- Bachelorarbeit -

# CATS - Ein System zur Stundenplanerstellung

(Constraint Aided Timetabling System)

vorgelegt von

**Timo Bozsolik**
**Lars Meyer**

Referent : Dr. Gunter Grieser

September 2007

Fachbereich Informatik
der Technischen Universität Darmstadt

---

**Erklärung**

Wir versichern hiermit, dass wir die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt haben.
Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht. Die Zeichnungen oder Abbildungen in dieser Arbeit sind von uns selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen. Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, den 20.09.2007

# Inhaltsverzeichnis

# Kapitel 1

# Einleitung

## 1.1 Motivation

Automatische Systeme zur Stundenplanerstellung sind mittlerweile Standard an Schulen. Die existierenden Systeme sind jedoch für kleine Schulen nicht geeignet, da sie auf der einen Seite zu unflexibel und auf der anderen zu unhandlich in der Bedienung sind. Dies führt dazu, dass diese Programme oft nur als Helfer einer dafür beauftragten Person dienen oder sich die betreffenden Schulen ganz gegen den Einsatz solcher Systeme stellen. So kommt es dazu, dass sich in vielen Bildungseinrichtungen regelmäßig eine Anzahl von Lehrkräften teilweise tagelang mit der reinen Erstellung eines Stundenplans beschäftigen. Die Ergebnisse dieser Arbeit hängen auf der einen Seite stark von der Erfahrung der entsprechenden Personen ab, während das Problem an sich rein kombinatorischer Natur ist.

Dieses Projekt wurde in Zusammenarbeit mit einer Grundschule in Darmstadt[1] realisiert, wobei sich schnell herausstellte, dass die Anforderungen an das zu implementierende System sehr speziell waren. In der Vergangenheit wurden dort mehrere verschiedene kommerzielle Systeme zur Stundenplanerstellung getestet, wovon keines den Ansprüchen dieser Schule gerecht werden konnte. Diese Programme hatten allesamt einen sehr großen Funktionsumfang, so dass deren Benutzung im Alltag schlicht zu aufwändig und kompliziert war. Auf der anderen Seite gab es auch bei dieser Vielfalt von Einstellungs- und Eingabemöglichkeiten immer noch Probleme, bestimmte Sachverhalte zu modellieren. Als Konsequenz daraus wurde in diesem Fall entschieden, vorerst auf eine computerbasierte Lösung zu verzichten und die Pläne weiter manuell zu erstellen. Diese Arbeit nimmt auch bei erfahrenen Personen mehrere Tage in Anspruch, bis der erstellte Plan allen gestellten Bedingungen entspricht und intuitiv möglichst „gut“ ist.

Gerade diese Frage nach der Güte der Stundenpläne wird bei vielen existierenden Programmen nicht ausreichend beleuchtet. Der Rechner kann bei einem solchen Problem auf-

[1] Die Ludwig-Schwamb-Schule in Eberstadt

tretende Konflike (zum Beispiel zeitliche Überschneidungen bei Räumen oder Lehrern) zwar wesentlich schneller und zuverlässiger erkennen als der Mensch, was zweifelsohne einen gültigen Stundenplan effizienter entstehen lässt. Allerdings ist dabei noch lange nichts über die Güte dieses Planes gesagt. Die Frage ist nun, wie man diese bestimmt bzw. einen Stundenplan eigentlich beurteilt. Dies ist für den Mensch meist intuitiv klar und beruht auf den eigenen Erfahrungen in der Schule bzw. der langjährigen Tätigkeit bei der Erstellung solcher Pläne. Diese Expertise nun so zu formulieren, dass ein Algorithmus damit arbeiten kann (etwa in Form einer Heuristik), stellt ein Kernproblem der heutigen Stundenplanerstellung sowie eine grundsätzliche Motivation für diese Arbeit dar. Meist werden solche Heuristiken in Studien an Schulen erhoben und enhalten dann auf diese Weise das Wissen der Experten. Im tatsächlichen Programm sieht das dann so aus, dass es eine Bewertungsfunktion gibt, welche durch mehr oder weniger Optionen gesteuert werden kann und dann zum Beispiel einen Zahlenwert für einen Stundenplan ausgibt. Beispielhaft wäre dann eine Komponente dieser Bewertungsfunktion diejenige, die die Anzahl der Freistunden für die Schüler analysiert und eine entsprechende Beurteilung abgibt.

Dieses Bewertungs-Modul ist normalerweise fest im Programm einprogrammiert, was die Auswertung zwar effizient, jedoch auch sehr unflexibel macht. Individuelle Anforderungen der Schule können so nicht mehr berücksichtigt werden, es sei denn durch einen Eingriff und eine Veränderung des Quellcodes. Verändern sich nun die Gegebenheiten oder weichen von der Norm ab, wird solch ein Programm für eine Schule unbrauchbar und es muss ein neuer Patch oder gar ein neues Produkt gekauft werden. Gleiches gilt, wenn ein neues Gesetz im Bildungswesen verabschiedet wird, durch welches in irgendeiner Art und Weise Bedingungen an die Stundenpläne der Schüler gestellt werden.

Daher wurde im Rahmen dieser Arbeit ein Programm entwickelt, welches es möglich macht, nahezu beliebige Bedingungen an die Erstellung eines Stundenplans zu stellen. In Form einer hierfür entwickelten Sprache ist der Mensch damit in der Lage, die Forderungen, die er an einen Plan stellt, zu formulieren. Auf der anderen Seite kann das Programm diese interpretieren und das Ergebnis dann nach diesen Kriterien optimieren. Dadurch ist das System gleichzeitig hochgradig variabel und kann sehr speziell an eine Schule angepasst werden werden. Da dieses Programm aber gerade für den alltäglichen Gebrauch entwickelt wurde, stehen hier auch Dinge wie die Benutzbarkeit, Erlernbarkeit, Akzeptanz und andere Prinzipien der Usability im Vordergrund. Auf diese Weise soll ein Mittelweg zwischen der wissenschaftlich fundierten und flexiblen Suche nach dem bestmöglichen Stundenplan für die gegebenen Randbedingungen und einer einfachen und intuitiven Benutzung des Programms gefunden werden.

## 1.2 Das Stundenplanproblem

In diesem Kontext beschreibt die Lösung des Stundenplanproblems die Technik, einen *Schulstundenplan* zu generieren und aufgrund diverser Kriterien zu optimieren. Andere ähnliche Probleme wären zum Beispiel das *Kurswahlproblem* oder das der *universitären Veranstaltungsplanung*. Ein *Schulstundenplan* regelt, zu welchen **Terminen** oder **Zeiteinheiten** welche **Klassen** bei welchem **Lehrer** in welchem **Raum** in welchem **Fach** unterrichtet werden. Dabei sind sowohl die Lehrer, als auch die Klassen und Räume beschränkte Ressourcen, deren maximale oder exakte Auslastung (z.B. Arbeitsstunden der Lehrer, Unterrichtsstunden der Klasse) festgelegt ist und eingehalten werden muss. Außerdem sind mögliche Konflikte (z.B. ein Lehrer unterrichtet zu einem Termin in zwei verschiedenen Räumen) zu vermeiden, die Ressourcen dürfen also nur exklusiv genutzt werden.

