

Die erste *timeVar* jeder Lesson, gibt an, zu welchem Timeslot die Lesson beginnt. Für manche Constraints, ist es aber interessant an welchem Wochentag oder zu welcher Stunde an diesem Tag, die Lesson stattfindet. Außerdem ist es sogar hilfreich für jeden Tag die Information als `BoolVar` zu haben, ob die Lesson an diesem Wochentag stattfindet oder nicht. Die Hilfsvariablen *weekdayVar* und *hourNumberVar*, werden für jede `Lesson`-Instanz durch die Funktion `createLessonTimeHelperVariables` der Datei ***HardConstraints.py*** erzeugt. *hourNumberVar* soll die Nummer der Stunde angeben, zu der die zugehörige Lesson beginnt. Also die Nummer des Timeslots zu dem die Lesson beginnt, am Tag an dem die Lesson stattfindet. *weekdayVar*, soll den Wochentag, an dem die Lesson stattfindet, in Form einer Nummer angeben. Die Nummerierung der Wochentage beginnt dazu mit 1 für Montag.

Die Information zu Wochentag und Stunde lassen sich direkt aus der ID des Timeslots berechnen, welche ja in der ersten ***timeVar*** für jede Lesson zur Verfügung steht. OR-Tools erlaubt es, lineare Zusammenhänge mehrere Variablen, in einem einzelnen Constraint anzugeben. Die für diesen Fall benötigte Gleichung ist:

$$t_{id} == (weekday - 1) \cdot TIMESLOTS\_PER\_DAY + hourNumber \tag{13}$$

Dabei ist t_{id} die ID des Timeslots zu dem die Lesson stattfindet, also bei 5 Wochentagen und 6 Stunden pro Tag, ein Wert zwischen 1 und 30. *weekday* ist die Nummer des Wochentages also ein Wert zwischen 1 und 5 und damit der Wert, den die Hilfsvariable ***weekdayVar***, annehmen soll. *hourNumber* entspricht der Nummer des Timeslots an einem Tag, also beim TH-Lübeck Stundenplan ein Wert zwischen 1 und 6, und damit dem Wert, der der Variablen ***hourNumberVar***, zugewiesen werden soll. Die Anzahl der Stunden die an einem Tag stattfinden ist in der Konstanten $TIMESLOTS\_PER\_DAY$ enthalten. Da die Wertebereiche der OR-Tools Variablen beliebig eingeschränkt werden können, gibt es für das hier angewendete Constraint für alle Timeslot IDs auch nur eine Lösung der Gleichung. Die Gleichung kann genauso wie oben beschrieben als Constraint dem `CpModel` hinzugefügt werden. Dazu werden zuerst die Variablen für Wochentag und Stunde mit genannten Wertebereichen erstellt und anschließend die Gleichung als lineares Constraint hinzugefügt. Dies ist in Listing 7 gezeigt.

```
lesson.weekdayVar = model.NewIntVar(1, WEEKDAYS, "")
lesson.hourNumberVar = model.NewIntVar(1, TIMESLOTS_PER_DAY, "")

model.Add(
    lesson.timeVars[0] ==
        (lesson.weekdayVar - 1) * TIMESLOTS_PER_DAY + lesson.hourNumberVar
)
```

Codeausschnitt 7: HardConstraints.py - createLessonTimeHelperVariables, Teil 1

WeekdayBool Hilfsvariablen, weekdayBoolVars

Nun soll für jeden Wochentag eine `BoolVar` angelegt werden, die den Wert True annehmen soll, wenn die Lesson an dem entsprechendem Tag stattfindet. Diese Variablen werden in einer Liste ***weekdayBoolVars*** gesammelt. Zunächst der Codeausschnitt, der zeigt, wie die Variablen angelegt werden und die korrekte Belegung erzwungen wird (Listing 8).

```
lesson.weekdayBoolVars = []
for weekday in orm.getTimeslotsPerDay():  # weekday is a list with the Timeslots of a day
    dayBool = model.NewBoolVar("weekdayBool")
    lesson.weekdayBoolVars.append(dayBool)

    bound = BoundedLinearExpression(lesson.timeVars[0], [weekday[0].id, weekday[-1].id])
    model.Add(bound).OnlyEnforceIf(dayBool)
 model.Add(sum(lesson.weekdayBoolVars) == 1)
```

Codeausschnitt 8: HardConstraints.py - createLessonTimeHelperVariables, Teil 2

Es wird über die einzelnen Wochentage iteriert. Die Iterationsvariable *weekday*, ist in jeder Iteration eine Liste der Timeslots, die am jeweiligen Wochentag stattfinden.

Die `BoundedLinearExpression` in Zeile 6, sichert zu, dass sich der Wert der Variablen des ersten Parameters, zwischen den Werten des zweiten und dritten Parameters, befindet. Das erste Argument ist das *timeVar* Objekt der Lesson, das angibt zu welchem Timeslot die Lesson beginnt. Für die beiden anderen Parameter, werden die IDs des ersten und letzten Timeslots des Tages, für den die *dayBool* Variable erstellt wird, verwendet. Die erstellte `BoundedLinearExpression` gilt also nur, wenn die Lesson zu einem Timeslot beginnt, der am Wochentag in der jeweiligen Iteration liegt.

Da diese `BoundedLinearExpression` mit einem `OnlyEnforceIf()` und der *dayBool* Variable hinzugefügt wird (Zeile 7), ist zugesichert, dass, wenn die *dayBool* Variable True ist, die Lesson auch tatsächlich am zugehörigen Wochentag stattfindet.

Dies ist also bereits eine Seite der Äquivalenz:

$$\text{„Lesson findet am Wochentag der } \textit{dayBool} \text{ Variable statt“} \iff \textit{dayBool} \tag{14}$$

Da im Anschluss der Erstellung der *dayBool* Variablen aller Wochentage, sichergestellt wird, dass immer genau eine von ihnen den Wert `True` annimmt, gilt auch die andere Seite der Äquivalenz.

LessonTimeslotBool Hilfsvariablen, *timeslotBoolVars*

Ähnlich zu den `BoolVar` Variablen im vorigen Abschnitt, die angeben ob eine Lesson an einem bestimmten Wochentag stattfindet, sollen nun noch `BoolVar` Variablen hinzugefügt werden, die angeben ob eine Lesson zu einem bestimmten Timeslot stattfindet. Es soll also für jede Lesson, eine Liste von `BoolVar` Variablen, für jeden Timeslot des Stundenplans, geben. Die Variablen, die für die für die Timeslots stehen, zu denen die Lesson stattfindet, sollen den Wert `True` annehmen.

Da eine Lesson auch mehrstündig sein kann und so zu mehreren Timeslots stattfindet, wird hier zwei Schritten vorgegangen. Im ersten Schritt wird für jede *timeVar* einer Lesson, eine Liste mit `BoolVar` Variablen für jeden Timeslot an dem die Lesson stattfinden kann, angelegt. Da sich diese Liste, nur auf einen einstündigen Teil einer Lesson bezieht, hat immer nur eine der Variablen, den Wert `True`. Dies erleichtert die tatsächliche Wertzuweisung.

Der zweite Schritt, muss lediglich für mehrstündige Lessons ausgeführt werden. Dabei werden die einzelnen `BoolVar` Listen, die jeweils für einen Teil der Lesson stehen, verknüpft, so dass eine einzelne Liste entsteht, die für die ganze Lesson steht. Ein Beispiel: Eine dreistündige Lesson, belegt drei aufeinanderfolgende Timeslots. Angenommen, dies seien die Timeslots 3,4 und 5. Dann wird zunächst für jeden der Timeslots, den die Lesson belegt, eine Liste mit `BoolVar` Variablen erstellt. Von den 30 Variablen, die jede dieser Listen enthält, nimmt immer nur eine Variable den Wert `True` an. Bei der ersten Liste ist dies die 3., bei der zweiten die 4. und bei der dritten Liste die 5. Variable, entsprechend zu den Timeslots, welche die Lesson belegt. Durch den zweiten Schritt, werden die Listen dann zu einer (neuen Liste) verknüpft. Dabei werden jeweils die ersten, zweiten, dritten,

etc. Variablen durch eine Oder-Verknüpfung, zusammengefasst. In der finalen Liste haben dann die 3., 4. und 5. Variable alle den Wert `True` und alle anderen den Wert `False`. So lässt sich aus dieser Liste genau ablesen, welche Timeslots die Lesson belegt.

Implementierung

Alles Folgende, wird jeweils für alle Lessons des Stundenplans ausgeführt. Die Implementierung des ersten Schritts, ist in Listing 9 gezeigt. Die Liste *timeslotBoolVarsAll*, in Zeile 1, soll alle einzelnen Listen der einzelnen Timeslots, den die Lesson belegt, enthalten. Ziel in diesem Schritt, ist es, dieser Liste, die einzelnen Listen hinzuzufügen.

Außerdem wird als Optimierung, eine „konstante“ `BoolVar` Variable verwendet, die nur den Wert 0 (`False`) annehmen kann. Diese wird an allen Stellen der Listen eingesetzt, zu denen die Lesson nicht stattfinden kann. Diese Information, wird der Liste der Timeslots, welche die Lesson potenziell belegen kann, *availableTimeslotsFiltered*, entnommen.

```
1  timeslotBoolVarsAll = []
2  falseVar = model.NewIntVar(0, 0, "False")
3  for timeVar in lesson.timeVars:
4      timeslotBoolVars = []
5      for t in orm.getTimeslots():
6          if t in lesson.availableTimeslotsFiltered:
7              timeslotBoolVars.append(model.NewBoolVar(""))
8          else:
9              timeslotBoolVars.append(falseVar)
10
11     model.Add(sum(timeslotBoolVars) == 1)
12     model.Add(
13         timeVar == sum(list(
14             map(lambda t: (t + 1) * timeslotBoolVars[t], range(len(orm.getTimeslots()))))
15         ))
16
17     timeslotBoolVarsAll.append(timeslotBoolVars)
```

Codeausschnitt 9: HardConstraints.py - createLessonTimeslotBoolHelperVariables, 1

Ab Zeile 3, werden nun die einzelnen Listen erstellt. Für jeden einzelnen Timeslot des Stundenplans, wird zunächst entschieden, ob die konstante `BoolVar` *falseVar* verwendet wird, oder eine neue Variable erstellt werden muss. In beiden Fällen, wird der Liste ein weiteres Element hinzugefügt. Ab Zeile 11, folgt die Wertzuweisung. Es wird zunächst sichergestellt, dass nur eine der Variablen den Wert 1 annehmen kann. Dann folgt ein lineares Constraint, das dafür sorgt, dass die richtige Variable den Wert 1 annimmt. Dazu wird der Wert der `IntVar` Variablen *timeVar* verwendet. Sie enthält die Nummer des Timeslots, zu dem der Teil der Lesson stattfindet.

Es wird eine Gleichung angegeben, bei der auf der einen Seite, die *timeVar* Variable steht, und auf der anderen Seite, die Summe aller `BoolVar` Variablen der Liste, jeweils multipliziert mit der ID des Timeslots für den sie stehen. Ein kleines Beispiel mit nur 4

Timeslots und einer Lesson die zu Timeslot 3 stattfindet:

$$3 == 1 \cdot a + 2 \cdot b + 3 \cdot c + 4 \cdot d \tag{15}$$

Auf der rechten Seite der Gleichung, ist jeweils die Nummer des Timeslots (1,2,3,4) mit der zugehörigen `BoolVar` (a-d) multipliziert. Da nur eine der 4 Variablen den Wert 1 annehmen kann (und alle anderen den Wert 0), gibt es nur eine Lösung.

Nun der zweite Schritt. Bei einstündigen Lessons ist das Ziel bereits erreicht. Die erstelle Liste mit `BoolVar` Variablen repräsentieren alle Timeslots welche die Lesson belegen kann, immer nur einen zur Zeit. Sie wird der Lesson als *`timeslotBoolVars`* Liste angehängt. Belegt die Lesson aber zum Beispiel immer drei Timeslots, wurden bisher drei Listen erstellt, in denen jeweils eine `BoolVar` für einen der drei Timeslots, den Wert True annimmt. Daraus soll nun eine Liste entstehen, in der alle drei `BoolVar` Variablen für die drei Timeslots den Wert True annehmen. Dies ist in Listing 10 gezeigt.

Zunächst wird so eine neue Liste *`timeslotBoolVars`* erzeugt, genauso wie im ersten Schritt. Für alle `BoolVar` in dieser Liste die nicht die *`falseVar`* sind, wird der Wert der neuen Variable gleich der Veroderung der zuvor erstellten Variablen in den bisherigen Listen gesetzt. Bei dem Beispiel mit eine Lesson, die drei Timeslots belegt gilt folgendes, wenn i jeweils den Index der Timeslots in den Listen, angibt. „+“ meint hier das logische Oder.

$$timeslotBoolVars[i] \iff \tag{16}$$
$$timeslotBoolVarsAll[0][i] + timeslotBoolVarsAll[1][i] + timeslotBoolVarsAll[2][i]$$

Da ausgeschlossen ist, dass mehr als eine der (im Beispiel drei) Variablen, den Wert 1 annimmt, kann einfach die Summe gebildet werden. Dies ist in Zeile 11 des Listings 10 zu sehen.

```
1  if len(timeslotBoolVarsAll) > 1:
2      timeslotBoolVars = []
3      for t in orm.getTimeslots():
4          if t in lesson.availableTimeslotsFiltered:
5              timeslotBoolVars.append(model.NewBoolVar(""))
6          else:
7              timeslotBoolVars.append(falseVar)
8
9          for i in range(len(orm.getTimeslots())):
10             if timeslotBoolVars[i] is not falseVar:
11                 model.Add(timeslotBoolVars[i] == sum([b[i] for b in timeslotBoolVarsAll]))
12
13     lesson.timeslotBoolVars = timeslotBoolVars
14 else:
15     lesson.timeslotBoolVars = timeslotBoolVarsAll[0]
```

Codeausschnitt 10: HardConstraints.py - createLessonTimeslotBoolHelperVariables, 2

Abschließend wird die finale Liste, dem `Lesson` Objekt hinzugefügt. Oder im Falle von einstündigen Lessons, für die der zweite Schritt nicht notwendig war, die im ersten Schritt erstellte Liste (Zeile 15).

Vorlesungs-Hilfsvariablen für Lehrende, *timeslotLectureBoolMap*
Für die Implementierung des Constraints MaxLecturesAsBlockTeacherConstraint (2.3.16), bei dem Blocklängen von Vorlesungen eingeschränkt werden, wird es später notwendig sein für jeden Lehrenden zu wissen ob er zu einem Timeslot eine Vorlesung hält oder nicht. Dies entspricht einer Sicht auf den Stundenplan des Lehrenden, die nur die Vorlesungen enthält. Dafür wird in der Funktion `createTeacherLectureAtTimeslotMap`, für jeden Lehrenden eine Map (in Python `Dictionary`) angelegt, die zu jedem Timeslot (`Key`) eine `BoolVar` liefert, die anzeigt ob der Lehrende zu diesem Timeslot eine Vorlesung hält oder nicht. Stellt man sich alle `BoolVar` Variablen dieser Map, hintereinander vor, erhält man eine Binärfolge der Länge aller Timeslots. Eine 1 gibt dabei das Stattfinden einer Vorlesung zu einem Timeslot an. In dieser Binärfolge, können später die Blöcke, also aufeinanderfolgende Einsen, erkannt werden.

Implementierung Alles Folgende, wird für jedes `Teacher` Objekt ausgeführt. Da in der vorigen Funktion bereits für jede Lesson und jeden Timeslot so eine `BoolVar` angelegt wurde, können diese hier direkt wiederverwendet werden. Dazu wird zunächst für jeden Timeslot eine Liste angelegt, die die zum Timeslot passenden `BoolVar` Variablen für alle Vorlesungen die der Lehrer gibt, enthalten soll. Diese Listen werden in einer Map *timeslotLecturesListTakePlaceMap* gespeichert. Dies ist in Listing 11 gezeigt. In Zeile 1-4, werden diese Listen angelegt. Anschließend wird über jeden Timeslot und alle Vorlesungs-Lessons des Lehrenden iteriert.

```
1  timeslotLecturesListTakePlaceMap = {}
2
3  for timeslot in orm.getTimeslots():
4      timeslotLecturesListTakePlaceMap[timeslot] = []
5
6  for timeslot in orm.getTimeslots():
7      for lesson in filter(lambda l: l.course.is_lecture, teacher.lessons):
8          timeslotLecturesListTakePlaceMap[timeslot].append(
9              lesson.timeslotBoolVars[timeslot.id - 1]
10         )
```

Codeausschnitt 11: HardConstraints.py - createTeacherLectureAtTimeslotMap, Teil 1

In Zeile 8-10, werden die angelegten Listen befüllt. Gibt der Lehrende zum Beispiel drei Vorlesungen, die alle einen Timeslot belegen, so enthält die Map anschließend für jeden Timeslot eine Liste mit drei `BoolVar` Variablen, die für die drei Vorlesungen angeben, ob sie zu diesem Timeslot stattfinden.