Im einfachsten Fall wird also eine Menge von Tupeln der Form $(Zeitslot, Klasse, Lehrer, Raum, Fach)$ gesucht (die *Stundenplaneinträge* oder *Meetings*), so dass die oben genannten Beschränkungen erfüllt werden und dabei keine Konflikte auftreten. In der Praxis exisitieren jedoch zusätzlich dazu noch diverse Nebenbedingungen. Dabei nennt man diejenigen Bedingungen, die die *Gültigkeit* eines Stundenplanes festlegen und folglich zwingend gelten müssen, *harte* Constraints und solche, die die *Güte* eines Plans bestimmen (also möglichst gelten sollten), *weiche* Constraints.

Eine gute und umfassende Übersicht über das Stundenplanproblem im Allgemeinen, die verschiedenen Ausprägungen, über bekannte Lösungsansätze und eine fundierte theoretische Analyse liefert [Löh98].

### 1.2.1 Generierung und Optimierung

Bei der Erstellung eines Stundenplanes, sieht man sich zunächst mit dem Problem konfrontiert, überhaupt einen *gültigen Stundenplan* zu erstellen (also einen, der die *harten* Bedingungen erfüllt, so dass es bspw. keine Ressourcenkonflikte und Überschreitungen der Kapazitäten gibt). Dies wird mit dem *Existenz-* oder *Generierungsproblem* bezeichnet. Weiterhin existiert in diesem Kontext das sogenannte *Optimierungsproblem*, dessen Lösung es zum Ziel hat, durch die Erfüllung möglichst vieler *weicher* Constraints den möglichst besten (gültigen) Stundenplan zu finden (siehe auch [Sch99]). Beispiele für solche Kriterien sind etwa, dass die einzelnen Klassen möglichst wenige Freistunden haben oder dass der Unterricht möglichst oft zur ersten Stunde beginnen soll.

Das Finden eines gültigen Stundenplans kann man hier als rein *kombinatorisches Problem* ansehen, welches von einem Rechner leicht unter Beachtung der genannten Parameter gelöst werden kann. Ein denkbarer Algorithmus wäre zum Beispiel das zufällige

Platzieren der Fächer für eine Klasse in noch freie Zeiteinheiten mit anschließender und wiederum zufälliger Zuordnung der Lehrer und Räume. Offensichtlich liefert dieser Algorithmus bei prinzipiell ausreichenden Ressourcen (gesamte Stundenanzahl der Klassen darf Lehrerstundenanzahl nicht überschreiten usw.) eine gültige Lösung, die jedoch je nach den angelegten Kriterien nicht besonders optimal sein wird (z.B. wird der Algorithmus bei spärlicher Besetzung eines Plans vorraussichtlich viele Freistunden erzeugen).

Die Selektion oder Generierung eines *optimalen Stundenplans*, der möglichst viele *weiche* Constraints erfüllt, stellt sich für einen Rechner schon wesentlich schwieriger dar. Zwar gibt es auch hier die verschiedensten Ansätze, das *Optimierungsproblem* zu lösen, jedoch sind die individuellen und stark vom Kontext einer jeden Schule abhängigen Kriterien meist nur durch manuelle Eingriffe oder gänzliche manuelle Planung erfüllbar. Dies liegt zum einen daran, dass formale Lösungsansätze um ein gewisses Maß an Allgemeinheit bemüht sind, während die mit der Planung beauftragten Personen spezifischere und damit einfacherere Lösungen anstreben können. Weiterhin besteht das Problem der mangelnden Kommunikation zwischen Mensch und Algorithmus. Dem Mensch ist meist intuitiv klar, was ein guter Plan ist, wie man diesen erstellt und welches überhaupt die Kriterien für die Bewertung eines Planes sind. Ein Algorithmus muss dann entweder von vorneherein so gestaltet sein, dass er diese Kriterien erfüllt (was relativ unflexibel ist), oder vor (oder sogar während) der Generierung vom Mensch vorgeschrieben bekommt. Mit genau dieser Problematik beschäftigt sich ein Großteil dieser Arbeit.

### 1.2.2 Teilprobleme

Zur besseren Verständlichkeit und Lösbarkeit kann man das Stundenplanproblem in verschiedene Teilprobleme zerlegen, welche einzeln und nacheinander gelöst werden können.

Die verschiedenen Teilprobleme sind im Einzelnen (siehe [Wei99] oder [Bar96]):

**Faculty Timetabling:** Die verfügbaren Lehrer werden nach ihren Fähigkeiten (also den Fächern, die sie unterrichten können, und der Anzahl ihrer verfügbaren Stunden) auf eine Menge von Unterrichtsstunden der Klassen verteilt.

**Class-Teacher Timetabling:** Für jede Klasse mit einem festen Lehrplan wird die zeitliche Planung der einzelnen Stunden so vorgenommen, dass es für Lehrer und Klassen keine zeitlichen Konflikte gibt, vorallem also keine Doppelzuordnungen.

**Course Scheduling:** Dies ist die Planung der nicht in kompletten Klassen organisierten Schüler und der betreffenden Fächer in Zusammenhang mit den zur Verfügung stehenden Lehrern. Hierbei muss darauf geachtet werden, dass sich die aktuelle Stundenanzahl jedes Schülers sich nicht aus der zugeordneten Klasse sondern den Kursen, die er besucht, ergibt.

**Classroom Assignment:** Damit bezeichnet man die Zuordnung eines ansonsten spezifizierten Lehrplanes (also einer Anzahl an Unterrichtsstunden der Klassen) auf die zur Verfügung stehenden Räume. Dabei muss je nach Fach berücksichtig werden, ob dafür spezielle Räumlichkeiten vorgesehen sind (z.B. Sport in der Turnhalle).

Ein weiteres verwandtes Problem in diesem Kontext ist das Problem des **Examination Timetabling**, welches die Verteilung der Prüfungen für eine Menge von Klassen / Kursen regelt. Dieses Problem ist jedoch hier nicht relevant und wird im Folgenden nicht näher betrachtet. Je nach den gestellten Anforderungen an einen Algorithmus werden durch diesen dann ein oder mehrere dieser Teilprobleme entweder isoliert oder gleichzeitig gelöst, siehe dazu auch Kapitel 2.2.

### 1.2.3 Komplexität des Problems

Die größte Schwierigkeit, einen optimalen Stundenplan für eine Menge an Klassen, Lehrern usw. zu finden, liegt in der Komplexität des Problems. Möchte man lediglich eine konfliktfreie Zuordnung von Lehrern und Klassen auf die zur Verfügung stehenden Zeitslots vornehmen (ohne jegliche Nebenbedingungen), kann man dies je nach verwendetem Verfahren noch effizient in polynomieller Zeit erreichen. Nimmt man einfache Constraints wie die Beschränkung der Verfügbarkeiten von Lehrern hinzu, so wird schon das *Existenzproblem* $\mathcal{NP}$-vollständig (siehe z.B. [Sch99]) und damit für reale Datensätze nicht mehr effizient lösbar. Selbst für die meisten Teilprobleme kann schon die $\mathcal{NP}$-Vollständigkeit nachgewiesen werden, was eine exakte Lösung des Problems generell ausschließt und die Verwendung von approximativen Verfahren nötig macht. Sucht man nämlich im Lösungsraum nach der besten Lösung und versucht das *Generierungsproblem* zu lösen, wird schnell klar, dass die komplette Generierung aller gültigen Pläne mit anschließender Auswahl des besten Planes (*Brute-Force-Methode*) auch schon bei kleinen Problemen nicht mehr in vertretbarer Zeit zu realisieren ist. Eine ausführliche Analyse der Komplexität der einzelnen Teilprobleme findet sich in [EIS76] oder [CK96]. Zwar sind diese Analysen meist sehr theoretisch und nicht direkt auf die Praxis anwendbar, da die in der Realität hinzukommenden Nebenbedingungen den Suchraum effektiv einschränken, jedoch ändert dies nichts an der Komplexität im Allgemeinen. Es kann je nach dem Grad, in dem die verschiedenen Nebenbedingungen miteinander konkurrieren, sogar vorkommen, dass diese die Laufzeit (wenn auch meist nur asymptotisch in der Komplexität) erhöhen.