Gesucht ist nun jeweils eine Variable die anzeigt ob mindestens eine der Variablen in der Liste den Wert True angenommen hat. Diese neuen Variablen werden jeweils *tOccupied* genannt, da sie für einen Timeslot angeben, ob dieser mit einer Vorlesung

besetzt ist. Dies ist in Listing 12 gezeigt. Die Wertzuweisung erfolgt durch zwei Constraints. In Zeile 4 wird bestimmt, dass wenn der Wert von *tOccupied* `True` ist, mindestens eine der Vorlesungs-Variablen wahr sein müssen. Die Umkehrung folgt in Zeile 5. Hat *tOccupied*, den Wert `False`, müssen alle Vorlesungs-Variablen den Wert `False` (0) haben. Also ist auch deren Summe 0.

```
1 for t in orm.getTimeslots():
2     tOccupied = model.NewBoolVar("")
3
4     model.AddBoolOr(timeslotLecturesListTakePlaceMap[t]).OnlyEnforceIf(tOccupied)
5     model.Add(sum(timeslotLecturesListTakePlaceMap[t]) ==
      ↪  0).OnlyEnforceIf(tOccupied.Not())
6
7     teacher.timeslotLectureBoolMap[t] = tOccupied
```

Codeausschnitt 12: HardConstraints.py - createTeacherLectureAtTimeslotMap, Teil 2

3.4.3 TeacherTimeConstraint

Dieser Abschnitt erläutert wie das TeacherTimeConstraint (2.3.2) umgesetzt wird. Für jeden Lehrenden soll zu einem Timeslot nie mehr als eine Veranstaltung stattfinden. Ausnahme bilden Lessons, die mit dem LessonsAtSameTimeConstraint verknüpft sind. Die Umsetzung ist sehr übersichtlich.

```
1 for teacher in orm.getTeachers():
2     model.AddAllDifferent(set(flatMap(lambda l: l.timeVars, teacher.lessons)))
```

Codeausschnitt 13: HardConstraints.py - addTeacherTimeConstraints

Alle Lessons die ein Lehrender gibt, sind jeweils als Liste abrufbar. (*teacher.lessons*) Nun muss lediglich sichergestellt werden, dass die Zeitpunkte dieser Lessons alle unterschiedlich sind.

Die Hilfsfunktion `flatMap`, die in der Datei ***HelperFunctions.py*** definiert ist, fügt hier die *timeVars* Listen aller Lessons des Lehrenden aneinander. Nun haben wir eine Liste mit allen *timeVar* Variablen aller Lessons eines Lehrenden. Solche Lessons, die zeitgleich stattfinden sollen, teilen sich ja ihre *timeVar* Variablen welche diesen Zeitpunkt angeben. Diese Duplikate können durch die `set()` Funktion eliminiert werden. Zu guter letzte stellt das `AllDifferent` Constraint sicher, dass die Werte aller dieser *timeVar* Variablen und damit die Zeitpunkte an denen die Lessons stattfinden unterschiedlich sind.

3.4.4 SemesterGroupTimeConstraints und PartSemesterGroupTimeConstraint

Genau wie Lehrende, können auch Semestergruppen normalerweise nicht zeitgleich an mehreren Veranstaltungen teilnehmen. Für alle Lessons bei denen das *whole_semester_group* Flag gesetzt ist, funktioniert die Umsetzung in der Funktion `addSemesterGroupTimeConstraints` dementsprechend genau wie bei Lehrenden. Allerdings gibt es eine

Ausnahme. Bei Lesson bei denen das Flag nicht gesetzt ist, sollen diese auch gleichzeitig stattfinden können, wenn mehrere Räume zur Verfügung stehen und diese von verschiedenen Lehrenden gehalten werden. Es könnten so allerdings auch Stundenpläne entstehen bei denen es einzelnen Teilgruppen von Semestergruppen nicht möglich ist alle Lessons zu besuchen. Da dies möglichst ausgeschlossen werden soll, ist dieses Constraint etwas aufwendiger.

Die folgenden Codeausschnitte werden jeweils für alle Semestergruppen ausgeführt. Zunächst zu den Lessons an denen die gesamte Semestergruppe teilnimmt. Für sie gilt, dass sie nie gleichzeitig stattfinden dürfen. Dies wird genauso umgesetzt wie schon für die Lehrenden. Dafür wird zuerst eine Liste mit den Lessons erstellt, an denen die gesamt Gruppe teilnimmt und ein `AddAllDifferent` Constraint hinzugefügt damit alle Timeslots die diese Lessons belegen unterschiedlich sind:

```
1 lessonsWithWholeSG = [l for l in semesterGroup.getLessons() if l.whole_semester_group]
2 model.AddAllDifferent(set(flatMap(lambda l: l.timeVars, lessonsWithWholeSG)))
```

Codeausschnitt 14: HardConstraints.py - addSemesterGroupTimeConstraints, Teil 1

Für alle Lessons an denen nur ein Teil der Semestergruppe teilnimmt, gestaltet sich die Umsetzung des Constraint etwas umständlicher. Es sollen beliebig viele Lessons desselben Kurses gleichzeitig stattfinden können, wenn keine Lesson eines anderen Kurses zur gleichen Zeit stattfindet. Finden Lessons mehrerer Kurse gleichzeitig statt, soll nur jeweils eine Lesson eines Kurses stattfinden. Dies verhindert zum Beispiel Konstellationen bei denen alle Lessons eines Kurses gleichzeitig stattfinden und noch Lessons anderer Kurse. In solchen Fällen könnten Studenten nicht an allen Lessons teilnehmen. Es könnte aber auch dazu kommen, dass eine mehrstündige Lesson zusammen mit allen einstündigen Lessons eines anderen Kurses nacheinander stattfinden. Auch in diesem Fall könnten einige Studenten nicht an allen Lessons teilnehmen. In Tabelle 9 sind diese beiden Fälle dargestellt.

Montag	Dienstag
	Kurs C L1 Kurs D L1
Kurs A L1 Kurs A L2 Kurs B L1 Kurs A L3	Kurs C L1 Kurs D L2
	Kurs C L1 Kurs D L3
Kurs B L2	
Kurs B L3	

Tabelle 9: Zeitkonflikte bei Lessons mit TeilSemestergruppen

Am Montag ist der Fall mit einstündigen Lessons. Studenten die in Lesson 1 von Kurs B sind, hätten gleichzeitig Kurs A und B. Am Dienstag findet Kurs C statt, bei dem eine Lesson über drei Timeslots geht. Kurs D hat zwar einstündige Lessons, diese finden aber alle zeitgleich mit der langen Kurs C Lesson statt. Studenten die in Lesson 1 von Kurs C sind könnten einen der beiden Kurse nicht belegen. Um diese zweite Konstellation

auszuschließen, sollen mehrstündige Lessons generell nicht mit Lessons anderer Kurse gleichzeitig stattfinden können.

Zunächst soll für jede Lesson an der nur ein Teil der SemesterGruppe teilnimmt, gelten, dass diese nicht zeitgleich mit einer Lesson für die gesamte SemesterGruppe stattfindet. Dies bedeutet ihre *timeVars* müssen unterschiedlich zu allen *timeVars* der Lessons mit gesamter Gruppe sein. Dies ist in folgendem Codeabschnitt umgesetzt. Es ähnelt stark dem bisherigen Constraint, nur werden jeweils die *timeVars* einer Lesson mit TeilSemesterGruppe der *timeVar* Liste mit der ganzen Gruppe hinzugefügt.

```
1 lessonsWithPartSG = list(filter(lambda l: not l.whole_semester_group,
  ↪  semesterGroup.getLessons()))
2 for lesson in lessonsWithPartSG:
3     model.AddAllDifferent(set(flatMmap(lambda l: l.timeVars, lessonsWithWholeSG +
  ↪  [lesson])))
```

Codeausschnitt 15: HardConstraints.py - addSemesterGroupTimeConstraint, Teil 2

Nun zu den oben erwähnten Einschränkungen, dass nie mehr als zwei Lessons verschiedener Kurse und nie mehrere mehrstündige Lessons, gleichzeitig stattfinden dürfen, damit Konflikte für einzelne Studierende verhindert werden. Diese werden für jeden Timeslot einzeln hinzugefügt. Folgendes wird also für jeden Timeslot wiederholt. Für jeden Kurs mit Lessons an denen nur ein Teil der Semestergruppe teilnimmt, wird eine `BoolVar` angelegt, die angibt ob der Kurs zu diesem Timeslot stattfindet. Dazu werden die `BoolVars` verwendet die für jede Lesson und jeden Timeslot angelegt wurden (*timeslotBoolVars*). Für die neu angelete `BoolVar` für jeden Kurs soll gelten:

$$courseTakePlace \iff lesson_1 \vee lesson_2 \vee ... \tag{17}$$

Gemeint ist, dass ermittelt wird, ob ein Kurs zu einem Zeitpunkt stattfindet, indem für alle seine Lessons geschaut wird, ob diese zu dem Zeitpunkt stattfinden. Dies wird dann in einer Variablen zusammengefasst. Dies wird mir dem Constraint `AddMaxEquality` (Listing 16, Zeile 11) erreicht, bei dem, dem ersten Argument der Maximalwert aller Elemente im zweiten (Listen-)Argument zugewiesen wird. Dies funktioniert auch mit boolschen Variablen und man spart sich so den Umweg über zwei Constraints die man normalerweise für die Zuweisung von boolschen Variablen benötigt. Die Liste *coursesPartSGLessonsBoolVars* der Variablen der einzelnen Lessons des Kurses wird dafür zuvor in Zeile 4, aufgebaut. Es werden jeweils die *timeslotBoolVars* der Lessons des Kurses ausgewählt, die Teilgruppen-Lessons sind.

```
for timeslot in orm.getTimeslots():
    courseTakePlace = list(map(lambda c: model.NewBoolVar(""), parallelCourses))
    for i in range(len(parallelCourses)):
        coursesPartSGLessonsBoolVars = [l.timeslotBoolVars[timeslot.id - 1] for l in
        ↪  parallelCourses[i].lessons if not l.whole_semester_group]
        model.AddMaxEquality(courseTakePlace[i], coursesPartSGLessonsBoolVars)
```

Codeausschnitt 16: HardConstraints.py - addSemesterGroupTimeConstraint, Teil 3

parallelCourses ist hier eine Liste mit allen Kursen, die Lessons enthalten, an denen nicht die gesamte Semestergruppe teilnimmt. Nun soll zwischen zwei Fällen unterschieden werden. Zu jedem Zeitslot sollen entweder die Lessons eines einzigen Kurses gleichzeitig stattfinden (Fall 1), oder einzelne Lessons mehrerer Kurse, dann aber soll keine Lessons dabei sein, die mehr als einen Timeslot belegt (Fall 2). Dies wird erreicht indem eine `BoolVar` angelegt wird, deren Wert aber nicht vorgegeben wird. Sie dient dazu, ein „entweder oder“ zweier Ausdrücke umzusetzen. Hat die Variable den Wert True, soll Fall 1 gelten, sonst Fall 2. Durch die in Listing 16 hinzugefügten Constraints, enthält die Liste *courseTakePlace* nun `BoolVar` Variablen, die angeben, ob die Kurse im aktuellen Timeslot (der Iteration) stattfinden. Die Summe dieser Variablen ist also die Anzahl Kurse die stattfinden. Diese soll bei Fall 1, kleiner gleich 1 sein. Bei Fall 2 soll die Anzahl Kurse gleich der Anzahl Lessons sein, da je Kurs nur eine Lesson stattfinden soll. Die Anzahl Lessons wird mit den *timeslotBoolVars* der Lessons gezählt. Außerdem soll für Fall 2, keine Lesson stattfinden die eine *timeslot_size* größer 1 hat. Dazu wurde eine Liste mit allen Lessons angelegt die länger als ein Timeslot sind.

```
boolV = model.NewBoolVar("")

# Case 1:
# Case for Lessons of only one Course.
model.Add(sum(courseTakePlace) <= 1).OnlyEnforceIf(boolV)

# Case 2:
# Case for parallel Lessons of more than one Course. Then only one Lesson per Course.
model.Add(sum([l.timeslotBoolVars[timeslot.id - 1] for l in lessonsWithPartSG])
          == sum(courseTakePlace)).OnlyEnforceIf(boolV.Not())
# And make sure that there is no Lesson with bigger timeslot_size than one.
model.Add(sum([l.timeslotBoolVars[timeslot.id - 1] for l in multiBlockLessons])
          == 0).OnlyEnforceIf(boolV.Not())
```

Codeausschnitt 17: HardConstraints.py - addSemesterGroupTimeConstraint, Teil 4

3.4.5 RoomTimeConstraints

Bei der Implementierung des Constraints RoomTimeConstraint (2.3.1), liegt die Schwierigkeit darin begründet, dass verschiedene Lessons nur dann nicht gleichzeitig stattfinden dürfen, wenn sie im selben Raum stattfinden beziehungsweise nicht im selben Raum stattfinden wenn sie zeitgleich stattfinden. Dieses Constraint vergrößert das OR-Tools

Modell erheblich, da die Anzahl hinzugefügter OR-Tools-Constraints - und gegebenenfalls hinzugefügte Variablen - quadratisch mit der Anzahl der Lessons im Stundenplan steigt ($\mathcal{O}(n^2)$). Die tatsächliche Anzahl ist jedoch geringer, da die Constraints nur für Lessons hinzugefügt werden die auch tatsächlich im selben Raum stattfinden können.

Es wird für alle Paare aus zwei Lessons ermittelt, ob diese im selben Raum stattfinden können. Ist dies der Fall, wird eine `BoolVar` ***sameRoom*** erstellt, die angibt ob sie tatsächlich im selben Raum stattfinden. Außerdem soll das RoomTimeConstraint nicht gelten, wenn die Lessons mit dem LessonsAtSameTimeConstraint (2.3.10) verknüpft sind. Dies ist hier in Codeausschnitt 18 gezeigt.

```
for i in range(len(orm.getLessons())):
    for j in range(i + 1, len(orm.getLessons())):  # Über Paare iterieren:
        lesson_i, lesson_j = orm.getLessons()[i], orm.getLessons()[j]

        room_intersection =
            set(lesson_i.course.possible_rooms) & set(lesson_j.course.possible_rooms)

        if room_intersection and lesson_j not in lesson_i.lessons_at_same_time:
            sameRoom = model.NewBoolVar("")
            model.Add(orm.getLessons()[i].roomVar ==
             ↪  orm.getLessons()[j].roomVar).OnlyEnforceIf(sameRoom)
            model.Add(orm.getLessons()[i].roomVar !=
             ↪  orm.getLessons()[j].roomVar).OnlyEnforceIf(sameRoom.Not())
```

Codeausschnitt 18: HardConstraints.py - addRoomTimeConstraints, Teil 1

Bei dem Constraint, dass die Lessons zu unterschiedlichen Zeiten stattfinden wenn der Raum gleich ist wird zwischen zwei Fällen unterschieden. Je nachdem ob beide Lessons mehrstündig sind oder nicht, werden zwei verschiedene Methoden angewendet. Ist mindestens eine der Lessons nur einen Timeslot lang, wird für jede ***timeVar*** der beiden Lessons mittels einem linearem Constraint angegeben, dass diese unterschiedlich sein müssen, wenn der Raum gleich ist. Die Anzahl der hinzugefügten Constraints ist dabei die größte Anzahl an Timeslots den eine der beiden Lessons belegt. Um nicht erst ermitteln zu müssen welche und ob eine der Lessons mehrere Timeslots belegt, wird dies trotzdem mit einer verschachtelten Schleife ausgeführt (Zeile 1 und 2, Codeausschnitt 19).

```
1 for k in range(lesson_i.timeslot_size):
2     for l in range(lesson_j.timeslot_size):
3         model.Add(lesson_i.timeVars[k] != lesson_j.timeVars[l]).OnlyEnforceIf(sameRoom)
```

Codeausschnitt 19: HardConstraints.py - addRoomTimeConstraints, Teil 2

Für den Fall, dass sich beide Lessons über mehrere Timeslots erstrecken, wird die Möglichkeit in OR-Tools genutzt, Intervalle angeben und deren Überschneiden verbieten zu können. Falls noch nicht geschehen, werden für die Lessons die `Interval` Variablen angelegt. Diese werden gespeichert um in der Kombination mit anderen Lessons wieder-

verwendet werden zu können. Hierbei ist zu beachten, dass die Intervalle als Zeitraum zwischen Zeitpunkten gedacht ist. Da wir aber einen Start- und einen Endzeitraum angeben, müssen die Intervalle jeweils um eins größer sein als sich aus den IDs der Timeslots ergibt. Beispiel: Angenommen eine Lesson beginnt im Timeslot 2 und endet im Timeslot 3. Die andere Lesson beginnt in Timeslot 3 und endet in Timeslot 4. Das `NoOverlap` Constraint der OR-Tools Bibliothek würde dies erlauben, da gewissermaßen angenommen wird, '3' sei ein Zeitpunkt und die Lessons würde sich daher nicht überschneiden. Um das `NoOverlap` Constraint dennoch verwenden zu können, wird der Beginn der Intervalle jeweils um Eins verringert. Die Intervalle gingen im obigen Beispiel dann von 1 bis 3 und von 2 bis 4. Hier greift das `NoOverlap` Constraint und würde diese Intervalle verbieten.