## 1.3 Übersicht

Das Ziel dieser Arbeit ist es, das in diesem Rahmen entwickelte Stundenplan-Programm detailliert zu beschreiben und die Vorgehensweise zu erläutern. Dafür gliedert sich die Arbeit in drei große Teile. Der erste Teil beschäftigt sich mit der theoretischen Sicht des Problems und den getroffenen Vorüberlegungen; er umfasst die Kapitel 1 und 2. Im zweiten Teil, welcher aus den Kapiteln 3 und 4 besteht, wird detailliert auf die Umsetzung des Projektes eingangen. Der letzte Teil enthält die Kapitel 5 und 6 und behandelt die Sicht des Programmes für den Nutzer und bewertet dieses anhand diverser Kriterien.

Die Motivation und eine Definition des Stundenplanproblems auf abstrakter Ebene wurden bereits in Kapitel 1 gegeben. In Kapitel 2 findet sich zunächst eine Anforderungsanalyse für das konkrete Programm, welche die Probleme und Erfordernisse im speziellen Fall aufzeigen soll. Weiterhin enthält dieses Kapitel eine Übersicht über bekannte Lösungswege und deren Vor- und Nachteile. Am Ende des Kapitels werden dem Leser dann die Entscheidungen, die aufgrund der vorhergehenden Betrachtungen getroffen wurden, erläutert und der Weg für den konkreten Lösungsansatz abgesteckt. In Kapitel 3 wird dann der gewählte Lösungsweg beschrieben, wobei zunächst eine Übersicht über die benötigten Komponenten und deren Zusammenhänge gegeben wird und dann die einzelnen Bestandteile näher beleuchtet werden. Hierbei werden die angewandten Verfahren und Konzepte ausführlich beschrieben, auf die konkrete Implementierung (also speziell Datenstrukturen u.ä.) wird jedoch nicht eingegangen.

Während in Kapitel 3 das System zur Bewertung der Stundenpläne als Black-Box betrachtet wurde und für diesen Lösungsweg durchaus austauschbar ist, widmet sich Kapitel 4 dessen konkreter Umsetzung. Diese Trennung erfolgte, weil das Bewertungs-System als eigenes Subsystem angesehen und auch für einen anderen Kontext isoliert eingesetzt werden kann. Kapitel 5 setzt sich mit der praktischen Benutzung des Systems auseinander und stellt die grafische Oberfläche ausführlich dar. Weiterhin wird darin eine Evaluierung des erstellten Programmes sowohl aus technischer als auch aus Benutzersicht durchgeführt. Eine Zusammenfassung in Kapitel 6 rundet diese Arbeit ab.

Da diese Arbeit das Werk von zwei Autoren ist, soll hier kurz aufgezeigt werden, welche Teile von welchem Verfasser stammen. Kapitel 1, 2 und 3 wurden hauptsächlich von Timo Bozsolik angefertigt, während Kapitel 4, 5 und 6 größtenteils Lars Meyer zuzuordnen sind. Natürlich ist diese Abgrenzung nicht starr zu sehen, die Verantwortung für die komplette Arbeit sowie die Implementierung liegt bei beiden Autoren gleichermaßen.

# Kapitel 2

# Problemstellung und Lösungsansatz

## 2.1 Anforderungen

Grundsätzlich beschränkt sich diese Arbeit im Gegensatz zu vielen sich im Einsatz befindlichen Systemen nicht auf die Lösung eines Teilproblems. Eine Anforderungsanalyse hat ergeben, dass in diesem Kontext alle in Kapitel 1.2.2 angesprochenen Probleme (außer dem **Examination Timetabling**) gleichzeitig, also das Stundenplanproblem als ganzes gelöst werden muss.

Dieses (wie zu Anfang in Kap. 1.2 definiert) trifft zunächst jedoch nur Aussagen über die Zuordnung von **Klassen**, **Lehrern**, **Fächern** und **Räumen** auf die zur Verfügung stehenden **Zeitslots**. Eine Analyse und eingehende Befragung der Person, die in der im Rahmen dieser Arbeit betreuten Schule für die Stundenplanerstellung zuständig ist, hat jedoch ergeben, dass dieses Modell in der Praxis nicht ausreichend ist und nur als Grundlage genommen werden kann. So müssen einerseits bei der Erstellung der Stundenpläne auf die verschiedensten Nebenbedingungen geachtet werden, andererseits müssen seitens des Programmes weitere Features zur Verfügung gestellt werden, die es erlauben, den vorhandenen Kontext so gut wie möglich zu modellieren. Im Folgenden werden die wichtigsten dieser Bedingungen erläutert sowie Vorüberlegungen zur Umsetzung dieser Erfordernisse dargestellt.

### 2.1.1 Strukturelle Rahmenbedingungen

Viele Anforderungen an die generierten Stundenpläne lassen sich zwar als Bedingung an den Plan an sich formulieren, dennoch muss das System strukturelle Dinge erfüllen, die es überhaupt erst möglich machen, den gegebenen Kontext darzustellen. Insbesondere ist hier die Anreicherung der Stammdaten (also den Lehrern, Fächern, Klassen etc.) mit zusätzlichen Informationen hervorzuheben.

**Zuordnung zu Jahrgangsstufen** Um die erforderliche Anzahl an Unterrichtsstunden festzulegen, ist eine Zuordnung der Klassen in verschiedene Jahrgangsstufen unerlässlich. Für jede Stufe kann dann für jedes Fach eine Anzahl abzuleistender Stunden definiert werden. Auch für das folgende Problem der Bandstunden ist eine solche Zuteilung nötig.

**Bandstunden** Zusätzlich zur normalen Verteilung der Schulstunden muss auf die sogenannten Bandstunden Rücksicht genommen werden. Diese sind für spezielle Fächer wie zum Beispiel Religion vorgesehen, welche von allen Klassen einer bestimmten Stufe zur gleichen Zeit abgehalten werden sollen. Innerhalb dieser Stunden kann der Klassenverband aufgelöst und der Unterricht in gemischten Gruppen abgehalten werden, wie zum Beispiel in evangelischer und katholischer Religion. Unter diesem Gesichtspunkt wurde das Konzept der Gruppen, welche aus mehreren Klassen gebildet werden können, eingeführt. Diese Gruppen werden dann jeweils für ein spezielles Fach erstellt, damit das System alle Gruppen des gleichen Faches und der gleichen Stufe identifizieren und auf den gleichen Zeitslot legen kann.

**Fachspezifische Räume** Außer den normalen Klassenräumen, in denen jedes Fach unterrichtet werden kann, exisitieren bestimmte fachspezifische Räume, die nur für spezielle Fächer zu Verfügung stehen. Auf der anderen Seite gibt es spezielle Fächer, die nur in dafür vorgesehenen Räumen unterrichtet werden dürfen. Beispiele dafür wären etwa das Fach Sport, welches nur in der Turnhalle abgehalten werden darf, oder die explizite Verwendung von Musikräumen. Die Einführung von Raumtypen und die Aufteilung in normale und spezielle Fächer (solche, die einen bestimmten Raumtypten vorraussetzen) wird dieser Problematik gerecht.

**Verfügbarkeit von Räumen und Lehrern** Für jeden Lehrer und jeden Raum sollte eine Menge von Zeitslots definierbar sein, zu welchen diese nicht verfügbar sind und somit nicht eingeplant werden dürfen. Die gemeinsame Nutzung einer Turnhalle mit einer anderen Schule ist ein Beispiel für eine solche Situation. Implementiert werden kann dies durch einfache Verfügbarkeitslisten, die der Benutzer dann entsprechend editieren kann.

**Festsetzen von Fächern** Das Programm sollte die Möglichkeit bieten, vor der Generierung eines Stundenplans bestimmte Fächerzuordnungen manuell zu setzen. Durch eine entsprechende Festsetzungsoption sollten diese dann von einer Verschiebung bzw. Vertauschung im Zuge des Optimierungsprozesses ausgenommen bleiben. Auch die nachträgliche Verschiebung von Fächern und das neue Anstoßen des Optimierungsprozesses auf Basis dieser Information sollte möglich sein.

Weitere Bedingungen, die das fertige Programm erfüllen muss, um dessen praktischen Nutzen zu gewährleisten, sind unter anderem:

- Jedem Lehrer sollte eine gewisse Menge von Fächern zugewiesen werden können, zu deren Unterricht er befugt ist. Während an der betrachteten Grundschule die meisten Lehrer zwar alle „gewöhnlichen“ Fächer unterrichten dürfen, gibt es auch hier zum Beispiel Sportlehrer, die nur dieses Fach lehren.
- Es gibt spezielle Differenzierungs-Stunden, zu denen nicht die komplette Klassse anwesend sein muss, sondern nur ein vorher vom Lehrer bestimmter Teil. Da diese Bedingung die eigentliche Generierung der Pläne nicht betrifft, reicht es in diesem Fall aus, diese Stunden als solche zu kennzeichnen.
- Im Falle von Referendaren, die von einem regulären Lehrer betreut werden, müssen für eine Stunde mehrere Lehrer eingeteilt werden. Gleiches gilt auch für Präventionslehrer, die zusätzlich zum Klassenlehrer den Unterricht besuchen.
- Ein Lehrer muss eine bestimmte Anzahl von Unterrichtsstunden erfüllen, welche individuell für jeden Lehrer festgelegt werden muss. Abhängig von dieser Stundenzahl nimmt ein Lehrer dann eine volle oder eine halbe Stelle ein, in welchem Fall er Anspruch auf einen freien Tag in der Woche hat.

### 2.1.2 Anforderungen an generierte Pläne

Die bisher genannten Rahmenbedingungen sind vor allem der Natur, dass sie sich voraussichtlich nicht mehr ändern und strukturelle Erfordernisse an das System stellen. Darüber hinaus gibt es eine Reihe von Bedingungen, die man an einen fertigen Stundenplan stellen kann, die sich dynamisch ändern und die von Schule zu Schule sehr speziell sein können. Diese können generell die Gültigkeit von Plänen oder aber einfach nur die Qualität eines Planes beschreiben (*Muss-* und *Soll-Funktionalität*), dabei sind die verschiedensten Anforderungen denkbar.

**Freistunden für Schüler:** Da die in diesem Projekt betreute Schule eine Grundschule ist, dürfen die einzelnen Klassen aus Gründen der Aufsichtspflicht keinerlei Freistunden haben. Diese Bedingung sollte aber nicht grundsätzlich und fest einprogrammiert sein, da eine mögliche Benutzung des Systems an anderen Schulen dies erlauben könnte.

**Freistunden für Lehrer:** Freie Stunden für Lehrer sollten zwar prinzipiell möglich sein, jedoch sollte deren Anzahl wenn möglich minimiert werden. Andererseits sollte es auch möglich sein, Freistunden für Lehrer zu fordern, damit diese sich auf den nachfolgenden Unterricht vorbereiten können.

**Verteilung der Unterrichtsstunden:** Es sollten Aussagen darüber gemacht werden können, wie die Unterrichtsstunden auf die Anzahl der zur Verfügung stehenden Zeits-

lots zu verteilen sind. So könnte zum Beispiel ein Beginn des Unterrichts zur ersten Stunde jedes Tages gewünscht sein oder aber möglichst wenige Stunden am Anfang oder Ende einer Woche.

**Resistenz des Planes gegenüber Erkrankungen:** Ein Plan sollte möglichst so beschaffen sein, dass sich im Falle einer dauerhaften Erkrankung eines Lehrers und einer Umplaung des Unterrichts möglichst wenige Änderungen für die Schüler ergeben. Eine weitere, noch schwieriger umzusetzende Bedingung könnte zum Beispiel sein, dass eine Grippewelle unter den Lehrern den bestehenden Plan möglichst wenig gefährden darf.

**Präferenzen über die Fächerverteilung:** Während bisher angenommen wurde, dass die Verteilung der Fächer über die einzelnen Zeitslots nicht von Bedeutung ist, sollten in der Praxis darüber sehr wohl Aussagen getroffen werden können. So sollten etwa nicht mehr als 2 Stunden des gleichen Faches hintereinander unterrichtet werden und Sport zusammenhängend abgehalten werden.

**Präferenzen über die Zuteilung von Lehrern:** Auch die Verteilung der Lehrer auf die verschiedenen Klassen und Zeitpunkte sollte variabel bestimmbar sein. So wäre ein Plan, der den Klassen hauptsächlich aus den Vorjahren bekannte Lehrer zuordnet, besser als einer, der dies eben nicht beachtet.

Diese beispielhafte Auflistung macht schnell klar, dass viele Bedingungen, die einen guten Stundenplan ausmachen, meist intuitiv in unserem Verständnis vorhanden sind. Für den Rechner müssen diese Bedingungen (im Folgenden **Constraints** genannt) jedoch explizit in einer für diesen interpretierbaren Weise formuliert werden. Wie dies hier umgesetzt wurde, wird in Kapitel 4 beschrieben.

### 2.1.3 Nichtfunktionale Anforderungen

Da das in diesem Projekt entstandene System nicht nur ein akademischer Prototyp, sondern vorallem in der Praxis eingesetzt werden soll, muss das Programm zudem weitere nichtfunktionale Bedingungen erfüllen, die die alltägliche Benutzung gewährleisten sollen. Diese Bedingungen sind vorallem dem Bereich der Usability zuzuordnen und werden in ISO / IEC 9126 (siehe [ISO91] [1]) und DIN 66272 spezifiziert. Sie sind in verschiedene Bereiche unterteilt, von denen die wichtigsten hier kurz dargestellt werden.