Zunächst die Erstellung der Intervalle, Ausschnitt 20. Dies wird jeweils für beide Lessons eines Paares ausgeführt. Der Beginn der Intervalle stellt die erste *`timeVar`* Variable der Lessons dar, aber um Eins vermindert. Dazu wird eine Variable erstellt die immer den Wert $timeVar - 1$ hat.

```
1  dummyVar = model.NewIntVar(0, 30, "")
2  model.Add(dummyVar == lesson.timeVars[0] - 1)
3  lesson.timeInterval = model.NewIntervalVar(
4      dummyVar,                                    # Lesson Start
5      lesson.timeslot_size,                        # Länge der Lesson
6      lesson.timeVars[lesson.timeslot_size - 1], "")  # Lessons Ende
```

Codeausschnitt 20: HardConstraints.py - addRoomTimeConstraints, Teil 3

Anschließend wird das Intervall erzeugt. Als Start wird die erzeugte Variable angegeben, danach die Länge der Lesson und dann die Variable für den letzten Timeslot den die Lesson belegt. Die Intervallvariablen funktionieren so, dass für die drei Argumente a, b, c immer gelten muss $a + b == c$. Diese Eigenschaft wird hierbei jedoch nicht genutzt, da sie ohnehin bereits durch die Erstellung der *`timeVars`* Variablen für ebendiese zutrifft.

Darum, dass sich die Lessons nun nicht mehr überschneiden kümmert sich die OR-Tools Bibliothek automatisch durch das `NoOverlap` Constraint.

```
1  model.AddNoOverlap([lesson_i.timeInterval, lesson_j.timeInterval]).OnlyEnforceIf(sameRoom)
```

Codeausschnitt 21: HardConstraints.py - addRoomTimeConstraints, Teil 4

Der Umweg mit der Unterschiedung zwischen den beiden Fällen wird hier gegangen, da angenommen wird, dass das Einhalten eines `NoOverlap` Constraints günstiger im Sinne der Laufzeit ist als eine größere Anzahl linearer Constraints, aber aufwendiger als nur ein oder zwei lineare Constraints.

3.4.6 StudyDayConstraints

Alle Lehrenden die denen ein Studientag zusteht haben für diesen eine Erst- und eine Zweitwahl. Dass die Erstwahl präferiert wird, ist in den SoftConstraints umgesetzt. An

dieser Stelle wird nur sichergestellt, dass an mindestens einem der Auswahltage für den Lehrenden keine Lessons stattfinden. Sind Erstwahl und Zweitwahl der gleiche Wochentag, ist zwingend an diesem einen Tag frei.

Zunächst werden für die Erst- und Zweitwahl eine `BoolVar` angelegt. Durch diese ist angegeben, welche der Wahlmöglichkeiten tatsächlich umgesetzt ist. Nun muss sichergestellt werden, dass an dem Tag für den die Variable True ist, keine Lessons für den Lehrenden stattfinden. Dazu wird für jede Lesson die der Lehrende gibt, ein Constraint hinzugefügt. An dieser Stelle wird die Hilfsvariable *weekdayVar* verwendet, die angibt an welchem Wochentag eine Lesson stattfindet.

```
1 for lesson in teacher.lessons:
2     model.Add(studyDay1ID != lesson.weekdayVar).OnlyEnforceIf(teacher.studyDay1BoolVar)
```

Codeausschnitt 22: HardConstraints.py - addStudyDayConstraints, Teil 1

In Codeausschnitt 22 wird dies beispielhaft für die Erstwahl gezeigt. Die Variable *studyDay1ID* in Zeile 2 gibt dabei die ID des Wochentages der Erstwahl an, also zum Beispiel „2 “für Dienstag.

Dass mindestens ein Studientag auch tatsächlich stattfindet ist über ein einfaches `Or`-Constraint umgesetzt, Ausschnitt 23.

```
1 model.AddBoolOr([teacher.studyDay1BoolVar, teacher.studyDay2BoolVar])
```

Codeausschnitt 23: HardConstraints.py - addStudyDayConstraints, Teil 1

3.4.7 RoomNotAvailableConstraints

Hier wird die Implementierung des Constraints NotAvailableRoomTimeConstraint (2.3.6), erläutert. Für jeden Raum können Timeslots angegeben werden, zu denen der Raum, für die Lessons aus dem Stundenplan nicht zur Verfügung steht. Ist die Liste leer, ist der Raum immer verfügbar. Dies soll umgesetzt werden, indem für jeden Raum und jede Lesson, die in dem Raum stattfinden könnte, eine `BoolVar` Variable angelegt wird, die angibt, ob die Lesson tatsächlich in dem Raum stattfindet[16]. Der Name dieser Variablen ist jeweils *inRoom*. Ist der Wert dieser Variablen wahr, darf die Lesson nicht zu einem der Zeitpunkte stattfinden, zu denen der Raum nicht verfügbar ist. Die Implementierung ähnelt daher der formalen Erklärung des Constraints in Abschnitt 2.3.6.

Codeausschnitt 24, zeigt die Implementierung. Es ist der gesamte, relevante Code der Funktion `addRoomNotAvailableConstraints` enthalten. Es wird über alle Räume, die mindestens einen, nicht-verfügbaren Zeitpunkt haben und dann jeweils über alle Lessons, die in dem Raum stattfinden können, iteriert. (Zeile 1-3) Die Belegung der *inRoom* Variable erfolgt in der üblichen Form (3.4. Absatz `BoolVar` Variablen mit der Erfüllung von Ausdrücken belegen) und mithilfe der Raum ID und der Variablen *roomVar* der Lesson, welche den Raum, in dem die Lesson stattfindet, durch seine ID, angibt. (Zeile 4-6)

[16] Dies ist notwendig, da für Lessons ja auch mehrere Räume zur Auswahl stehen können.

Anschließend muss nur noch, für jeden der nicht verfügbaren Timeslots sichergestellt werden, dass die Lesson nicht zu ihnen stattfindet, wenn *inRoom* wahr ist. Dies geschieht über den Vergleich der Werte der *timeVar* Variablen, der Lesson, und den IDs der Timeslots. (Zeile 9)

```
1 for room in orm.getRooms():
2     if len(room.not_available_timeslots) > 0:
3         for lesson in filter(lambda l: room in l.course.possible_rooms, orm.getLessons()):
4             inRoom = model.NewBoolVar("")
5             model.Add(lesson.roomVar == room.id).OnlyEnforceIf(inRoom)
6             model.Add(lesson.roomVar != room.id).OnlyEnforceIf(inRoom.Not())
7             for timeslot in room.not_available_timeslots:
8                 for timeVar in lesson.timeVars:
9                     model.Add(timeVar != timeslot.id).OnlyEnforceIf(inRoom)
```

Codeausschnitt 24: HardConstraints.py - addRoomNotAvailableConstraints

Anschließend wird für jede Kombination aus nicht verfügbarem Timeslot und *timeVar* der Lesson ein lineares Constraint hinzugefügt, dass diese Timeslot IDs unterschiedlich sein müssen, vorausgesetzt die Lesson findet in dem Raum statt. Zeile 7 bis 9.

3.4.8 CourseAllInOneBlockConstraints

Das CourseAllInOneBlockConstraint (2.3.11), gibt für einen Kurs an, dass alle Lessons dieses Kurses als Block direkt nacheinander und innerhalb eines Tages stattfinden sollen. Die Reihenfolge soll beliebig bleiben. Die Größe dieses Blocks ergibt sich aus der Summe der Längen der einzelnen Lessons.

Um die Lessons nacheinander stattfinden zu lassen, wird erneut ein Intervall Constraint verwendet. Diesmal wird jedoch tatsächlich von dem Constraint dahinter Gebrauch gemacht. Es werden Variablen für den Start und das Ende des gesamten Blocks erstellt. Die Funktion `AddMinEquality` (Listing 25, Zeile 7) weist dem ersten Argument (einer OR-Tools Variablen) den kleinsten Wert aus der Liste (zweites Argument), zu. Das zweite Argument ist in diesem Fall eine Liste aller *timeVars* Variablen aller Lessons des Kurses, also alle *timeVar* Variablen die genau aufeinander folgende Werte annehmen sollen. `AddMaxEquality` funktioniert entsprechend. Für das `IntervalConstraint` werden die Start- und Endvariablen zusammen mit der Gesamtdauer des Blockes als Intervallgröße verwendet. Man kann sich vorstellen, wenn ein Zusammenhang dieser drei Werte hergestellt wird, so dass $start + blocksize - 1 == ende$ gilt dass alle Lessons direkt aufeinanderfolgen müssen, da der Start der ersten und das Ende der letzten Lesson, genau soweit voneinander erntfernt sind, wie die Lessons insgesamt an Zeit belegen.[17]. Diese Implementierung entspricht zudem sehr, der formalen Veranschaulichung des Constraints in Abschnitt 2.3.11.

In Zeile 10, wird dies als `IntervalVar` hinzugefügt. Diese Intervall Constraints, sichern den linearen Zusammenhang der drei Parameter `a`, `b`, `c` zu, so dass $a + b == c$

[17] Dies gilt insbesondere deswegen, weil die Lessons eines Kurses nicht gleichzeitig stattfinden können. Es wird daher auch angenommen, dass das CourseAllInOneBlockConstraint, nie zusammen mit Lessons verwendet wird, an denen nicht die gesamte Semestergruppe teilnimmt (PartSemesterGroupConstraint).

gilt.

```
blocksize = sum(map(lambda l: l.timeslot_size, course.lessons))
    if blocksize > 1:
        lastStartPossible = TIMESLOTS_PER_DAY + 1 - blocksize

        minVar = model.NewIntVar(1, 30, "")
        maxVar = model.NewIntVar(1, 30, "")
        model.AddMinEquality(minVar, course.getAllTimeVars())
        model.AddMaxEquality(maxVar, course.getAllTimeVars())

        model.NewIntervalVar(minVar, blocksize - 1, maxVar, "")
```

Codeausschnitt 25: HardConstraints.py - addCourseAllInOneBlockConstraints, Teil 1

Es muss noch garantiert werden, dass die Lessons auch alle am selben Tag stattfinden. Es wird wieder das gleiche Verfahren wie auch schon beim Erstellen der Basisvariablen in Abschnitt 3.4.1 Zeit- und Raumvariablen verwendet, bei dem der Blockanfang nicht später am Tag als die rechnerisch letztmögliche Stunde beginnen darf. Das Problem an dieser Stelle ist, den Timeslot des Tages zu ermitteln, zu dem der Block beginnt. Die *hourNumberVar* der Lessons kann nicht benutzt werden, da nicht klar ist welche Lesson zuerst stattfindet. Daher wird der Timeslot über ein Modulo Constraint ermittelt. Hier hilft die Einschränkung, dass die Blockgröße immer größer eins ist. (Andernfalls würde das gesamte Constraint keinen Sinn ergeben und dies wird bei der Implementierung auch bereits zu Beginn abgefragt) Es wird zunächst eine Variable (*minHourNumberVar* angelegt, die die Nummer des ersten Timeslots des Blockes an einem Tag enthalten soll. Der Wertebereich wird direkt so eingegrenzt, dass sie höchstens die Nummer des errechneten letztmöglichen Timeslots eines Tages annehmen kann. Also bei sechs Timeslots pro Tag ein Wert zwischen eins und $sechs + 1 - blockgröße$. Da die IDs aller Timeslots zwingend aufeinanderfolgend sind, kann der Wert der *minHourNumberVar* nun mittels des Modulo Constraints zugewiesen werden.

$$minHourNumberVar == minVar \bmod TIMESLOTS\_PER\_DAY \tag{18}$$

Der Fall der letzten Stunde, also $minVar == TIMESLOTS\_PER\_DAY$, bei dem das Ergebnis null anstelle der Nummber der letzten Stunde wäre, ist ausgeschlossen, da die Blockgröße mindestens eins ist.

```
minHourNumberVar = model.NewIntVar(1, lastStartPossible, "")
model.AddModuloEquality(minHourNumberVar, minVar, TIMESLOTS_PER_DAY)
```

Codeausschnitt 26: HardConstraints.py - addCourseAllInOneBlockConstraints, Teil 2

Zusätzlich sollen alle Lessons innerhalb eines Blockes immer im gleichen Raum stattfinden. Die Implementierung geschieht über simple lineare Constraints die angeben, dass die Werte der Raumvariablen der Lessons im Block gleich sein sollen, Ausschnitt 27 .

```
for i in range(1, len(course.lessons)):
    model.Add(course.lessons[i - 1].roomVar == course.lessons[i].roomVar)
```

Codeausschnitt 27: HardConstraints.py - addCourseAllInOneBlockConstraints, Teil 3

3.4.9 ConsecutiveLessonsConstraints

Das ConsecutiveLessonsConstraint (2.3.19), ermöglicht die Angabe von Lessons, die direkt nach einer anderen Lesson stattfinden sollen. Für die Implementierung wird benötigt:

- Eine Variable *lesson*, die zur Iterierung über alle Lesson Objekte genutzt wird.
- Die Liste *lessons_consecutive*, die in jedem Lesson Objekt vorhanden ist und die Folgelessons enthält.
- Eine Variable *consecutiveLesson*, die zur Iterierung über alle Folgelessons, der *lessons_consecutive* Liste, dient.
- Die Variable *weekdayVar* ist in jedem Lesson Objekt enthalten. Eine OR-Tools `IntVar` Variable die angibt an welchem Wochentag die Lesson stattfindet.
- Die Liste *timeVars* von jeder Lesson, die die Zeitpunkte der einzelnen Timeslots angibt, die die Lesson belegt. Sie enthält OR-Tools `IntVar` Variablen die die ID des jeweiligen Timeslots angeben.

In Lising 28 ist der gesamt notwendige Code für dieses Constraint enthalten. Es wird über alle Lessons und dann jeweils über alle Folgelessons der einzelnen Lessons iteriert. Es wird dann jeweils ein Vergleich zwischen Startlesson und Folgelesson hergestellt. Einmal über die *weekdayVar* um den gleichen Wochentag sicherzustellen (Zeile 3) und einmal über die letzte *timeVar* der Startlesson (*timeVars[-1]*) und die erste *timeVar* der Folgelesson, um das direkte Aufeinanderfolgen zu garantieren (Zeile 4).

```
1 for lesson in orm.getLessons():
2     for consecutiveLesson in lesson.lessons_consecutive:
3         model.Add(lesson.weekdayVar == consecutiveLesson.weekdayVar)
4         model.Add(lesson.timeVars[-1] + 1 == consecutiveLesson.timeVars[0])
```

Codeausschnitt 28: HardConstraints.py - addConsecutiveLessonsConstraints

3.4.10 MaxLessonsPerDayTeacherConstraints

Bei diesem Constraint, soll die Anzahl mit Veranstaltungen belegter Timeslots pro Tag, jedes Lehrenden, begrenzt werden. Um das MaxLessonsPerDayTeacherConstraint (2.3.14) zu implementieren ist es zunächst notwendig, die Lessons zu ermitteln, welche tatsächlich gezählt werden sollen. Das sind für jeden Lehrenden alle Lessons die nicht in einem LessonsAtSameTimeConstraint enthalten sind und von den Lessons in einem LessonsAtSameTimeConstraint jeweils die Längsten. Diese Unterscheidung ist notwendig, um Lessons die gleichzeitig im Stundenplan eingetragen sind nicht doppelt zu zählen. Zunächst wird erläutert wie diese Lessons ausgewählt werden.

Alles folgende wird jeweils für jeden Lehrenden ausgeführt. Die Variable, die genutzt wird um über alle Lehrenden zu iterieren heißt *teacher*. Sie enthält das jeweilige `Teacher` Objekt.

Zunächst werden die `Lesson` Objekte ermittelt, die nicht Teil eines LessonsAtSameTimeConstraints sind. Dies ist immer dann der Fall, wenn die Liste *lessons_at_same_time* eines `Lesson` Objektes leer ist. Zeile 1 von Listing 29 zeigt wie die Liste aller Lessons des Lehrenden mit diesem Kriterium gefiltert wird. Das Ergebnis wird in der Menge *lessonsForTeacher* gespeichert.

Die Skriptdatei ***ORM.py*** bietet eine Methode, die alle Mengen von Lessons liefert, die jeweils durch das LessonsAtSameTimeConstraint zeigtleich stattfinden. Aus jeder dieser Mengen werden jeweils die Lessons ausgewählt, an denen der Lehrende teilnimmt. Aus diesen wird dann die längste Lesson ausgewählt und der Menge *lessonsForTeacher* hinzugefügt werden. Dies zeigen die Zeilen 3 bis 6 des Listings 29.

```
1 lessonsForTeacher = set([x for x in teacher.lessons if not x.lessons_at_same_time])
2
3 for sameTimeSet in orm.getLessonsAtSameTimeSets():
4     sameTimeLessons = [x for x in sameTimeSet if x in teacher.lessons]
5     if sameTimeLessons:  # Only if list not empty and only add longest Lesson.
6         lessonsForTeacher.add(max(sameTimeLessons, key=lambda l: l.timeslot_size))
```

Codeausschnitt 29: HardConstraints.py - addMaxLessonsPerDayTeacherConstraints

Nun muss für den Lehrenden lediglich für jeden Wochentag des Stundenplans dem `CpModel` ein Constraint hinzugefügt werden, das die Anzahl an belegten Timeslots für den Lehrenden zählt und diese Summe begrenzt. Dafür wird die zuvor erstellte Menge *lessonsForTeacher* verwendet. Die Hilfsvariable *weekdayBoolVars* jedes `Lesson` Objektes ist eine Liste mit `BoolVar` Instanzen. Für jeden Wochentag gibt es in dieser Liste ein Boolean, dass nur wahr ist, wenn die Lesson an diesem Wochentag stattfindet. Diese Variable wird jeweils ausgewählt und mit der Länge der Lesson multipliziert. Das Produkt ist 0, für alle Lessons die nicht am jeweiligen Wochentag stattfinden. Dies ist in Zeile 2 von Listing 30 gezeigt. Python und die OR-Tools Bibliothek erlauben es, diese Produkte von je einem Skalar und einer `BoolVar` als Liste zusammenzufügen, die Summe zu bilden und dies als Argument beim Hinzufügen des Constraints aufzufassen[18]. Das eigentliche lineare Constraint besagt dann, dass diese Summe kleiner-gleich der maximalen Anzahl belegter Timeslots des Lehrenden pro Tag ist.

```
1 for day in range(orm.WEEKDAYS):
2     model.Add( sum(list(map(lambda l: l.timeslot_size * l.weekdayBoolVars[day],
      ↪  lessonsForTeacher))) <= teacher.max_lessons_per_day )
```

Codeausschnitt 30: HardConstraints.py - addMaxLessonsPerDayTeacherConstraints

[18] Dies funktioniert auf diese Weise natürlich nur, weil die `IntVar` Klasse (beziehungsweise dessen Basisklasse `LinearExpr`), Pythons `Sum` Funktion und den Multiplikations-Operator überlädt, sodass an jeder Stelle $int * BoolVar$ ein neues `LinearExpr` Objekt entsteht.

3.4.11 MaxLessonsPerDaySemesterGroupConstraints

Bei diesem Constraint, soll die Anzahl mit Veranstaltungen belegter Timeslots pro Tag, für jede Semestergruppe, begrenzt werden. Dabei stellen sich einige Probleme. Es gibt Kurse, an deren Lessons immer nur Teile einer Semestergruppe teilnehmen. Diese können auch zeitgleich stattfinden. Außerdem ist es sinnvoll, von diesen Lessons pro Tag nur eine zu zählen. Schließlich nimmt jeder einzelne Studierende für gewöhnlich nur an einer dieser Lessons teil. Wie auch im Abschnitt 2.3.4 PartSemesterGroupConstraint beschrieben, wird bei Teilgruppen-Kursen immer angenommen, dass vorgesehen ist, dass jeder Studierende nur an einer der Lessons der Kurses teilnehmen soll. Würde dennoch ein Kurs modelliert, bei dem dies anders vorgesehen ist, werden entsprechend für die Semestergruppe zu wenig Lessons an einem Tag gezählt.

Außerdem sollen wie bei der Implementierung des MaxLessonsPerDayTeacherConstraints, von Lessons die durch das LessonsAtSameTimeConstraint zeitgleich stattfinden, nur die längste gezählt werden.

Mögliche Ungenauigkeiten

Vorab sei gesagt, dass es bei der verwendeten Implementierung in zwei Spezialfällen zu Ungenauigkeiten kommen kann. Damit ist gemeint, dass zwei Lessons auch als zwei Lessons gezählt werden, obwohl die den gleichen Timeslot belegen. Dieser Timeslot wird dann quasi doppelt gezählt:

- Wenn von allen Lessons eines Teilgruppen-Kurses, einige, aber nicht alle, Teil eines LessonsAtSameTimeConstraints sind, werden von diesen Lessons die doppelt gezählt, die zufällig zeitgleich zu den Lessons in dem Constraint stattfinden.
- Auch wenn in einem LessonsAtSameTimeConstraint sowohl Lessons mit der ganzen und mit Teilgruppen enthalten sind, werden diese gegebenenfalls mehrfach gezählt.
- Wenn die Lessons eines Kurses mit Teilgruppen, unterschiedlich lang sind, wird für einige dieser Lesson eine falsche Lessonlänge gezählt. Dies ist der Fall, da für solche Kurse eine Lessonlänge gewählt und für jeden Tag gezählt wird, an dem der Kurs stattfindet.
- Wenn Lessons zweier verschiedener Kurse mit Teilgruppen zeitgleich stattfinden, werden beide gezählt.

Die ersten drei Konstellationen dieser Liste sind stark konstruiert und es ist unwahrscheinlich, dass diese in einem Stundenplan geplant werden. (Dennoch sind sie durch das gewählte Stundenplanmodell realisierbar, daher werden sie hier aufgeführt) Der vierte Fall ist jedoch sehr gut möglich und muss mit der gewählten Implementierung in Kauf genommen werden. Durch alle der genannten Ungenauigkeiten wird das MaxLessonsPerDaySemesterGroupConstraint jedoch niemals verletzt, da immer mehr Timeslots als gewünscht gezählt werden. Es werden jedoch gewisse Stundenplanlösungen ausgeschlossen, da so, wenn durch die Ungenauigkeit, die maximale Anzahl belegter Timeslots an einem Tag erreicht wird, ein Timeslot weniger für eine andere Lesson an dem Tag zur Verfügung steht. Die Wahrscheinlichkeit, dass aus diesem Grund kein Stundenplan gefunden werden

kann, ist verschwindend gering und könnte zudem nur bei einem Stundenplan mit sowieso schon grenzwertig hoher Stundenplanfülle (2.1, Fülle des Stundenplans) auftreten.

Implementierung

Bei der Implementierung muss zwischen Lessons aus Teilgruppen-Kursen und Kursen an denen ganze Semestergruppen teilnehmen unterschieden werden. Für beide Kategorien wird jeweils eine Liste mit zu zählenden Objekten erstellt. Für die Teilgruppen-Kurse werden jeweils Variablen erstellt, die angeben, ob an einem Wochentag eine (Teilgruppen-)Lesson des Kurses stattfindet. Für die anderen Kurse einfach eine Liste der zu zählenden Lessons. Am Ende wird für jeden Tag aus den beiden Listen ein Constraint gebaut, dass dem `CpModel` hinzugefügt wird. Dies ist dann sehr ähnlich wie bei der Implementierung des MaxLessonsPerDayTeacherConstraint in Abschnitt 3.4.10 MaxLessonsPerDayTeacherConstraints.

Alles folgende wird jeweils jür jede Semestergruppe ausgeführt. Der Name der Variablen, mit der über alle Semestergruppen iteriert wird ist *semesterGroup*.

Lessons für ganze Semestergruppen, Schritt 1

Es wird gezeigt, wie die zu zählenden Lesson mit ganzen Semestergruppen zusammengetragen werden. Genau wie bei der Implementierung des MaxLessonsPerDayTeacherConstraints, soll darauf geachtet werden, dass von Lessons die Teil eines LessonsAtSameTimeConstraints sind, nur die längste gezählt wird. Sollte die längste Lesson des Constraints aber eine Teilgruppen-Lesson sein, sollen alle Lessons des Constraints in diesem Schnitt ignoriert werden. Denn in dem Fall wird die Lesson ohnehin schon im zweiten Schritt gezählt.

Es wird ein leeres Mengenobjekt erzeugt und alle Lessons hinzugefügt, die keine Teilgruppen-Lessons sind und die nicht Teil eines LessonsAtSameTimeConstraints sind (Listing 31.

```
1 lessonsWithWholeSG = set()
2 for l in semesterGroup.getLessons():
3     if l.whole_semester_group and not l.lessons_at_same_time:
4         lessonsWithWholeSG.add(l)
```

Codeausschnitt 31: HardConstraints.py - MaxLessonsPerDaySemesterGroupConstraints 1

Anschließend werden die Lessons der LessonsAtSameTimeConstraints betrachtet. Das Skript ***ORM.py*** liefert dazu eine Liste alle Mengen von Lessons, die zeitgleich stattfinden sollen. Aus diesen Mengen sollen jeweils maximal eine Lesson ausgewählt werden, die mitgezählt werden soll. Dies müssen natürlich Lessons sein, an denen die aktuelle Semestergruppe überhaupt teilnimmt. Daher wird zunächst danach gefiltert, Listing 32 Zeile 2. Anschließend wird die längste der Lessons aus der Liste gewählt. Es wird als sekundärer Sortierschlüssel die Variable *whole_semester_group* der Lessons (in negierter Form) verwendet. Da im Anschluss geprüft wird, ob die ausgewählte, längste Lesson tatsächlich eine Lesson mit ganzen Semestergruppen ist, findet so keine Auswahl statt, wenn

unter den längsten Lessons des LessonsAtSameTimeConstraints eine Teilgruppen-Lesson ist. Dies verhindert, dass die längste Lesson des Constraints zweimal gezählt wird, da alle Teilgruppen-Lessons im zweiten Schritt separat gezählt werden. Ist die gefundene, längste Lesson jedoch eine Lesson mit ganzen Semestergruppen, wird sie der Liste der zu zählenden Lessons *lessonsWithWholeSG* hinzugefügt (Listing 32 Zeile 5 und 6).

```
1 for sameTimeSet in orm.getLessonsAtSameTimeSets():
2     sameTimeLessons = [x for x in sameTimeSet if x in semesterGroup.getLessons()]
3     if sameTimeLessons:
4         longestLesson = max(sameTimeLessons,
5                             key=lambda l: (l.timeslot_size, not l.whole_semester_group))
6         if longestLesson.whole_semester_group:
7             lessonsWithWholeSG.add(longestLesson)
```

Codeausschnitt 32: HardConstraints.py - MaxLessonsPerDaySemesterGroupConstraints 2

Lessons für Teilgruppen, Schritt 2

Für alle Teilgruppen-Lessons eines Kurses und jeden Wochentag soll ermittelt werden, ob so eine Lesson des Kurses an dem Wochentag stattfindet. Auf diese Weise werden pro Tag und Kurs nur maximal eine Teilgruppen-Lesson gezählt. Die Anweisungen des zweiten Schrittes sind in Listing 33 zu sehen. Es wird über alle Wochentage iteriert (Zeile 1). Für jeden Wochentag wird eine Liste *partSGCourses* angelegt (Zeile 2). In dieser Liste soll für jeden Teilgruppen-Kurs ein 2-Tupel gespeichert werden. Ein Element jedes Tupels enthält eine `BoolVar`, die angibt, ob eine Lesson des Kurses des zugehörigen Tupels an dem Tag stattfindet. Das andere Element ist die Länge einer der Teilgruppen-Lessons[19]. Zunächst werden die Lessons der einzelnen Teilgruppen-Kurse gesammelt (Zeile 3-4).

Die Variable, die angibt, ob mindestens eine Lesson des Kurses am jeweiligen Wochentag stattfindet, muss erst erstellt werden. Um sie mit den richtigen Werten zu füllen, ist wiederum ein OR-Tools Constraint nötig. In Zeile 6 wird diese Variable jeweils mit dem Namen *courseTakePlaceVar* erstellt. Der Wert dieser Variablen soll nun 1 sein, wenn mindestens eine der zuvor gesammelten Lessons am jeweiligen Wochentag stattfindet. Für jede Lesson einzeln, ist diese Information in der Liste *weekdayBoolVars* enthalten. Mit dem Index des jeweiligen Wochentages, wird aus dieser Liste die `BoolVar` des entsprechenden Wochentages ausgewählt. Das Constraint soll nun *courseTakePlaceVar* den Wert 1 zuweisen, wenn mindestens eine der *weekdayBoolVars* der Lessons den Wert 1 hat. Dazu wird das `AddMaxEquality` Constraint verwendet. Es stellt sicher, dass das ersten Argument (die Zielvariable) gleich dem maximalen Wert aller Variablen, die als Liste und zweites Argument angegeben sind[20]. Findet keine der Lessons am jeweiligen Wochentag statt, ist das Maximum der angegebenen Variablen 0.

[19] Da angenommen wird, dass diese Längen pro Kurs alle gleich sind.

[20] Das Constraint selber gibt keine Richtung oder Zuweisung an. Nur die Gleichheit. Da jedoch die neu erstellte (Ziel)-Variable an keine anderen Bedingungen geknüpft ist, alle in der Liste enthaltenen Variablen jedoch schon, kommt das Constraint in diesem Fall einer Zuweisung gleich.

```
for weekdayIndex in range(orm.WEEKDAYS):
    partSGCourses = []
    for course in semesterGroup.courses:
        partSemesterGroupLessons = [x for x in course.lessons if not x.whole_semester_group]
        if partSemesterGroupLessons:
            courseTakePlaceVar = model.NewBoolVar("")
            model.AddMaxEquality(courseTakePlaceVar,
                        [l.weekdayBoolVars[weekdayIndex] for l in partSemesterGroupLessons])

            lessonSize = partSemesterGroupLessons[0].timeslot_size
            partSGCourses.append((courseTakePlaceVar, lessonSize))
```

Codeausschnitt 33: HardConstraints.py - MaxLessonsPerDaySemesterGroupConstraints 3

Hinzufügen der linearen Constraints

Abschließend müssen für jeden Wochentag noch die eigentlichen Constraints hinzugefügt werden. Dadurch wird dass MaxLessonsPerDaySemesterGroupConstraint erst garantiert. Um die in Schritt 2 erstellten Listen, die es ja für jeden Wochentag gibt, nicht noch einmal alle zusammen speichern zu müssen findet dies in der gleichen Schleife statt, die über alle Wochentage iteriert. Wie das Constraint hinzugefügt wird, ist in Listing 34 gezeigt. Es werden die in Schritt 1 und 2 erstellten Listen verwendet. Die Liste aus Schritt 1 ist unabhängig vom Wochentag. In Zeile 2 ist gezeigt, wie aus dieser Liste von Lessons die `BoolVar` Variablen, die angeben, ob die Lesson am aktuellen Wochentag stattfinden und die Lessonlängen (*timeslot_size*) extrahiert werden. Genau wie bei der Implementierung des 3.4.10, wird jeweils das Produkt gebildet. Die Summe ergibt die Anzahl belegter Timeslots an dem Wochentag. In Zeile 3 geschieht dasselbe mit den in Schritt 2 erstellten 2-Tupeln.

```
model.Add(
    sum(map(lambda l: l.timeslot_size * l.weekdayBoolVars[weekdayIndex], lessonsWithWholeSG)) +
    sum(map(lambda c: c[0] * c[1], partSGCourses))
    <= semesterGroup.max_lessons_per_day)
```

Codeausschnitt 34: HardConstraints.py - MaxLessonsPerDaySemesterGroupConstraints 4

3.4.12 MaxLessonsPerDayCourseConstraints

Es soll verhindert werden, dass an einem Tag mehrere Lessons eines Kurses stattfinden. Davon ausgenommen sind Kurse mit dem CourseAllInOneBlockConstraint, Lessons an denen nur Teilgruppen teilnehmen und Lessons die Teil eines LessonsAtSameTimeConstraints sind.

Die Implementierung gestaltet sich einfach. Der gesamte Code ist in Listing 35 enthalten. Es wird über alle Nicht-AllInOneBlock-Kurse und alle Wochentage iteriert. (Zeile 1-2) Dann werden die Lessons des Kurses ermittelt, an denen ganze Semestergruppen teilnehmen und die nicht Teil eines LessonsAtSameTimeConstraints sind (*relevantLessons*).

Anschließend werden die Hilfsvariablen *weekdayBoolVars* jeder Lesson verwendet. Diese geben jeweils für einen Wochentag an, ob die Lesson an diesem stattfindet. Als lineares Cosntraint wird so erreicht, dass die Summe der selektierten Lessons des Kurses an jedem Wochentag kleiner-gleich 1 ist. (Zeile 6)

```
1 for course in filter(lambda c: not c.all_in_one_block, orm.getCourses()):
2     for weekdayIndex in range(orm.WEEKDAYS):
3         relevantLessons = list(filter(lambda l: l.whole_semester_group and not
          ↪  l.lessons_at_same_time, course.lessons))
4         if relevantLessons:
5             model.Add(sum([l.weekdayBoolVars[weekdayIndex] for l in relevantLessons]) <=
              ↪  1)
```

Codeausschnitt 35: HardConstraints.py - MaxLessonsPerDayCourseConstraints

3.4.13 MaxLecturesAsBlockTeacherConstraints

Die Implementierung dieses Constraints ist in der Funktion `addMaxLecturesAsBlockTeacherConstraints` der Datei ***HardConstraints.py*** zu finden. Es wird sichergestellt, dass jeder Lehrende niemals mehr Timeslots mit Vorlesungen hintereinander hat, als für ihn als Maximum angegeben.

Zunächst wurde nach einer Möglichkeit gesucht, mit den Möglichkeiten, die durch die OR-Tools Bibliothek gegeben sind, die Blockgrößen bestimmter Lessons innerhalb eines Tages zu zählen. Mit Blockgröße ist hier die Anzahl direkt aufeinander folgender Timeslots gemeint, die mit bestimmten Lessons belegt sind. Diese Suche hat sich jedoch als erfolglos erwiesen. Als Alternative bietet sich die Möglichkeit, alle Blöcke, das heißt Folgen von belegten oder nicht belegter Timeslots, aufzuzählen und je nach gewünschter maximaler Blockgröße zu verbieten. Dazu wurden diese Blöcke manuell aufgezählt. Die gefundene Lösung funktioniert daher nur mit Stundenplänen, die sechs Timeslots je Tag enthalten. Aus Zeitgründen, wurde auf die Implementierung einer Lösung, die dynamisch die möglichen Blöcke mit variabler Anzahl Timeslots pro Tag aufzählt, verzichtet.