**Benutzbarkeit:** Diese Kriterien beschreiben, den Aufwand, den der Benutzer hat, um das System sinnvoll einzusetzen. Dazu zählt die unmittelbare **Bedienbarkeit**, die **Verständlichkei**t seitens des Benutzers, die **Erlernbarkeit** für den Benutzer, die **Attraktivität** des Programmes und dessen **Konformität** mit bestehenden Prinzipien in einschlägig bekannten Anwendungen.

[1] z.B. verfügbar unter http://de.wikipedia.org/wiki/ISO_9126

**Funktionalität:** Das fertige Endprodukt soll die geforderten Funktionen mit den festgelegten Eigenschaften erfüllen. Um dies zu erreichen, ist vorallem die **Richtigkeit** und die **Angemessenheit** (der Aufwand für die Erledigung einer Aufgabe) des Programmes zu gewährleisten. Die Kriterien der **Interoperabilität** und der **Sicherheit** spielen in diesem Kontext keine entscheidende Rolle.

**Zuverlässigkeit:** Dies ist die Fähigkeit einer Software, ein bestimmtes Leistungsniveau zu etablieren und über einen bestimmten Zeitraum hinweg aufrecht zu erhalten. Dazu gehören die generelle **Reife** des Systems, die **Robustheit** gegenüber unvollständigen oder gar falschen Eingaben, die **Fehlertoleranz** und die **Wiederherstellbarkeit** von Daten im Falle einer Ausnahmesituation.

**Änderbarkeit:** Der Aufwand, die bestehende Software an andere Gegebenheiten anzupassen, Funkitonalität zu ändern oder neue zu integrieren, sollte möglichst gering sein. Um dies sicherzustellen, sind die Prinzipien der **Modifizierbarkeit**, der **Stabilität** der einzelnen Module, der **Analysierbarkeit** und der **Prüfbarkeit** einzuhalten.

**Effizienz:** Das Verhältnis zwischen der erbrachten Leistung von Programmfunktionen und deren verbrauchten Ressourcen sollte möglichst gering sein. Dafür muss das **Zeitverhalten** sowie das **Verbrauchsverhalten** von Programmen möglichst niedrig gehalten werden.

Im Rahmen dieses Projektes liegt die Priorisierung dieser Bedingungen klar auf der Benutzbarkeit und der Zuverlässigkeit; um diese zu gewährleisten, ist jedoch die Beachtung der Funktionalität und der Änderbarkeit unabdinglich. Wie sich im Entwicklungsprozess herausstellte, ist auch die Effizienz in diesem Kontext von großer Wichtigkeit und darf nicht außer Acht gelassen werden.

## 2.2 Bekannte Lösungsansätze

Die Vergangenheit und die bisherige Forschung haben die verschiedensten Arten an Lösungsmöglichkeiten für das Stundenplanproblem hervorgebracht. Entsprechend groß ist auch die Vielzahl an (oft nur in akademischen Prototypen vorliegenden) Programmen, die diese Lösungsmöglichkeiten implementieren und für diesen Weg typische Vor- und Nachteile aufweisen (siehe [Sch99]).

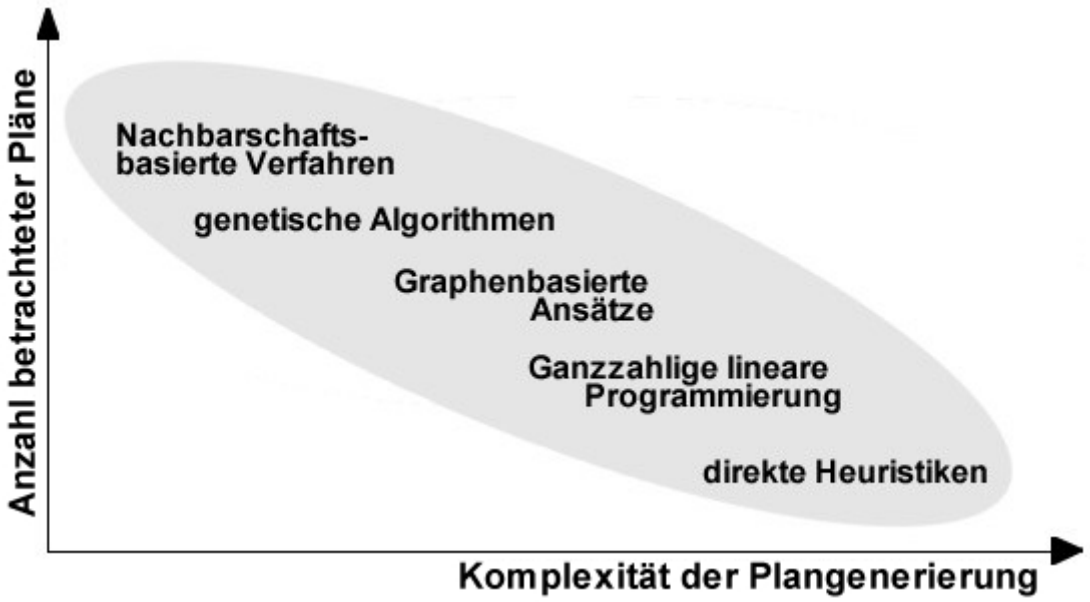


Abbildung 2.1: Betrachtete Stundenpläne vs. Komplexität der Plangenerierung

Alle Verfahren lassen sich in Bezug auf die Komplexität der Erstellung eines Stundenplanes und die während der Generierung betrachtete Anzahl an Plänen kategorisieren, was Grafik 2.1 verdeutlichen soll. Beispielsweise ist es im Falle der *direkten Heuristiken* sehr komplex, überhaupt eine gültige Lösung zu finden, jedoch wird im Zuge der Lösungsfindung nur ein einziger kompletter Plan untersucht. Im Falle einer *Lokalen Suche* als nachbarschaftsbasiertem Verfahren ist die Generierung einer (bzw. der nächsten) gültigen Lösung zwar relativ einfach, jedoch werden hier zum Auffinden einer möglichst guten Lösung sehr viele verschiedene Stundenpläne betrachtet. Deshalb ist es auch schwer, von der Güte der einzelnen Verfahren zu sprechen, da man hier immer diesen Tradeoff beachten muss.

### 2.2.1 Exakte Methoden

Wie schon in Kapitel 1.2.3 beschrieben, ist die exakte Bestimmung der besten Lösung (indem man alle möglichen Pläne generiert und daraus den besten auswählt) durch die

$\mathcal{NP}$-Vollständigkeit des Existanzproblems nahezu unmöglich, was diese Verfahren für die Realität unbrauchbar macht. Eine Zerlegung in Teilprobleme (siehe Kap. 1.2.2) würde die Laufzeit zwar stark verbessern, jedoch führt die starke Verflechtung zwischen den Teilproblemen in der Praxis zu Schwierigkeiten. Eine Bedingung, die Aussagen über die zu verteilenden Einheiten mehrerer Teilprobleme trifft (etwa könnte ein Lehrer bevorzugt in einem bestimmten Raum unterrichten möchten), kann meist in keiner der betreffenden Suchphasen berücksichtigt werden. Außerdem ist das Optimum in den einzelnen Teilproblemen noch lange kein Anzeichen dafür, dass das Gesamtproblem optimal gelöst ist.

### 2.2.2 Direkte Heuristiken

Diese Verfahren basieren allesamt auf Variationen des *Greedy-Algorithmus* und benötigen eine Heuristik, die einen (bis dahin evtl. nur teilweise generierten Stundenplan) auf bestimmte Kriterien hin bewertet. Der Greedy-Algorithmus wählt dann jeweils für die nächste Platzierung eines Eintrages im Stundenplan die Variante mit der besten Bewertung aus. Da es bei dieser Technik vorkommen kann, dass die Suche im Suchbaum in eine Sackgasse läuft, aus der man zu keiner gültigen Lösung mehr kommen kann, ist hier ein *Backtracking* nötig. Hierbei springt der Algorithmus wieder ein oder mehrere Stufen im Generierungsprozess zurück, um dann mit anderen Kombinationen der Platzierung doch noch eine gültige Lösung zu finden. Fast sämtliche kommerziell angebotenen Systeme beruhen auf diesem Verfahren oder Variationen davon. Im Allgemeinen bringt dies eine gute Laufzeit mit sich, jedoch können vorab keine Aussagen über die Qualität der so gewonnenen Stundenpläne getroffen werden, da eine einmal vom Greedy-Algorithmus getroffene Entscheidung im Regelfall nicht rückgängig gemacht wird.

Die Kriterien für die Heuristik sind in diesen Fällen meist mit sehr wenigen oder gar keinen Optionen im System fest einprogrammiert, was sich als unflexibel herausstellt und eine grundsätzliche Motivation für diese Arbeit ist. Eine Liste der Algorithmen, die dieses Prinzip nutzen, findet sich in [SS79]; Programme, die dieses umsetzen, sind etwa SCHOLA (siehe [USK69]) oder aSc-Stundenpläne[2], welches zur Zeit weit verbreitet ist.

### 2.2.3 Nachbarschaftsbasierte Lösungen

Im Gegensatz zum *Greedy-Verfahren* benutzen nachbarschaftsbasierte Lösungen zwar auch Heuristiken, um die generierten Pläne zu erstellen, jedoch beschränken diese Verfahren sich nicht auf die Generierung eines einzelnen, gültigen und möglichst guten

[2] http://www.asctimetables.com/

Planes. Vielmehr werden hier ausgehend von einer initialen Startlösung weitere Lösungen erstellt und deren Güte bewertet, um so zu einem möglichst guten Stundenplan zu gelangen. Es handelt sich hier also um ein reines *Optimierungsproblem*, welches die Lösung des *Generierungsproblems* voraussetzt.

Um durch den Raum der gültigen Lösungen iterieren zu können, ist es nötig eine Nachbarschaftsrelation auf gültigen Lösungen zu definieren. Bei Stundenplanproblemen könnten so zum Beispiel zwei Pläne im Lösungsraum benachbart sein, wenn sie sich bis auf die Änderung eines Raumes einer einzelnen Stunde nicht unterscheiden. Mögliche Suchverfahren können dann die jeweiligen Nachbarschaften nach optimalen Lösungen durchsuchen; benötigt wird hierfür eine gültige (aber womöglich schlechte) Startlösung, eine Heuristik zur Bewertung der Pläne sowie eine Definition der Nachbarschaft bzw. ein Verfahren, durch diese zu iterieren.

Diese Klasse der Suchverfahren ist die mit Abstand vielseitigste und flexibelste, da es hierfür viele verschiedene und generisch verwendbare Suchverfahren gibt. Beispiele für solche Suchverfahren sind etwa die *Lokale Suche*, *Simulated Annealing* oder die *Tabu-Suche*. Alle diese Verfahren wurden bereits in verschiedenen Variationen auf das Stundenplanproblem angewandt, Beispiele sind in [SOS05] oder [ADK99] zu finden. Da einige dieser Verfahren im Rahmen dieses Projektes implementiert wurden, findet sich eine detaillierte Beschreibung aller zugrunde liegenden Techniken sowie eine Erläuterung zur Anwendbarkeit auf das Stundenplanproblem in Kapitel 3 ab Seite 27.

### 2.2.4 Evolutionsstrategien

Diese Techniken versuchen durch starke Anlehnung an die Natur und das durch Charles Darwin geprägte Prinzip der natürlichen Auslese (siehe [Dar59]) eine möglichst gute Lösung zu finden. Eine Ausprägung dieser Verfahren sind *genetische Algorithmen*. Hier transformiert man die Lösung eines Problems in eine Gen-ähnliche Repräsentation. In diesem Falle könnte dies eine simple Liste der Stundenplaneinträge sein. Der Algorithmus verwaltet dann eine Menge dieser Gen-Repräsentationen, die *Population*. In jedem Schritt der Suche lässt man nun schlechte Lösungen sterben (man nimmt sie aus der Population), während man gute miteinander kreuzt und sich so vermehren lässt (z.B. über Multipoint- oder Uniform-Crossover, siehe [FCMR99]). Dadurch nimmt zum einen die Anzahl der guten Lösungen in der Population zu und zum anderen verbessern sich die besten Lösungen mit hoher Wahrscheinlichkeit weiter. Verbesserungen dieses Ansatzes werden oft durch die Einführung eines Mutations-Schrittes erreicht, bei der sich eine vorhandene Lösung mit einer sehr kleinen Wahrscheinlichkeit minimal ändert. Um die Güte der Lösungen bestimmen zu können, benötigt man auch hier eine Heuristik.

Das *Survival of the fittest*-Prinzip lässt sich in diesem Zusammenhang aber nicht nur auf die genetische Repräsentation von Lösungen anwenden. Nimmt man andere runden-

basierte Suchverfahren zur Hilfe, kann man mehrere dieser Suchen (welche in diesem Fall die Population darstellen) auf einmal starten und dann im Rahmen der Evolutionssuche schrittweise synchron vorantreiben. Alle $n$ Schritte dupliziert man dann die besten Suchen und nimmt die schlechtesten aus der Population, wodurch man eine Qualitätssteigerung erreicht. Um dann am Ende nicht nur Duplikate der besten Lösungen in der Population zu haben, lässt man die Suchen bei einer Vervielfältigung entweder geringfügig mutieren oder wählt randomisierte Suchverfahren. So gesehen stellen die Evolutionsstrategien eine *Metasuche* dar. Weitere Betrachtungen zu Evolutionsstrategien finden sich in Kapitel 3.5, allgemeinere Überlegungen zu evolutionsbasierten Verfahren in [Wei07].

### 2.2.5 Graphentheoretische Verfahren

Das Generierungsproblem lässt sich durch verschiedene graphentheoretische Ansätze modellieren, von welchen hier zwei beispielhaft kurz aufgezeigt werden sollen.