Aufzählung der Blöcke

Für dieses Constraint sind nur die Lessons relevant, die mit dem Flag *is_lecture* versehen sind. Die Anzahl an Timeslots mit diesen Vorlesungen pro Tag und Lehrenden sind bereits durch das MaxLecturesPerDayTeacherConstraint (2.3.15) beschränkt. Dies bedeutet, dass die maximale Blockgröße der Vorlesungen ohnehin nicht größer sein kann, als die angegebene maximale Anzahl belegter Timeslots pro Tag[21]. Daher sind nur gewisse sinnvolle Kombinationen aus maximaler Vorlesungs-Timeslots und maximal großen Vorlesungs-Blöcken denkbar. Diese sind in Tabelle 10 aufgeführt. Warum sind andere Kombinationen nicht sinnvoll? Eine der beiden Größen lässt sich immer reduzieren, ohne den Inhalt der Vorgabe zu verändern. Größere Blöcke als die Anzahl erlaubter Timeslots pro Tag sind sowieso nicht möglich. Eine sinnvolle maximale Blockgröße ist daher immer kleiner gleich der maximalen Anzahl Timeslots am Tag. Auch die maximale Anzahl an

[21] Wie die beide Größen im Stundenplanmodell angegeben sind, soll jedoch nicht eingeschränkt werden. Solange sie zu der Anzahl Timeslots pro Tag passen.

Timeslots lässt sich bei bestimmten maximalen Blockgrößen reduzieren. Nehmen wir als Beispiel die Angabe maximal 5 belegte Timeslots pro Tag und eine maximale Blockgröße von 2. Wenn maximal 2er Blöcke erlaubt sind, können an einem sechs Timeslots langem Tag niemals 5 Timeslots belegt sein. Hier schränkt die erlaubte Blockgröße die Anzahl an belegten Timeslots ein. Daher kann letztere auf 4 reduziert werden.

Max Vorlesungs Timeslots	**Max Blockgröße**
6	6
5	5
5	4
5	3
4	4
4	3
4	2
3	3
3	2
3	1
2	2
2	1
1	1

Tabelle 10: Sinnvolle maximale Timeslots und Blockgrößen Kombinationen

Von den aufgezählten Kombinationen sind jene für dieses Constraint irrelevant, bei denen beide Größen gleich sind. In diesen Fällen wird die Blockgröße bereits durch das MaxLecturesPerDayTeacherConstraint beschränkt und für das MaxLecturesAsBlockTeacherConstraint muss dem `CpModel` nichts hinzugefügt werden um es zu erfüllen. Bleiben 7, für dieses Constraint relevante Kombinationen der beiden Größen. Diese sind in Tabelle 10 grau markiert.

Es ist bei der gefundenen Implementierung nötig, für jede der markierten Kombinationen die nicht erlaubten Belegungen eines Tages mit Timeslots, aufzuzählen. Dies bedeutet, dass jeweils Angegeben wird, in welchen Timeslots des Tages eine Vorlesung stattfindet und in welchen nicht. Dies geschieht durch Angabe einer 1 für einen belegten Timeslot und einer 0 für einen freien Timeslots. Listing 36 zeigt dies beispielhaft für die ersten drei markierten Kombinationen. In der ersten Zeile geschieht dies für die Kombination maximale Anzahl mit Vorlesungen belegter Timeslots gleich 5 und maximale Blockgröße gleich 4. Verboten sind dann die beiden möglichen 5er Blöcke. Unterscheiden sich die beiden Größen um mehr als 1, können die zuvor erstellten Listen verwendet werden, um die jeweils größeren, verbotenen Blöcke hinzufügen.

```
max5_maxblock_4 = [[1, 1, 1, 1, 1, 0], [0, 1, 1, 1, 1, 1]]

max4_maxblock_3 = [[1, 1, 1, 1, 0, 0], [0, 1, 1, 1, 1, 0], [0, 0, 1, 1, 1, 1]]
max5_maxblock_3 = [[1, 1, 1, 1, 0, 1], [1, 0, 1, 1, 1, 1]] + max4_maxblock_3 +
 ↪  max5_maxblock_4
```

Codeausschnitt 36: HardConstraints.py - MaxLecturesAsBlockTeacherConstraints, 1

Alles folgende, wird jeweils für jedes `Teacher` Objekt ausgeführt. Die beiden oben genannten Größen beziehungsweise deren Angabe als Variablen *max_lectures_per_day* und *max_lectures_as_block* der `Teacher` Objekte, müssen noch auf die zuvor genannten Kombinationen reduziert werden. Dies ist nötig, um jeweils die richtigen Listen mit verbotenen Timeslot Konstellationen auswählen zu können. Außerdem ist wichtig, dass das MaxLecturesPerDayTeacherConstraint die reduzierte Anzahl Timeslots pro Tag umsetzt. Nur dann werden wie beschrieben die Blockgrößen auch durch das MaxLecturesPerDayTeacherConstraint beschränkt wie oben angegeben.

Letztlich ist die Reduzierung eine Abbildung

$$\{1,2,3,4,5,6\}^2 \Rightarrow \{1,2,3,4,5,6\}^2, \tag{19}$$

implementierbar, zum Beispiel durch ein Dictionary, dass die Abbildungsvorschrift aller 36 Möglichkeiten enthält. Um gleichzeitig die passende Liste zu verhindernder belegter Timeslot Konstellationen auszuwählen, die zuvor aufgezählt wurden, geschieht die Reduzierung durch klassische if-else Anweisungen.

Anschließend wird die OR-Tools Funktion `AddForbiddenAssignments` und die ausgewählte Liste mit Timeslot Konstellationen die die zu großen Blöcke enthalten genutzt um das Constraint umzusetzen.

Das passende Gegenstück zu den aufgezählten Konstellationen ist eine Liste mit 6 `BoolVar` Variablen, die angeben, ob zum 1., 2., 3., ... Timeslot eines Tages für den jeweiligen Lehrenden eine Vorlesung stattfindet. Die im Abschnitt zu den Hilfsvariablen vorgestellte Variable (vom Typ Dictionary) *timeslotLectureBoolMap* enthält für jeden Lehrenden und jeden Timeslot des Stundenplans diese `BoolVar` Variablen, die angeben ob zu dem Timeslot eine Vorlesung[22] stattfindet.

```
for day in orm.getTimeslotsPerDay():
    boolVarList = list(map(lambda t: teacher.timeslotLectureBoolMap[t], day))
    model.AddForbiddenAssignments(boolVarList, tuple(forbidden_assignments))
```

Codeausschnitt 37: HardConstraints.py - MaxLecturesAsBlockTeacherConstraints, 2

Wie in Listing 37 gezeigt, wird für jeden Wochentag eine Liste der `BoolVar` Variablen erstellt. Die verwendete Iterationsvariable *day* enthält jeweils die Timeslots des Wochentages. Angenommen ein Lehrender hätte zu Beginn eines Tages 4 Vorlesungs-Timeslots.

[22] Das heißt eine Lesson, bei der das *is_lecture* Flag gesetzt ist.

Dann wäre der ausgewertete Inhalt der für den Tag erstellen Liste *boolVarList*:

$$[1, 1, 1, 1, 0, 0] \tag{20}$$

Ist die maximale Blockgröße für Vorlesungen des Lehrenden aber zum Beispiel 3, ist unter anderem genau diese Kombination in der Liste der nicht erlaubten Konstellationen enthalten. (Für *max_lectures_per_day* = 4, ist dies das erste Element, was in Listing 36 in Zeile 3 aufgezählt wird.) Das OR-Tools Constraint `AddForbiddenAssignments` verbietet dann genau diese Kombination, wodurch erreicht wird, dass der Stundenplan des Lehrenden keine solchen Blöcke enthält.

3.4.14 MaxLecturesPerDayTeacherConstraints

Es soll für jeden Lehrenden die Anzahl an mit Vorlesungen belegter Timeslots an jedem Wochentag beschränkt werden. Umsetzung des Constraints MaxLecturesPerDayTeacherConstraint (2.3.15). Die Implementierung befindet sich in der Funktion `addMaxLecturesPerDayTeacherConstraints`. Die Implementierung entspricht im Wesentlichen der des MaxLessonsPerDayTeacherConstraints, in Abschnitt (3.4.10 MaxLessonsPerDayTeacherConstraints. Die folgenden Schritte, werden jeweils für jeden Lehrenden durchgeführt. Die Variable *teacher* dient als Iterationsvariable über alle `Teacher` Objekte.

Zunächst werden alle Lessons des Lehrenden gefunden, bei denen das *is_lecture* Flag gesetzt ist und die nicht Teil eines LessonsAtSameTimeConstraints sind. Diese werden in der Menge *lecturesForTeacher* gespeichert, Listing 38, Zeile 1-2.

Anschließend wird aus den Mengen der gleichzeitig stattfindenden Lessons der LessonsAtSameTimeConstraints, jeweils die längste Vorlesung herausgesucht, an der der Lehrende teilnimmt. Sollte es für ein LessonsAtSameTimeConstraint so ein `Lesson` Objekt geben, wird es ebenfalls der Menge *lecturesForTeacher* hinzugefügt. Listing 38, Zeile 4-7. Die Funktion `getLessonsAtSameTimeSets` der Skriptdatei ***ORM.py*** liefert genau diese Mengen.

```
1 nonSameTimeLectures = [l for l in teacher.lessons if l.course.is_lecture and not
   ↪  l.lessons_at_same_time]
2 lecturesForTeacher.update(nonSameTimeLectures)
3
4 for sameTimeSet in orm.getLessonsAtSameTimeSets():
5     lectures = [l for l in sameTimeSet if l in teacher.lessons and l.course.is_lecture]
6     if lectures:
7         lecturesForTeacher.add(max(lectures, key=lambda l: l.timeslot_size))
```

Codeausschnitt 38: HardConstraints.py - MaxLecturesAsBlockTeacherConstraints

3.4.15 OneCoursePerDayTeacherConstraints

Die Besonderheit der Kurse für die das OneCoursePerDayTeacherConstraint (2.3.13) gilt, ist, dass ein Lehrender nie Lessons von verschiedenen dieser Kurse an einem Tag unterrichten muss. Anders gesagt, ist derselbe Lehrende an mehreren dieser Kurse beteiligt, dürfen keine Lessons dieser Kurse ab selben Tag stattfinden. Die Implementierung dieses

Constraints befindet sich in der Funktion `addOneCoursePerDayPerTeacherConstraints` in der Datei ***HardConstraints.py***. Der gesamte relevante Inhalt ist in Listing 39 zu sehen.

Alles folgende wird jeweils für jeden Lehrenden ausgeführt. Zunächst wird eine Liste mit allen Kursen mit dem OneCoursePerDayTeacherConstraint und an denen der Lehrende teilnimmt, erstellt. (Listing 39, Zeile 1)

Der folgende Teil entspricht genau der Anschauung die bei der Erklärung des Constraints in Abschnitt 2.3.13 OneCoursePerDayTeacherConstraint und in der dort aufgeführten Formel 15 beschrieben wurde.

Es wird zunächst über alle 2-Kombinationen[23] der ermittelten Kurse iteriert. Dies geschieht durch die beiden Zählvariablen *i, j*, in Zeile 2 und 3 des Codeausschnitts 39.

Nun wird für jedes dieser Kurs-Paare, jeweils die Liste mit `Lesson` Objekten ausgewählt und auf die Lessons beschränkt, an denen der Lehrende auch tatsächlich teilnimmt. Aus diesen beiden Listen der Kurs-Paare wird nun das Kartesische Produkt gebildet. Ein einzelnes Element des Kartesischen Produktes wird jeweils durch die Variablen *i_lesson* und *j_lesson* gebildet. (Zeile 4-5)

Diese Lessons dürfen jeweils nicht am gleichem Tag stattfinden. Dies wird sichergestellt, indem dem `CpModel` ein Constraint hinzugefügt wird, dass die beiden Hilfsvariablen der Lessons, die angeben an welchem Wochentag diese stattfinden (*weekdayVar*), unterschiedlich sein müssen. (Zeile 6)

```
1 courses = list(filter(lambda c: c.one_per_day_per_teacher, teacher.getCourses()))
2 for i in range(len(courses)):
3     for j in range(i + 1, len(courses)):
4         for i_lesson in [l for l in courses[i].lessons if teacher in l.teachers]:
5             for j_lesson in [l for l in courses[j].lessons if teacher in l.teachers]:
6                 model.AddAllDifferent([i_lesson.weekdayVar, j_lesson.weekdayVar])
```

Codeausschnitt 39: HardConstraints.py - MaxLecturesAsBlockTeacherConstraints

3.5 SoftConstraints, Implementierung

Bei den optionalen oder weichen Anforderungen, geht es darum, diese so oft wie möglich zu erfüllen. Durch die verschiedenen Gewichtungen der einzelnen Constraints gibt es eine Vorgabe wann die Erfüllung eines der Constraints der Erfüllung eines anderen vorzuziehen ist. Dass die optionalen Constraints überhaupt möglichst eingehalten werden und die Präferenzen der Constraints untereinander, wird durch die Optimierungsfunktion erreicht. Ihr Wert ist die Anzahl aller nicht erfüllten weichen Anforderungen, jeweils multipliziert mit dem Gewicht der Anforderung. Bei der eigentlichen Lösungssuche wird dann nach Lösungen gesucht, bei denen der Wert der Optimierungsfunktion möglichst niedrig ist. Der Wert der Optimierungsfunktion wird im Folgenden als Objective bezeichnet.

[23] Kombination mit 2 Elementen, ohne Wiederholung, ohne Berücksichtigung der Reihenfolge. Entspricht allen 2-elementigen Teilmengen.

Um das Objective berechnen zu können, ist es notwendig für die einzelnen Constraints zu zählen, wie oft sie verletzt sind. Der Test auf Erfüllung eines Constraints und der Aufbau der Optimierungsfunktion sollen möglichst stark voneinander getrennt stattfinden. Daher gibt es für jede (mit einer Ausnahme) optionale Anforderung eine Funktion, die grundlegende Variablen erstellt mit denen die nicht-Erfüllung der Constraints festgestellt werden kann und Variablen um dies zu zählen. Mit Ausnahme der Constraints AvoidLateTimeslotsConstraint und 2.4.3, sind diese erstellen Variablen immer an die Objekte geknüpft, für die sie gelten. Zum Beispiel wird eine Variable, die für eine Semestergruppe zählt, wie viele Lücken der Größe 1 ihr Stundenplan enthält, direkt an das entsprechende `SemesterGroup` Objekt angehängt. Bei den genannten Ausnahmen gibt es keine Objekte, an die das Constraint gebunden ist. Daher liefern die Funktionen die diese Constraints überprüfen, direkt Variablen zurück, die die Anzahl an Verletzungen enthalten. Alle Funktionen die ein optionales Constraint überprüfen, befinden sich in der Datei ***SoftConstraints.py***.

Des weiteren enthält diese Datei eine Funktion, welche nach Erstellung aller nötigen Variablen für die einzelnen Constraints, die Elemente der Optimierungsfunktion erstellt und zusammenfasst. Der Name der Funktion lautet `createObjectiveFunctionSummands`.

Die Funktion `createObjectiveFunctionSummands`

An dieser Stelle kommen auch die Gewichtungen ins Spiel. Die Funktion liefert eine Liste mit Objekten, aus denen sich der Wert der Optimierungsfunktion berechnen lässt. Diese Objekte haben den Typ `LinearExpr`, unter anderem die Basisklasse von `IntVar`. Die Liste enthält Summanden, deren Summe für das Objective steht.

Während des Erstellens dieser Liste, ruft diese Funktion auch alle Funktionen der einzelnen weichen Anforderungen in der Datei ***SoftConstraints.py*** auf. Damit genügt es diese Funktion aufzurufen und die zurückgegebene Liste der Summanden dem `CpModel` als Objective hinzuzufügen.

Inhalt der Funktion ist es, die Liste der Summanden zu initialisieren und einen Codeteil für jede einzelne optionale Anforderung auszuführen. Diese Codeteile fügen der Summanden-Liste jeweils die Summanden der jeweiligen Anforderung hinzu. Da sie Constraint-spezifisch sind, werden sie jeweils im Abschnitt des zugehörigen optionalen Constraints erläutert.

Speicherung der Gewichte

Durch manuelle Angabe der Gewichte der einzelnen Constraints, kann angepasst werden, welche Anforderungen anderen zu bevorzugen ist. Der Einfachheit halber, wird eine Python Datei mit Variablen für jedes Gewicht verwendet. Der Name der Datei ist ***SoftConstraintWeights.py***. Da außer den Gewichten und Kommentaren nichts in der Datei enthalten ist, gleicht sie einer Konfigurationsdatei mit Key-Value Paaren. Die dort angegebenen Gewichte müssen Integer, größer-gleich 0 sein[24]. Als obere Grenze für die

[24]Theoretisch wäre es auch möglich negative Werte anzugeben. In diesen Fällen würde die Erfüllung des jeweiligen Constraints möglichst vermieden werden. Da dies aber nicht im Sinne der implementierten Stundenplansuche ist und nicht ausführlich getestet wurde, sollte davon Abstand genommen werden.