#### Reduzierung auf das Färbungsproblem

Zur besseren Verständlichkeit wird hier nur das Teilproblem des **Classroom Assignments** (siehe 1.2.2) betrachtet, andere Teilprobleme lassen sich analog modellieren. Gegeben sei eine Zuordnung von Klassen und ihren Fächern zu den verschiedenen Terminen bzw. Zeitslots. Ein Stundenplaneintrag bzw. Meeting besteht in diesem Fall also aus einem Tripel $(Klasse, Fach, Zeitpunkt)$. Jedes solcher Tripel wird als Knoten eines Graphen modelliert, wobei zwischen Knoten, die zeitlich kollidieren, eine ungerichtete Kante erstellt wird. Formal lässt sich dies folgendermaßen beschreiben:
Sei $G = (V, E)$ der zu untersuchende Graph, wobei allen $v \in V$ ein entsprechendes Unterrichts-Tripel zugeordnet ist. Weiter sei $t(v)$ die Funktion, die zu einem $v \in V$ aus dessen zugeordnetem Tripel den Zeitpunkt extrahiert und zurückliefert. Dann gilt:
$E = \{(u, v) \quad | \quad u \in V \wedge v \in V \wedge t(u) = t(v)\}$.
Nun existiert eine gültige Raumzuordnung genau dann, wenn der oben beschriebene Graph in $k$ Farben eingefärbt werden kann, so dass keine zwei benachbarten Knoten derselben Farbe zugeordnet sind. Hierbei entspricht $k$ der Anzahl der Räume und jeder unterschiedlicher Farbwert repräsentiert einen Raum. Die Korrektheit dieser Vorgehensweise folgt direkt aus der Korrektheit der Färbung (die angenommen wird) und der einfachen Feststellung, dass durch die jeweils unterschiedliche Färbung benachbarter Knoten auch unterschiedliche Räume impliziert werden. Eine ausführliche Beschreibung der Reduzierung des Stundenplanproblems auf das Färbungsproblem findet sich in [NT74] oder [Red04], wo auch Ansätze aufgezeigt werden, mithilfe von optionalen, gewünschten Bedingungen einen möglichst optimalen Stundenplan zu erhalten.

### Modellierung als Flussproblem

Sieht man das Stundenplanproblem als Flussproblem an, so modelliert man die verschiedenen Teilschritte meist als miteinander verknüpfte Ebenen, die zwischen der Quelle und der Senke entsprechend durch Kanten verbunden sind. Wir beschränken uns hier wieder nur auf ein Teilproblem; aus der Vorgehensweise wird aber klar, dass man je nach Anwendung mehrere Ebenen hintereinander schalten kann. In diesem Fall soll für jede Klasse für die entsprechenden Fächer eine Lehrerzuordnung gefunden werden, die den Fähigkeiten der Lehrer gerecht wird und außerdem deren maximale Stundenanzahl berücksichtigt (das **Faculty Timetabling**-Problem, siehe Kap. 1.2.2). Die Abbildung 2.2 erläutert das Prinzip:

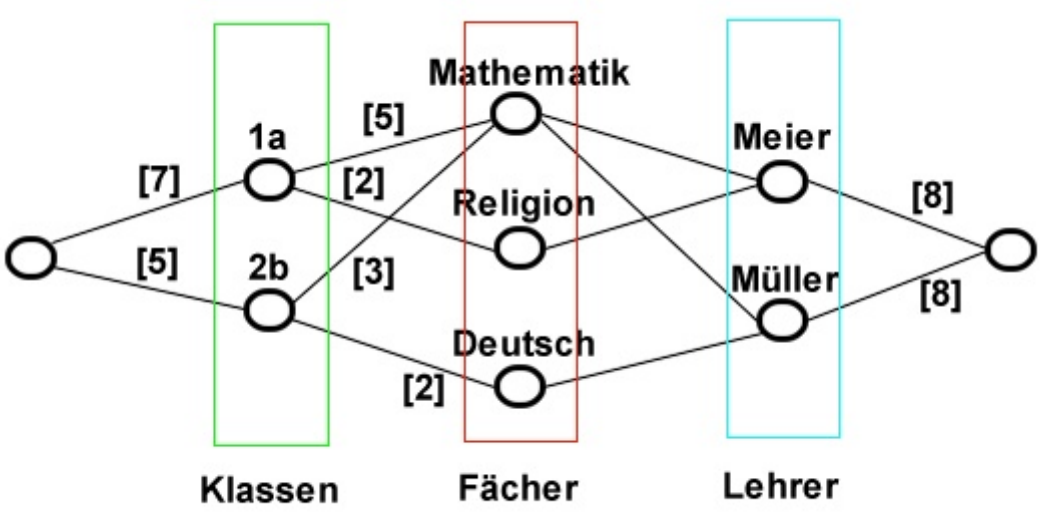


Abbildung 2.2: Das Stundenplanproblem als Flussproblem

Die erste Ebene (von links) stellt die vorhandenen Klassen als Knoten dar, zu welchen von der Quelle jeweils Kanten mit der maximalen Kapazität der abzuleistenden Stundenanzahl der jeweiligen Klasse zeigen. In der nächsten Ebene werden den Klassen wieder über kapazitätsbeschränkte Kanten deren Fächer zugewiesen. Danach reichen von diesen Fächerknoten Kanten zu den Lehrern genau so, dass von jedem Fach auch nur eine Kante auf einen Lehrer zeigt, wenn dieser dieses Fach tatsächlich unterrichten kann. Diese Kanten sind hier nicht in ihrer Kapazität beschränkt, da angenommen wird, dass die Verteilung der Lehrerstunden bis auf die Einhaltung der maximalen Anzahl keinen weiteren Restriktionen unterliegt. Zuletzt führen Kanten von den Lehrerknoten zur Senke mit der maximalen Kapazität der Arbeitsstunden für diesen Lehrer. Offensichtlich gibt es genau dann eine gültige Zuordnung der Lehrer zu den Fächern der einzelnen Klassen, wenn ein maximaler Fluss (d.h. der Größe der aufsummierten Klassenstunden) von der Quelle zur Senke möglich ist. [OdW83] und [DMV89] setzen sich tiefergehend mit dieser

Problematik auseinander.

### 2.2.6 Ganzzahlige Lineare Programmierung

Ganzzahlige Lineare Programmierung ist eine Technik, kombinatorische Probleme mithilfe von Gleichungen über numerischen Variablen zu lösen. Dabei werden für jede an die Lösung gestellte Bedingung ein oder mehrere Gleichungen formuliert; eine Variablenzuordnung, die diese Gleichungen erfüllt, stellt dann wiederum eine Lösung des Ursprungsproblems dar.

Übertragen auf das Stundenplanproblem bedeutet dies, dass insbesondere Gleichungen, die die abzuleistenden Stunden von Klassen und Lehrern, sowie solche zur Beschreibung von Konflikten erstellt werden müssen. Will man beispielsweise die Stunden der Klassen auf die verfügbaren Zeitpunkte und die zur Verfügung stehenden Lehrer verteilen, wäre folgendes Vorgehen denkbar. Sei $K$ die Menge der Klassen, $L$ die Menge der Lehrer und $Z$ die Menge der Zeitslots. Die Anzahl der für jede Klasse abzuleistenden Stunden sei mit $SK_k(k \in K)$ bezeichnet und die maximale Stundenzahl für Lehrer mit $SL_l(l \in L)$. Eine Variable $x_{k,l,z}$ ist 1 (mit $k \in K, l \in L, z \in Z$), wenn die Klasse $k$ bei Lehrer $l$ zum Zeitpunkt $z$ unterricht hat, andernfalls 0. Mit diesen Definitionen lässt sich das Problem mit folgenden Gleichungen beschreiben:

$$\sum_{k \in K} x_{k,l,z} \leq 1 \qquad (\forall l \in L, \forall z \in Z) \tag{2.1}$$

$$\sum_{l \in L} x_{k,l,z} \leq 1 \qquad (\forall k \in K, \forall z \in Z) \tag{2.2}$$

$$\sum_{l \in L} \sum_{z \in Z} x_{k,l,z} = SK_k \qquad (\forall k \in K) \tag{2.3}$$

$$\sum_{k \in k} \sum_{z \in Z} x_{k,l,z} \leq SL_l \qquad (\forall l \in L) \tag{2.4}$$