Gewichte muss lediglich bedacht werden, dass der Wert des Objectives den Wertebereich einer 64-bit signed Integer Variable nicht übersteigen darf.

Im Folgendem wird für jedes optionale Constraint gezeigt, wie es überprüft wird und wie die Liste der Summanden für die Optimierungsfunktion in der Funktion `createObjectiveFunctionSummands` gebildet wird.

3.5.1 PreferFirstStudyDayChoiceConstraint

Dieses Constraint soll sicherstellen, dass von den beiden Wahlmöglichkeiten welche die Lehrenden für ihren Studientag haben, möglichst die Erstwahl umgesetzt wird. Dazu soll für jeden Lehrenden dessen Erstwahl nicht umgesetzt wird, der Wert des Gewichts dieses Constraints, dem Objective hinzugefügt werden.

Wenn das StudyDayConstraint (2.3.5) umgesetzt wird, geschieht dies über zwei Variablen, die jeweils angeben, ob der Erstwahl-Tag beziehungsweise der Zweitwahl-Tag erfüllt ist. Da die erstgenannte Variable an dieser Stelle wiederverwendet werden kann, muss nichts weiter implementiert werden um den Umgesetzten Studientag eines Lehrenden festzustellen. Daher ist dies das einzige optionale Constraint, für das es keine gesonderte Funktion in der Datei ***SoftConstraints.py*** gibt. Der Name der verwendeten Variable ist *`studyDay1BoolVar`* und ist nach dem Hinzufügen der harten Anforderungen jedem `Teacher` Objekt eines Lehrenden dem ein Studientag zusteht, angehängt.

Im folgendem wird gezeigt, wie in der Funktion `createObjectiveFunctionSummands` aus den Variablen der Lehrenden, welche die Erfüllung des Erstwahl-Studientages angeben, die passenden Summanden der Optimierungsfunktion erstellt werden. Diese Summanden sind letztlich der Anteil des PreferFirstStudyDayChoiceConstraints an der Berechnung des Objectives.

Bildung der Optimierungsfunktion

Für jeden Lehrenden mit einem Studientag, soll für die Optimierungsfunktion ein Ausdruck gebildet werden. Dieser Ausdruck soll zu dem Wert der Gewichtung des Constraints oder zu 0 ausgewertet werden, je nach dem ob das PreferFirstStudyDayChoiceConstraint erfüllt ist oder nicht.

Dazu soll die Variable *`studyDay1BoolVar`*, in negierter Form, mit dem Wert des Gewichtes multipliziert werden. Die Negierung ist notwendig, da die Variable angibt, ob das Constraint erfüllt ist. Benötigt wird jedoch das Gegenteil. Der Wert dieser `BoolVar` Variable soll 1 sein, wenn es nicht erfüllt ist. Listing 40 zeigt die Implementierung.

```
1 for teacher in filter(lambda t: t.hasStudyday() and t.lessons, orm.getTeachers()):
2     summands.append(teacher.studyDay1BoolVar.Not() * PREFER_FIRST_STUDYDAY_PENALTY)
```

Codeausschnitt 40: SoftConstraints.py - createObjectiveFunctionSummands - PreferFirstStudyDay

Es soll nur für Lehrende die eine Studientag haben und denen auch tatsächlich Lessons zugeordnet sind, Summanden erstellt werden. Daher werden die Lehrenden in Zeile 1 auf

diese beiden Attribute hin gefiltert. Dann wird die Negierung der *studyDay1BoolVar* mit dem Gewicht des Constraints multipliziert. Das Gewicht ist als Konstante *PREFER_FIRST_STUDYDAY_PENALTY* angegeben.

Das Ergebnis dieser Multiplikation ist ein Objekt des Typs `LinearExpr` beziehungsweise spezifischer der Subklasse `_ProductCst`. `LinearExpr` ist die Basisklasse der OR-Tools Bibliothek, die verwendet wird um lineare Integer Ausdrücke abzubilden. [Goog]

3.5.2 CountLessonsAtHour

Um das Stattfinden von Lessons zu einem bestimmten Timeslot eines Tages zu bestrafen, muss ermittelt werden wie viele Lessons dies sind. Die Implementierung des Zählens dieser Lessons befindet sich in der Funktion `addCountLessonsAtNthHour`. Sie bekommt als Parameter unter anderem die Nummer (= SStunde") übergeben zu deren Stattfinden die Lessons gezählt werden sollen. Gezeigt wird hier eine allgemeine Implementierung, die mit jeder Timeslot Nummer funktioniert. Mit Nummer eines Timeslots ist hier die Nummer an einem Wochentag gemeint. Mögliche Nummern sind daher 1 bis zur Anzahl der Timeslots an einem Tag.

Es wurde zunächst auch spezielle Implementierungen für bestimmte Timeslot Nummern gefunden. Diese können dann spezielle Umstände ausnutzen, zum Beispiel dass es sich um den letzten oder ersten Timeslot eines Tages handelt. Auch würden bei diesen, bestimmte Hilfsvariablen für die Implementierung nicht benötigt. Da sich jedoch kein signifikanter Zeitvorteil bei der Lösungssuche feststellen ließ und die Hilfsvariablen sowieso auch für andere Constraints benötigt werden, wurde von diesen speziellen Implementierungen wieder Abstand genommen.

Die Liste mit Hilfsvariablen *timeslotBoolVars*, die für jedes `Lesson` Objekt existiert, enthält `BoolVar` Variablen für jeden Timeslot des Stundenplans. Sie geben jeweils an, ob die Lesson zu dem jeweiligen Timeslot stattfindet. Diese Variablen können verwendet werden, um die Lessons zu zählen, die zu bestimmten Timeslots eines Tages stattfinden. Dazu müssen aus der genannten Liste jedes `Lesson` Objektes, die Variablen herausgesucht werden, die zu Timeslots gehören, zu deren Stattfinden die Lessons gezählt werden sollen. Angenommen es sollen die Lessons gezählt werden, die in der ersten Stunde, das heißt als erstes an einem Tag stattfinden. Wenn der Stundenplan 5 Wochentage und 6 Timeslots pro Tag enthält, sind dies die Timeslots mit den IDs 1, 7, 13, 19, 25. Da die Variablen in der Liste vom Typ `BoolVar` sind, die nur den Wert 1 enthalten, wenn die Lesson zum zugehörigen Timeslot stattfindet, genügt es die Summe aus allen dieser herausgesuchten Variablen, von allen Lessons zu bilden um die Anzahl von Lesson zu erhalten die zur gegebenen Stunde stattfinden.

Die gesamte Implementierung ist in Listing 41 zu sehen. Die Nummer der entsprechenden Timeslots wird der Funktion durch den Paramater *timeslotNumber* übergeben. Zuerst wird die `IntVar` Variable erstellt, die am Ende die Anzahl der Lessons enthalten soll, die zur gegebenen Stunde stattfinden. (Zeile 1)

Dann wird die Liste der `BoolVar` Hilfsvariablen erstellt. Dazu werden die Timeslots ermittelt, deren Nummer mit der gegebenen Nummer *timeslotNumber* überein-

stimmt. (Zeile 6) Mithilfe der ID des jeweiligen Timeslots wird die zum Timeslot passende *timeslotBoolVar* gefunden.

Anschließend wird aus allen gesammelten *timeslotBoolVars* die Summe gebildet und der Variablen *nthHourCount* zugewiesen.

```
1 nthHourCount = model.NewIntVar(0, len(orm.getLessons()), "nthHourCount")
2
3 boolVars = []
4
5 for lesson in orm.getLessons():
6     for timeslot in [t for t in orm.getTimeslots() if t.number == timeslotNumber]:
7         boolVars.append(lesson.timeslotBoolVars[timeslot.id-1])
8
9 model.Add(nthHourCount == LinearExpr.Sum(boolVars))
```

Codeausschnitt 41: SoftConstraints.py - createObjectiveFunctionSummands - PreferFirstStudyDay

Bildung der Optimierungsfunktion

Um in der Funktion `createObjectiveFunctionSummands` aus den erstellen Variablen für die Anzahlen der Lessons im ersten, fünften und sechsten Timeslot eines Wochentages, müssen diese lediglich mit dem jeweiligen Gewicht multipliziert werden. Die Gewicht für die einzelnen Lückengrößen sind in der Datei ***SoftConstraintsWeights.py*** enthalten. Listing 42 zeigt wie die Funktionen aufgerufen werden um jeweils die Variable mit der Anzahl Lessons zu der jeweiligen Timeslot Nummer zu erhalten. Diese Variable wird dann multipliziert mit dem Gewicht des jeweiligen Constraints zur Liste der Summanden hinzugefügt.

```
1 sixthHourCount = addCountLessonsAtNthHour(model, orm, 6)
2 summands.append(sixthHourCount * SIXTH_HOUR_PENALTY)
3
4 fifthHourCount = addCountLessonsAtNthHour(model, orm, 5)
5 summands.append(fifthHourCount * FIFTH_HOUR_PENALTY)
6
7 firstHourCount = addCountLessonsAtNthHour(model, orm, 1)
8 summands.append(firstHourCount * FIRST_HOUR_PENALTY)
```

Codeausschnitt 42: SoftConstraints.py - createObjectiveFunctionSummands - AvoidLateAndEarlyTimeslots

Anmerkung: Zwar besteht bei dieser Implementierung wie auch schon bei der Beschreibung der Constraints AvoidLateTimeslotsConstraint und AvoidEarlyTimeslotsConstraint eine Abhängigkeit zu einem Stundenplan mit 6 Timeslots pro Tag, dies ließe sich jedoch leicht Aufheben. Es würde genügen die Gewichte zum Beispiel in *LAST_HOUR_PENALTY* und *SECOND_LAST_HOUR_PENALTY* umzubenennen und beim Aufruf der Funktion `addCountLessonsAtNthHour` (Listing 42 Zeile 1 und 4) jeweils die Konstante *TIMESLOTS_PER_DAY* der ***ORM.py*** Datei zu verwenden.

3.5.3 CountGapsBetweenLessonsSemesterGroup

Um Lücken im Stundenplan für die Semestergruppen zu vermeiden, müssen diese gezählt werden. Dann kann die Anzahl der Lücken in der Optimierungsfunktion bestraft werden. Als Lücke wird eine Konstellation im Stundenplan bezeichnet, bei der ein oder mehrere Timeslots frei sind, das heißt keine Lesson mit der jeweiligen Semestergruppe stattfindet, vor und nach diesen Timeslots jedoch schon. Bei einem Stundenplan mit 6 Timeslots pro Tag sind Lücken bis zur Größe 4 denkbar.

Die Implementierung befindet sich in der Funktion `addCountGapsVariables` in der Datei ***SoftConstraints.py***

Alles folgende wird jeweils für jede Semestergruppe ausgeführt.

Schritt 1: Um die Lücken feststellen zu können, müssen zunächst (`BoolVar`) Hilfsvariablen für jeden Timeslot erstellt werden, die angeben, ob zu dem jeweiligen Timeslot eine Lesson für die Semestergruppe stattfindet.

Schritt 2: Dann werden `BoolVar` Variablen für jede Lücke erstellt, die theoretisch auftreten kann. Das heißt für jeden Timeslots und jede Gruppe von Timeslots an denen eine Lücke sein kann. Damit diese Variablen die richtigen Werten annehmen, werden jeweils die Hilfsvariablen für die Timeslots der Lücke, den Timeslot davor und danach verwendet.

Schritt 3: Abschließend wird für jede Lückengröße eine `IntVar` Variable erstellt, die die Anzahl an Lücken für die jeweilige Semestergruppe und der jeweiligen Größe enthalten sollen. Die Anzahl wird über die Summe (der Werte) aller erstellten `BoolVar` Variablen der möglichen Lücken einer Größe ermittelt.

Da der Inhalt der der Funktion umfangreich ist, wird im Folgendem nur ein Ausschnitt gezeigt.

Um die Hilfsvariablen in Schritt 1 mit Werten belegen zu können, werden Hilfsvariablen der Lessons verwendet, die jeweils für einen Timeslot angeben, ob die Lesson zu ihm stattfindet (*`timeslotBoolVars`*). In Listing 43 ist der Code gezeigt, der dazu für jede Semestergruppe (Variable *`sg`*) und jeden Timeslot *`timeslot`* ausgeführt wird. Für jeden Timeslot werden diese von allen Lesson gesammelt, an denen die jeweilige Semestergruppe teilnimmt. (Zeile 1-3)

```
1 lessonTimeslotBoolVars = []
2 for lesson in sg.getLessons():
3     lessonTimeslotBoolVars.append(lesson.timeslotBoolVars[timeslot.id - 1])
4
5 tOccupied = model.NewBoolVar("")
6
7 model.AddBoolOr(lessonTimeslotBoolVars).OnlyEnforceIf(tOccupied)
8 model.AddBoolAnd([b.Not() for b in lessonTimeslotBoolVars]).OnlyEnforceIf(tOccupied.Not())
```

Codeausschnitt 43: SoftConstraints.py - addCountGapsVariables - Teil 1

Dann wird die Variable *`tOccupied`* erstellt, die angibt ob zum aktuellen Timeslot

der Iteration eine Lesson stattfindet, an der die Semestergruppe teilnimmt. Dies ist der Fall, wenn eine der gesammelten *timeslotBoolVars* wahr ist. (Or-Constraint in Zeile 7) Beziehungsweise nicht der Fall, wenn alle von ihnen den Wert falsch haben. (Zeile 8) Alle *tOccupied* Variablen werden anschließend in einem Dictionary, mit dem zugehörigen `Timeslot` Objekt als Key, gespeichert.

Die Umsetzung des zweiten Schritts wird nun für Lücken der Größe 1 gezeigt. Der Code ist in Listing 44 zu sehen. `BoolVar` Variablen für jede mögliche Lücke werden in einer Liste *oneGaps* gesammelt. Nun wird über jeden Wochentag und jede mögliche Lücke iteriert. Eine Lücke kann nur im zweiten (Index = 1) bis zum vorletzten Timeslot eines Tages stattfinden. (Zeile 3) Für eine Lücke die durch die Variable *gap* representiert wird, werden die Timeslots davor, der Lücke selber und danach benötigt. (Zeile 5-7) Die `BoolVar` der Lücke soll dann den Wert True annehmen, wenn zum Timeslot vor und nach der Lücke eine Lesson für die Semestergruppe stattfindet und zum Timeslot der Lücke nicht. (Zeile 8-10) Anschließend wird noch angegeben wann keine Lücke vorliegt, der Wert der Variablen also False sein soll.

```
1  oneGaps = []
2  for day in range(orm.WEEKDAYS):
3      for i in range(1, orm.TIMESLOTS_PER_DAY-1):
4          gap = model.NewBoolVar("")
5          timeslotBefore = orm.getTimeslots()[day * orm.TIMESLOTS_PER_DAY + i - 1]
6          timeslot = orm.getTimeslots()[day * orm.TIMESLOTS_PER_DAY + i]
7          timeslotAfter = orm.getTimeslots()[day * orm.TIMESLOTS_PER_DAY + i + 1]
8          model.AddBoolAnd([sg.timeslotOccupiedMap[timeslotBefore],
9                            sg.timeslotOccupiedMap[timeslot].Not(),
10                           sg.timeslotOccupiedMap[timeslotAfter]]).OnlyEnforceIf(gap)
11         model.AddBoolOr([sg.timeslotOccupiedMap[timeslotBefore].Not(),
12                          sg.timeslotOccupiedMap[timeslot],
13                     sg.timeslotOccupiedMap[timeslotAfter].Not()]).OnlyEnforceIf(gap.Not())
14         oneGaps.append(gap)
```

Codeausschnitt 44: SoftConstraints.py - addCountGapsVariables - Teil 2

Als dritter Schritt muss nun nur noch gezählt werden, wie oft eine Lücke vorliegt. Dies ist die Anzahl der zuvor erstelleten Variablen, die den Wert True haben. Dies wird über ein einfaches lineares Constraint mit Bildung der Summe aller `BoolVar` Variablen der einzelnen möglichen Lücken erreicht. (Listing 45, Zeile 2)

```
1  sg.oneGapCount = model.NewIntVar(0, len(oneGaps), "")
2  model.Add(sg.oneGapCount == sum(oneGaps))
```

Codeausschnitt 45: SoftConstraints.py - addCountGapsVariables - Teil 3

Bildung der Optimierungsfunktion

Um in der Funktion `createObjectiveFunctionSummands` aus den erstellen Variablen für die Anzahlen der Lücken die Summanden für die Optimierungsfunktion zu erstellen, müssen diese lediglich mit dem jeweiligen Gewicht multipliziert werden. Die Gewicht für

die einzelnen Lückengrößen sind in der Datei ***SoftConstraintsWeights.py*** enthalten. Listing 46 zeigt wie die Summanden für die Lücken der Liste aller Summanden der Optimierungsfunktion hinzugefügt werden. Die Gewichte für die Lückengrößen sind jeweils als zweiter Operand der Multiplikation angegeben.

```
1 for sg in orm.getSemesterGroups():
2     summands.append(sg.oneGapCount * ONE_TIMESLOT_GAP_PENALTY)
3     summands.append(sg.twoGapCount * TWO_TIMESLOT_GAP_PENALTY)
4     summands.append(sg.threeGapCount * THREE_TIMESLOT_GAP_PENALTY)
5     summands.append(sg.fourGapCount * FOUR_TIMESLOT_GAP_PENALTY)
```

Codeausschnitt 46: SoftConstraints.py - createObjectiveFunctionSummands - Timeslot Lücken

3.5.4 CountGapsBetweenDaysTeacher

Für bestimmte Lehrende, die dies wünschen, sollen Lücken mit freien Tagen, zwischen Wochentagen mit Veranstaltungen für sie, vermieden werden. Dies ist die Umsetzung des Constraints AvoidGapBetweenDaysTeacherConstraint (2.4.5). Die Implementierung befindet sich in der Funktion `addGapBetweenDaysTeacherVariables` in der Datei ***SoftConstraints.py***. Die Angabe, ob solche Lücken für einen Lehrenden vermieden werden sollen, geschieht durch die `bool` Variable *`avoid_free_day_gaps`* in jedem `Teacher` Objekt. Die genannte Funktion fügt jedem `Teacher` Objekt, bei dem diese Variable True ist und dessen *`lessons`* Liste mindestens zwei `Lesson` Objekte enthält, drei Variablen *`oneDayGapCount`*, *`twoDayGapCount`* und *`threeDayGapCount`* des Typs `IntVar` hinzu. Diese Variablen geben jeweils an, wie viele Lücken der jeweiligen Größe es im Stundenplan des Lehrenden gibt.

Das im Folgendem beschriebene Verfahren um diese Variablen zu erstellen, ähnelt sehr der Implementierung des Constraints AvoidGapBetweenLessonsSemesterGroupConstraint (3.5.3 CountGapsBetweenLessonsSemesterGroup).

Um Lücken in Form von Tagen ohne Veranstaltungen eines Lehrenden erkennen zu können, werden zunächst Hilfsvariablen benötigt, die angeben, ob an einem Wochentag mindestens eine Veranstaltung des Lehrenden stattfindet. Diese werden mit den Hilfsvariablen *`weekdayBoolVars`* der `Lesson` Objekte des Lehrenden erstellt. Diese geben für jeden Wochentag an, ob die Lesson an dem Tag stattfindet.

Als zweiter Schritt werden für jede mögliche Lücke der Größen 1, 2 und 3, `BoolVar` Variablen erstellt, die angeben sollen, ob eine Lücke am jeweiligen Tag existiert. Um diesen Variablen die richtigen Werte zuzuweisen, werden die zuvor erstellten Hilfsvariablen verwendet.

Im letzten Schritt werden Variablen für die Anzahlen der Lücken jeweiliger Größe erstellt und den Objekten der Lehrenden hinzugefügt. Ihr Wert ist jeweils die Anzahl der Variablen welche die Existenz spezifischer Lücken anzeigen, die den Wert $True$ haben.

Folgendes wird für jeden Lehrenden ausgeführt, der die Vermeidung dieser freien Tage wünscht und der mindestens zwei Lessons unterrichtet.

Für jeden Wochentag wird eine `BoolVar` Variable erstellt und in der Liste *workingDayBools* gespeichert. Für den zweiten Schritt werden zudem negierte Versionen dieser Variablen benötigt. Diese werden in der Liste *workingDayBoolsNeg* gespeichert. (Listing 47, Zeile 1-2)

```
1 workingDayBools = [model.NewBoolVar("") for x in range(orm.WEEKDAYS)]
2 workingDayBoolsNeg = [model.NewBoolVar("") for x in range(orm.WEEKDAYS)]
3
4 for i in range(orm.WEEKDAYS):
5     model.AddMaxEquality(workingDayBools[i], [l.weekdayBoolVars[i] for l in teacher.lessons])
6     model.Add(workingDayBoolsNeg[i] == workingDayBools[i].Not())
```

Codeausschnitt 47: SoftConstraints.py - addGapBetweenDaysTeacherVariables, Teil 1

Um einer *woorkingDayBools* Variablen für den Wochentag *i* den richtigen Wert zuzuweisen, wird die Methode `AddMaxEquality` der Klasse `CpModel` verwendet. Diese fügt das Constraint hinzu, dass der Wert der als ersten Parameter angegebenen Variable, gleich dem maximalen Wert der als zweiten Parameter übergebenen Liste von Variablen entspricht. So wird erreicht, dass die *woorkingDayBool* Variable den Wert 1 bekommt, sobald mindestens eine der *weekdayBoolVars* für den Wochentag *i* den Wert 1 (also True) hat. (Zeile 5) Es werden die *weekdayBoolVars* aller `Lesson` Objekte des Lehrenden genommen. Diese haben den Wert 1, wenn die Lesson am zugehörigen Wochentag stattfindet.

Anschließend wird noch der negierten Variante der Variablen der negierte Wert zugewiesen. (Zeile 6)

Nun wird der zweite Schritt exemplarisch für Tages-Lücken der Größe 1 gezeigt. Es wir eine Liste mit `BoolVar` Variablen für alle Lücken der Größe 1 erstellt. Diese können ab dem zweiten bis zum vorletzten Wochentag auftreten. Daher wird über Indexe dieser Tage iteriert (Listing 48 Zeile 2)

Anders als bei der Implementierung 3.5.3 CountGapsBetweenLessonsSemesterGroup, wird hier nicht die in 3.4 vorgestellte Methode verwendet, um der *oneDayGap* Variable den richtigen Wert zuzuweisen. Diese Variante kommt mit einem einzigen anstelle von zwei Constraints aus. Jedoch nimmt die verwendetet Methode `AddMinEquality` keine `_NotBooleanVariable` Objekte entgegen, welche beim Aufruf der `Not()` Methode auf `BoolVar` Objekten zurückgegeben werden. Daher wurden im vorigen Abschnitt negierte Varianten der einzelnen *workingDayBools* erstellt.

In Zeile 4-6 ist die Verwendung des `MinEquality` Constraints zu sehen um der *oneDayGap* Variablen den Wert 1 zuzuweisen, wenn am Wochentag mit dem Index *dayIndex* der Schleife, eine Lücke auftritt. Da nur `BoolVar` Variablen verwendet werden, kann das Minimum der als zweiten Parameter übergebenen Liste von Variablen nur 0 oder 1 sein. Es ist nur eins, wenn alle Variablen der Liste den Wert 1 haben. Daher werden die Variablen übergeben, die genau den Umstand der Lücke anzeigen. Am Tag vor der Lücke muss mindestens eine Lesson des Lehrenden stattfinden (Hilfsvariable *workingDayBools[dayIndex -1]*), am Tag der Lücke darf keine Lesson stattfinden (Hilfsvariable

workingDayBoolsNeg[dayIndex]) und am Tag nach der Lücke muss wieder mindestens eine Lesson des Lehrenden stattfinden (Hilfsvariable *workingDayBools[dayIndex + 1]*).

```
oneDayGaps = []
for dayIndex in range(1, orm.WEEKDAYS - 1):
    oneDayGap = model.NewBoolVar("One_Day_Gap")
    model.AddMinEquality(oneDayGap, [workingDayBools[dayIndex - 1],
                                     workingDayBoolsNeg[dayIndex],
                                     workingDayBools[dayIndex + 1]])
    oneDayGaps.append(oneDayGap)
```

Codeausschnitt 48: SoftConstraints.py - addGapBetweenDaysTeacherVariables, Teil 2

Als letzter Schritt muss nur noch die Anzahl der Lücken ermittelt werden. Dies ist die Anzahl der erstellten *oneDayGap* Variablen, die den Wert $True(=1)$ enthalten. Dies entspricht der Summe (der Werte) all dieser Variablen. Für Lücken der Größe 1 ist dies in Listing 49 gezeigt.

```
teacher.oneDayGapCount = model.NewIntVar(0, len(oneDayGaps), "")
model.Add(teacher.oneDayGapCount == sum(oneDayGaps))
```

Codeausschnitt 49: SoftConstraints.py - addGapBetweenDaysTeacherVariables, Teil 3

Bildung der Optimierungsfunktion

Um in der Funktion `createObjectiveFunctionSummands` aus den erstellen Variablen für die Anzahlen der Tages-Lücken die Summanden für die Optimierungsfunktion zu erstellen, müssen diese lediglich mit dem jeweiligen Gewicht multipliziert werden. Dies ist in Listing 50 zu sehen. Um festzustellen für welche `Teacher` Objekte die Variablen erstellt wurden, wird nach dem Attribut *oneDayGapCount* gesucht. Dieser Filter ist äquivalent zur Suche nach Lehrenden mit dem *avoir_free_day_gaps* Flag und mindestens zwei Lessons.

```
addGapBetweenDaysTeacherVariables(model, orm)
for teacher in filter(lambda t: hasattr(t, "oneDayGapCount"), orm.getTeachers()):
    summands.append(teacher.oneDayGapCount * ONE_DAY_GAP_PENALTY)
    summands.append(teacher.twoDayGapCount * TWO_DAY_GAP_PENALTY)
    summands.append(teacher.threeDayGapCount * THREE_DAY_GAP_PENALTY)
```

Codeausschnitt 50: SoftConstraints.py - createObjectiveFunctionSummands, TeacherDayGaps

3.5.5 CountLessonsOnFreeDaySemesterGroup

Wenn Semestergruppen einen freien Tag wünschen (2.4.6 FreeDaySemesterGroupConstraint), so für jede Lesson die an diesem Tag stattfindet, der Wert der Constraint-Gewichtung dem Objective hinzugefügt werden. Dazu muss lediglich eine Variable erstellt

werden, welche die Anzahl an Lessons an diesem Wochentag zählt. An dieser Stelle soll nicht wie sonst, die Anzahl belegter Timeslots gezählt werden, sondern tatsächlich nur die Lessons selber. Die Implementierung befindet sich in `addFreeDaySemesterGroupVariables` in Datei ***SoftConstraints.py***.

Der gesamte Inhalt der Funktion ist in Listing 51 gezeigt. Die Variable *`free_day`* jedes `SemesterGroup` Objektes, zeigt an, ob und an welchem Wochentag möglichst keine Lessons mit der Semestergruppe stattfinden sollen. Ist der Wert der Variablen nicht `None`, enthält sie einen String, der den Wochentag angibt. Die erlaubten Strings sind in der Datei ***Timeslot.py*** als Konstanten zu finden.

Zunächst soll über alle Semestergruppen iteriert werden, die einen freien Tag wünschen. Daher wird in Zeile 1 geprüft, für welche `SemesterGroup` Objekte die Variablen nicht `None` ist. Um aus dem Wochentags-String die Nummer des Tages zu ermitteln, hilft die statische Funktion `getWeekdayID(str)` der Klasse `Timeslot`. (Zeile 2)

In Zeile 3 wird die `IntVar` Variable erstellt, die später die Anzahl an Lessons enthalten soll, die am jeweiligen Tag stattfinden. Anschließend werden die *`weekdayBoolVars`* aller `Lesson` Objekte der Lesson an denen die Semestergruppe teilnimmt, gesammelt. (Zeile 4) Diese geben jeweils an, ob die zugehörige Lesson am jeweiligen Tag stattfindet. Von diesen Variablen braucht nun lediglich die Summe gebildet, und der erstellten *`lessonsOnFreeDay`* Variable zugewiesen zu werden. (Zeile 5)

```
1 for group in [sg for sg in orm.getSemesterGroups() if sg.free_day is not None]:
2     freeDayId = Timeslot.getWeekdayID(group.free_day)
3     group.lessonsOnFreeDay = model.NewIntVar(0, group.max_lessons_per_day, "")
4     boolVarsOnFreeDay = [l.weekdayBoolVars[freeDayId - 1] for l in group.getLessons()]
5     model.Add(group.lessonsOnFreeDay == sum(boolVarsOnFreeDay))
```

Codeausschnitt 51: SoftConstraints.py - addFreeDaySemesterGroupVariables

Das Erstellen der Summanden für dieses Constraint, in der Funktion `createObjectiveFunctionSummands`, geschieht, in dem die einzelnen *`lessonsOnFreeDay`* Variablen mit dem Constraint-Gewicht multipliziert werden. (Listing 52 Zeile 3)

```
1 addFreeDaySemesterGroupVariables(model, orm)
2 for sg in filter(lambda sg: sg.free_day is not None, orm.getSemesterGroups()):
3     summands.append(sg.lessonsOnFreeDay * LESSONS_ON_FREE_DAY_PENALTY)
```

Codeausschnitt 52: SoftConstraints.py - createObjectiveFunctionSummands, FreeDaySemesterGroup

3.6 Abhängigkeiten Stundenplangrößen und mögliche Konfigurationen

Obwohl bei der Implementierung versucht wurde, diese möglichst unabhängig von dem konkreten Stundenplan der TH-Lübeck mit 5 Wochentagen und 6 Timeslots pro Tag zu gestalten, ist dies nicht an allen Stellen gelungen. Im Folgendem wird kurz auf die Constraints eingegangen, bei denen eine Abhängigkeit zu diesen beiden Werten besteht.

Im Anschluss wird kurz erläutert was geändert werden müsste, um einen Stundenplan mit einer anderen Anzahl Wochentage oder Timeslots pro Tag erstellen zu können.

Bei der Implementierung des Constraints MaxLecturesAsBlockTeacherConstraint, werden alle relevanten Blöcke aufgezählt, die an einem Tag mit 6 Timeslots auftreten können. Eine allgemeine Implementierung ist vermutlich möglich, jedoch wurde aus Zeitgründen verzichtet diese zu finden. Sollte ein Stundenplan mit mehr oder weniger als 6 Timeslots pro Tag verwendet werden, muss auf dieses Constraint verzichtet werden. Dazu genügt es den Funktionsaufruf der Funktion `addMaxLecturesAsBlockTeacherConstraints` in der Datei ***TimeTabling*** zu entfernen.

Für das Vermeiden der späten Timeslots (2.4.2 AvoidLateTimeslotsConstraint) werden derzeit Literale für den 5. und 6. Timeslots verwendet. Zwar ist die eigentliche Implementierung nicht mehr von diesen Abhängig, da auf spezialisierte Formen verzichtet wurde, die Constraints sind aber noch für den 5. und 6. Timeslot definiert. Um diese zu ändern, genügt es in der Funktion `createObjectiveFunctionSummands` der Datei ***SoftConstraints.py*** diese Literale zu ändern. Außerdem sollten die Namen der Gewichte für dieses Cosntraint geändet werden. Siehe dazu auch die Anmerkung in Abschnitt 3.5.2 CountLessonsAtHour.

Die Implementierung des Constraints AvoidGapBetweenLessonsSemesterGroupConstraint ist zwar unabhängig von der Anzahl Timeslots pro Tag, sollte diese aber erhöht werden, werden Lücken größer 4, die bei mehr als 6 Timeslots pro Tag auftreten können, nicht gezählt und vermieden werden können. Bei Tageslängen kleiner 6, gibt es keine Einschränkungen.

Gleiches gilt für das AvoidGapBetweenDaysTeacherConstraint. Bei mehr als 5 Arbeitstagen im Stundenplan, würden Lücken von mehr als 3 Tagen nicht erkannt werden. Weniger als 5 Tage sind ohne Anpassungen dieses Constraints möglich.

Änderung der Stundenplangrößen

Um Wochentage hinzuzufügen oder zu entfernen, muss der Inhalt der Tabelle *`timeslot`* in der Datenbank geändert werden. Die Timeslots neuer Tage müssen hinzugefügt werden. Beziehungsweise die Timeslots von zu entfernenden Tagen entfernt werden. Die IDs (Primärschlüssel) müssen aber in jedem Fall zählend von 1 an aufsteigen. Diese Folge darf auch keine Lücke enthalten.

In der Datei ***Timeslot.py*** müssen die Konstanten für die Wochentage angepasst werden. Dies beinhaltet auch den Inhalt der statischen Methode `getWeekdayID` in dieser Datei.

In der Datei ***ORM.py*** muss die Konstante *`WEEKDAYS`* angepasst werden. Sie enthält die Anzahl an Tagen im Stundenplan. In der gleichen Datei muss die Funktion `getTimeslotsPerDay` angepasst werden. Sollten neue Tage hinzugefügt werden, muss für diese ein Name für die Erstellung der Excel-Datei angegeben werden. Dies geschieht über die Variable *`dayMap`* in der Datei ***ExcelWriter.py***.

Um die Anzahl Timeslots pro Wochentag zu ändern, muss der Inhalt der Tabelle *timeslot* in der Datenbank geändert werden. Hier gelten die gleichen Vorgaben wie bei der Änderung der Tage.

In der Datei ***ORM.py*** muss die Konstante *TIMESLOTS_PER_DAY* entsprechend zur neuen Anzahl Timeslots geändert werde.

In der Datei ***Timeslots.py*** muss gegebenenfalls die Liste mit den Timeslots die vormittags stattfinden, angepasst werden. Diese ist dort in der statischem Methode `getForenoonTimeslotNumbers` definiert.

3.7 Anwendungshinweise

In diesem Abschnitt soll erklärt werden, was zu tun ist um das Programm zur Stundenplansuche verwenden zu können. Dies beinhaltet die Installation, Erstellung der Stundenplandaten und Programmoptionen.

3.7.1 Installation

Im folgendem werden die Schritte gezeigt, um das Programm zur Stundenplansuche verwenden zu können. Dies wird für das Betriebssystem Windows 10 gezeigt. Die Installation unter Linux Distributionen oder Mac OS, sollte aber sehr ähnlich funktionieren.

Windows 10
Da das Programm in Python implementiert ist, wird eine Python Installation benötigt. Da die OR-Tools Bibliothek die 64-bit Version erfordert, ist diese zu wählen. Die zum Zeitpunkt der Erstellung dieser Arbeit aktuellste Version ist 3.8. Die OR-Tools Bibliothek ist jedoch noch nicht kompatibel zu dieser Version. Daher sollte Python 3.7.4 installiert werden. Diese ist auf der offiziellen Python Website zu finden [Pyta]. Hier ist darauf zu achten die 64-bit Version zu wählen.

Wenn die Python Installation standardmäßig ausgeführt wurde, können die benötigten Python Bibliotheken mit dem Paketverwaltungstool pip, über die Konsole installiert werden. Wurde Python nicht den Umgebungsvariablen hinzugefügt, muss dazu ins Verzeichnis der Python Installation gewechselt werden, dies ist Standardmäßig:

C:\Users\%Username%\AppData\Local\Programs\Python\Python37

Die OR-Tools Bibliothek kann mit dem Befehl

python -m pip install –upgrade –user ortools

installiert werden. [Gooh] Bei der Entwicklung wurde die OR-Tools Version 7.2.6977 verwendet.

Anschließend werden noch die SQLAlchemy und die XlsxWriter Bibliothek benötigt:

python -m pip install sqlalchemy
python -m pip install xlsxwriter

Die OR-Tools Bibliothek benötigt eine Installation von Microsoft Visual Sudio (Community) 2017 um lauffähig zu sein. [Gooi] Möglicherweise wird auch nur die .Net Core-Runtime oder das .NET Framework SDK benötigt.

Um das Programm verwenden zu können, werden alle Python Dateien im Ordner ***TimeTabling*** und die SQLite Datenbank ***TimeTableData.db*** benötigt.

3.7.2 Verwendung

Um das Programm über die Konsole auszuführen, vom Ordner aus,in dem sich die Programmdateien befinden, die Datei ***TimeTableing.py*** mit Python ausführen. Zum Beispiel unter Windows, wenn Python den Umgebungsvariablen hinzugefügt wurde:

%PathToTimeTablingDir%>python TimeTabling.py

Das Programm gibt zunächst die aktuellen Programmparameter und Infos über die eingelesenen Daten aus der Datenbank aus. Nachdem das Modell für die Stundenplansuche erstellt und die Suche gestartet wurde, wird der Suchfortschritt ausgegeben. Für jede gefundene Lösung wird der Wert des Objectives durch eine horizontale Linie dargestellt. Nachdem die Suche beendet wurde, entweder, weil eine optimale Lösung gefunden wurde, oder die maximale Suchzeit abgelaufen ist, wird der gefundene Stundenplan auf der Konsole ausgegeben. Dies ist eine Ansicht aller Lessons, nicht nach Semestergruppen oder Lehrenden gefiltert. Abschließend folgen einige Daten über die Suche und über die gefundene Lösung. Für letzteres, Angaben, wie häufig die einzelnen optionalen Constraints verletzt sind.

Wird in der Datei ***TimeTabling.py*** der Wert der Konstanten *DEBUG* auf $True$ gesetzt, werden noch etwas ausführlichere Daten ausgegeben. Zum Beispiel Infos zu den einzelnen `SemesterGroup` und `Teacher` Objekten, wie viele Variablen und Constraint dem `CpModel` bei der Umsetzung des Stundenplans hinzugefügt wurden sowie eine Ausgabe der Statistiken des Erstellten `CpModels`. Außerdem wird der Stundenplan zusätzlich einzeln für jeden Lehrenden und jede Semestergruppe ausgegeben.

Programmoptionen

Es gibt einige Programmoptionen, die als optionale Argumente beim Aufruf der Datei ***TimeTabling.py*** angegeben werden können. Sie können alternativ auch direkt in dieser Datei geändert werden, wo sie als Konstanten angegeben sind. Wird ein Parameter angegeben, wird die jeweilige Konstante überschrieben.

Es wurde das Modul `argparse` verwendet um die Programmparameter einzulesen. [Pytb] Dies ermöglicht zum Beispiel eine Ausgabe aller Programm-Parameter durch das Argument *-h* oder *-help*. Folgend werden alle Parameter einmal erläutert.

- *-o {True, False}*: Ermöglicht die Angabe, ob die optionalen Anforderungen hinzugefügt werden sollen und nach einer optimalen Lösung gesucht werden soll.
- *-m N*: Angabe der Zeit, nach der die Suche spätestens abgebrochen werden soll. Angabe in Sekunden.

- *-p {0,1,2}*: Angabe, ob keine (0), die letzte (1), oder alle gefundenen Lösungen (2) auf der Konsole ausgegeben werden sollen.
- *-e*: Angabe, ob die letzte gefundene Lösung als Excel-Datei exportiert werden soll.
- *-u NAME*: Name der Hochschule für den Stundenplan. Wird für den Export der Excel-Datei verwendet.
- *-d NAME*: Name des Fachbereichs des Stundenplans. Für den Export der Excel-Datei.
- *-s NAME*: Name für das Semester des Stundenplans. Für den Export der Excel-Datei.

Folgend ist ein Programmaufruf mit den aktuellen Default Argumenten gezeigt. Diese Werte werden also verwendet, wenn man das Programm ohne Argumente aufruft.

TimeTabling.py -o True -m 300 -p 1 -e -u 'TH-Lübeck' -d 'Elektrotechnik' -s 'WiSe 2018-19'

3.7.3 Datenerstellung

Die Datei ***SampleDataGeneration*** enthält eine Hauptfunktion mit der Beispieldatensätze in die Datenbank geladen werden können. Neben der manuellen Eingabe in die Datenbank durch einen SQLite-Datenbank-Explorer, ist dies derzeit die einzige Möglichkeit um Stundenplandaten zu erzeugen. Es kann zwischen zwischen drei Datensätzen ausgewählt werden. Dazu einen der Funktionsaufrufe `generateSmallDataset(session)`, `generateSmallFullTestDataset(session)` und `generateBigDataset(session)` unkommentiert lassen. Um eigene Daten zu erstellen, bietet die Funktion `generateSmallDataset` das übersichtlichste Beispiel.

4 Abschlussbetrachtung

Die in Abschnitt 1 aufgeführten Ziele dieser Arbeit, konnten erreicht werden. Zunächst wurde ein Stundenplanmodell beschrieben, dass alle bekannten Anforderungen an den Stundenplan des Fachbereichs Elektrotechnik und Informatik, erfüllen kann. Dabei hat sich gezeigt, dass einige Besonderheiten und Ausnahmen, dieses Modell sehr komplex werden lassen können und Kompromisse zwischen einem möglichst flexiblen Modell und der damit einhergehenden Komplexität gefunden werden müssen.

Bei der Implementierung, konnte für alle Anforderungen eine zufriedenstellende Lösung gefunden werden. Sie wurden alle im Programm umgesetzt, und funktionieren wie gewünscht. Es hat sich gezeigt, dass die OR-Tools Bibliothek für so eine Aufgabe durchaus geeignet ist. Es wurde ein Teststundenplan erstellt, der mit 27 Lehrenden, 14 verschiedenen Räumen, 14 Semestergruppen und 61 Kursen, welche insgesamt 135 Lessons enthalten, welche wiederum 151 Timeslots belegen, einem Stundenplan eines kleinen Fachbereichs entspricht. Auf dem Rechner, der zum Erstellen dieser Arbeit genutzt wurde, wird mit dem erstellten Programm bereits nach einigen Sekunden, eine erste Lösung gefunden. In den folgenden Minuten, werden Lösungen, mit besseren Werten der Optimierungsfunktion gefunden. Ein Stundenplan, der nach einigen Minuten gefunden wurde, dürfte bereits eine sehr praxistaugliche Stundenplangüte aufweisen.

Jedoch wurde auch festgestellt, dass sich die Laufzeit, bis das Programm den optimalen Stundenplan findet, beziehungsweise bewiesen hat, dass die bisher gefundene Lösung optimal ist, mit steigender Größe des Stundenplans, sehr stark erhöht. Daher scheint es eher praktikabel, eine maximale Suchzeit von einigen Minuten einzustellen, um zwar keinen optimalen, jedoch einen gut optimierten Stundenplan zu erhalten.

4.1 Ausblick

Wie im zweiten Abschnitt dieser Arbeit gezeigt, bedarf besonders die Erstellung des zugrundeliegenden Stundenplanmodells, viel Planung und Überlegung. An einigen Stellen wurden alternative Ansätze vorgeschlagen, die vielleicht zu einem Modell führen würden, dass noch bessere Optimierungsmöglichkeiten bietet.

Ein praxisnaher Test, mit den genauen Stundenplandaten des Stundenplans eines Semesters des Fachbereichs, konnte nicht durchgeführt werden. Dazu wäre es nötig gewesen, die Ausgangsdaten eines Stundenplans zu erhalten. Nur dann würde die Stundenplanfülle und der Grad der Vernetzung aller Stundenplanobjekte, realitätsnah abgebildet werden können. Hier wäre interessant zu erfahren, wie sich die beste, gefundene Lösung, nach zum Beispiel 30 Minuten Suche, von dem tatsächlichen, manuell erstellten Stundenplan, unterscheidet. Dazu könnte beispielsweise das Objective des tatsächlichen Stundenplans ermittelt werden.

A Implementierende Funktionen, Übersicht

In Tabelle 11 ist eine Übersicht dargestellt, an welcher Stelle, d.h. in welcher Funktion, in der Python Datei HardConstraints.py, die jeweiligen harten Anforderungen umgesetzt werden.

Constraint Name	Implementierende Funktion
RoomTimeConstraint	addRoomTimeConstraints
TeacherTimeConstraint	addTeacherTimeConstraints
SemesterGroupTimeConstraint	addSemesterGroupTimeConstraints
PartSemesterGroupConstraint	addSemesterGroupTimeConstraints
StudyDayConstaint	addStudyDayConstraint
NotAvailableRoomTimeConstraint	addRoomNotAvailableConstraints
NotAvailableTeacherTimeConstraint	createLessonTimeAndRoomVariables
OnlyForenoonConstraint	createLessonTimeAndRoomVariables
LessonTakePlaceConstraint	createLessonTimeAndRoomVariables
LessonsAtSameTimeConstraint	createLessonTimeAndRoomVariables
CourseAsBlockConstraint	addCourseAllInOneBlockConstraints
BlockInSameRoomConstraint	addCourseAllInOneBlockConstraints
OneLessonPerDayConstraint	addMaxLessonsPerDayCourseConstraints
OneCoursePerDayTeacherConstraint	addOneCoursePerDayTeacherConstraints
MaxLessonsPerDayTeacherConstraint	addMaxLessonsPerDayTeacherConstraints
MaxLecturesPerDayTeacherConstraint	addMaxLecturesPerDayTeacherConstraints
MaxLecturesAsBlockTeacherConstraint	addMaxLecturesAsBlockTeacherConstraints
MaxLessonsPerDaySemesterGroupConstraint	addMaxLessonsPerDaySemesterGroupConstraints
MaxLessonsPerDayCourseConstraint	addMaxLessonsPerDayCourseConstraints
ConsecutiveLessonsConstraint	addConsecutiveLessonsConstraints

Tabelle 11: Harte Anforderungen Anforderungen und ihre Implementierungsfunktionen

In Tabelle 12 ist eine Übersicht der optionalen Anforderungen zu finden, zusammen mit der Python Funktion in der Datei SoftConstraints.py in der die Variablen zur Erfüllung dieser Anforderungen erstellt und ihnen die korrekten Werte zugewiesen werden. Das PreferFirstStudyDayChoiceConstraint bildet hierbei eine Ausnahme. Die nötigen Variablen werden bereits bei der Implementierung des TeacherStudyDayConstrains in der Datei HardConstraints.py erzeugt.

Constraint Name	Implementierende Funktion
PreferFirstStudyDayChoiceConstraint	addTeacherStudyDayConstraints
AvoidLateTimeslotsConstraint	addCountLessonsAtNthHour
AvoidEarlyTimeslotsConstraint	addCountLessonsAtNthHour
AvoidGapBetweenLessonsSemesterGroupConstraint	addGapVariables
AvoidGapBetweenDaysTeacherConstraint	addGapBetweenDaysTeacherVariables
FreeDaySemesterGroupConstraint	addFreeDaySemesterGroupVariables

Tabelle 12: Optionale Anforderungen und ihre Implementierungsfunktionen

Die in obigen Funktionen erzeugten Variablen, welche das Vorhandensein zu vermeidender Eigenschaften im Stundenplan anzeigen, werden in der Funktion `SoftConstraints.py - createObjectiveFunctionSummands` mit den jeweiligen Gewichten multipliziert und als Liste zurückgegeben. Diese Liste stellt in Form von Summanden die Optimierungsfunktion dar.

B Quellcode einzelner Funktionen

B.1 `createLessonTimeAndRoomVariables(...)`

Der gesamte Inhalt der Funktion `createLessonTimeAndRoomVariables`. Kommentare und nicht relevante Codeteile wie das Zählen neu angelegter Variablen wurden entfernt.

```
1  def createLessonTimeAndRoomVariables(model: CpModel, orm: ORM):
2      for course in orm.getCourses():
3          roomDomain = Domain.FromValues(map(lambda r: r.id, course.possible_rooms))
4
5          for lesson in course.lessons:
6              lesson.roomVar = model.NewIntVarFromDomain(roomDomain, "Course roomVar
               ↪  %i_%i" % (course.id, lesson.id))
7
8              if not hasattr(lesson, "timeVars"):
9                  lessonsSameTime = [lesson] + lesson.lessons_at_same_time
10                 maxLessonSize = max(map(lambda l: l.timeslot_size, lessonsSameTime))
11                 for l in lessonsSameTime:
12                     l.timeVars = []
13
14                 lastTimeVar = None
15
16                 for i in range(0, maxLessonSize):
17                     lessonsWithCurrentSize = list(filter(lambda l: l.timeslot_size > i,
                       ↪  lessonsSameTime))
18
19                     availableTimeslots = intersectAll(map(lambda l:
                       ↪  l.availableTimeslotsFiltered, lessonsWithCurrentSize))
20
21                     lastPossibleTimeslotNumber = TIMESLOTS_PER_DAY - maxLessonSize + i +
                       ↪  1
22                     timeslots = list(filter(lambda t: t.number <=
                       ↪  lastPossibleTimeslotNumber, availableTimeslots))
23
24                     timeDomain = Domain.FromValues(map(lambda t: t.id, timeslots))
25                     timeVar = model.NewIntVarFromDomain(timeDomain, "Lesson timeVar
                       ↪  %i_%i" % (course.id, lesson.id))
26
27                     for lessonSameTime in lessonsWithCurrentSize:
28                         lessonSameTime.timeVars.append(timeVar)
29
30                     if lastTimeVar:
31                         model.Add(timeVar == lastTimeVar + 1)
32                     lastTimeVar = timeVar
```

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[Lov10] April Lin Lovelace. *ON THE COMPLEXITY OF SCHEDULING UNIVERSITY COURSES*. 2010, S. 85–98. URL: https://pdfs.semanticscholar.org/ee3b/2fdc81deee071ad56137d0615e31c648bd14.pdf.

[Gooa] Google LLC. *Constraint Optimization*. URL: https://developers.google.com/optimization/cp. (Zugriff: 15.8.2019).

[Uni] UniTime LLC. *UniTime.org - Constraint Solver Library*. URL: https://www.unitime.org/index.php?tab=1. (Zugriff: 05.11.2019).

[Goob] Google LLC. *OR-Tools - Google Optimization Tools*. URL: https://github.com/google/or-tools#about-or-tools. (Zugriff: 02.10.2019).

[Gooc] Google LLC. *About OR-Tools*. URL: https://developers.google.com/optimization/introduction/overview. (Zugriff: 15.8.2019).

[Good] Google LLC. *Python Reference: CP-SAT*. URL: https://developers.google.com/optimization/reference/python/sat/python/cp_model. (Zugriff: 20.9.2019).

[Gooe] Google LLC. *Using the CP-SAT solver*. URL: https://github.com/google/or-tools/tree/master/ortools/sat/doc. (Zugriff: 02.10.2019).

[Goof] Google LLC. *cover_rectangle example -> BoolVar Assignment (Zeile 72-73)*. URL: https://github.com/google/or-tools/blob/stable/examples/python/cover_rectangle_sat.py. (Zugriff: 13.09.2019).

[Goog] Google LLC. *Python Reference: CPModel, LinearExpr*. URL: https://developers.google.com/optimization/reference/python/sat/python/cp_model#linearexpr. (Zugriff: 02.11.2019).

[Pyta] Python Software Foundation. *Python 3.7.4*. URL: https://www.python.org/downloads/release/python-374/. (Zugriff: 10.11.2019).

[Gooh] Google LLC. *Install OR-Tools*. URL: https://developers.google.com/optimization/install. (Zugriff: 10.11.2019).

[Gooi] Google LLC. *Install OR-Tools*. URL: https://developers.google.com/optimization/install/python/windows. (Zugriff: 10.11.2019).

[Pytb] Python Software Foundation. *argparse — Parser for command-line options, arguments and sub-commands*. URL: https://docs.python.org/3/library/argparse.html. (Zugriff: 29.10.2019).