Die Gleichungen aus 2.1 besagen, dass jeder Lehrer zu jedem Zeitpunkt maximal in einer Klasse unterrichten darf, während 2.2 analog die Konfliktfreiheit für Klassen beschreibt. Gleichungen 2.3 und 2.4 treffen Aussagen über die gesamten Stundenanzahlen von Lehrern und Schülern. Die generierten Gleichungen können dann als Eingabe für einen generischen Algorithmus dienen, der solche Probleme lösen kann. Dafür gibt es auch wieder verschiedene Techniken und Methoden, Beispiele werden in [BE72] oder [TS74] beschrieben. Als Ausgabe erhält man dann in diesem Fall eine Variablenbelegung für die jeweiligen $x_{k,l,z}$, welche einen gültigen Stundenplan beschreibt. Ein Konzept zur Lösung des Stundenplanproblems mithilfe von Ganzzahliger Linearer Programmierung findet sich etwa in [Bea97].

## 2.3 Unser Lösungsweg

### 2.3.1 Entscheidungen

#### Wahl der Algorithmen

Da das Stundenplanproblem in der Theorie schon sehr gut und ausgiebig erforscht wurde, existieren für die Praxis auch die verschiedensten Ansätze zur Lösungsfindung. Ein exemplarischer Auszug befindet sich in Kapitel 2.2. Aus Gründen der Generizität, der Flexibilität und der großen Anzahl an Lösungsmöglichkeiten und -algorithmen, die sich mithilfe des Konzeptes der *Nachbarschaften* umsetzen lassen, haben wir uns im Rahmen dieses Projektes für diese Möglichkeit entschieden. Da es eine grundsätzliches Ziel dieser Arbeit ist, die Generierung und Optimierung der Stundenpläne so dynamisch und variabel wie möglich zu halten, liegt diese Entscheidung neben der Verwendung benutzerspezifizierbarer Constraints auf der Hand.

Zudem ist das Stundenplanproblem $\mathcal{NP}$-vollständig (siehe Kap. 1.2.3), was eine exakte Lösung (also das Finden des besten Stundenplans für die gegebenen Rahmenbedingungen) praktisch nicht möglich macht. Nachbarschaftsbasierte Verfahren versuchen die beste Lösung so gut wie möglich anzunähern, wobei die Laufzeit erheblich gesenkt wird. Diese sind auf die verschiedensten Probleme der Kombinatorik anwendbar (beispielsweise das Problem des Handlungsreisenden, siehe [ABCC98]), also auch wiederverwendbar.

Die Wahl der konkret implementierten Algorithmen fiel dabei auf die *Lokale Suche*, die Technik des *Simulated Annealing*, die *Tabu-Suche*, eine Variante der *Evolutionsstrategien* und einen *Greedy*-basierten Ansatz. Letzterer basiert auf dem Ansatz der *direkten Heuristiken* und soll einen Vergleich und Kontrast zu den nachbarschaftsbasierten Verfahren bilden, um nicht gänzlich von den Vor- und Nachteilen dieser Verfahren abhängig zu sein. Alle diese Techniken sind in der Theorie gut erforscht und konnten in der Praxis gute Ergebnisse liefern. Dabei hat jeder Algorithmus natürlich gewisse Vor- und Nachteile (siehe dazu Kapitel 3.5), weswegen es im Endeffekt dem Nutzer überlassen wurde, welches Verfahren benutzt werden soll.

#### Constraints

Die Entscheidung, zur Bewertung der Stundenpläne keine fest einprogrammierte Heuristik zu benutzen, sondern dem Benutzer die Möglichkeit zu geben, diese über selbst spezifizierbare *Constraints* zu steuern, ist eine Besonderheit dieses Systems und unterscheidet es von den meisten anderen.

Um die Güte eines generierten Stundenplanes zu bestimmen, benötigt man eine Funktion, die einem konkreten Plan in irgendeiner Weise eine Güte zuordnet. Bei den meisten kommerziellen Produkten wie auch in vielen wissenschaftlichen Projekten geschah dies durch feste Einprogrammierung der Constraints in einem Bewertungs-Modul (zum Beispiel durch Implementierung einer Methode, die die Anzahl der Freistunden eines Planes zählt und ein entsprechendes Feedback zurückgibt). Diese Vorgehensweise erweist sich jedoch als sehr unflexibel. So kann die spezielle Situation in einer Schule eine komplett andere Gewichtung der Bedingungen fordern oder aber komplett neue Constraints, die im Programm (noch) gar nicht vorhanden oder berücksichtigt worden sind. In einem solchen Fall würde eine Schule dann eine Speziallösung oder aber eine Änderung des bestehenden Systems benötigen. In beiden Fällen ist eine (Um-)Programmierung notwendig, welche neben dem erforderlichen Aufwand auch meist hohe Kosten mit sich bringt. Auch eine Änderung der Constraints etwa durch neue Gesetze im Bildungsbereich kann ein solches Programm im schlimmsten Fall unbrauchbar machen.

Aus diesem Grund soll mit dem vorgestellten System eine Lösung geschaffen werden, welche es ermöglicht, dynamisch und flexibel Bedingungen an die Generierung von Plänen zu stellen. Dies sollte auf der einen Seite ohne tiefergehende programmiertechnische Kenntnisse möglich sein, auf der anderen Seite jedoch immer noch eindeutig vom Computer interpretierbar bleiben. Damit wird erreicht, dass das System hochgradig generisch und universell einsetzbar ist. Auf der anderen Seite kann dieses durch die geschickte Formulierung dieser Constraints speziell auf die jeweilige Schule angepasst werden. Eine solche Spezialisierung konnte im Fall der betreuten Schule durch kein anderes getestetes Programm geleistet werden. Das resultierende System, das die Auswertung der Constraints vornimmt, wird in Kapitel 4 beschrieben.

### System-Bedingungen

Die Entscheidung, anstelle einer fest einprogrammierten Heuristik ein System zur dynamischen Beschreibung der Bedingungen zu verwenden, hat neben den beschriebenen Vorteilen einen ganz klaren Nachteil. Das Auswerten dieser Constraints macht sich ganz deutlich in der Laufzeit bemerkbar, wodurch verschiedene Optimierungen notwendig wurden (siehe auch Kapitel 5.2.1). Unabhängig davon wurde entschieden, dass bestimmte Bedingungen, die sowieso immer gelten müssen und kontextunabhängig sind, fest im Programm verankert werden und so nicht durch die Constraints abgeprüft werden müssen. Diese nennen wir im folgenden *System-Bedingungen*. In erster Instanz beschreiben diese die Gültigkeit der Stundenpläne und sorgen so dafür, dass ein Plan, der diese Bedingungen schon nicht erfüllt, auch gar nicht vom Algorithmus betrachtet wird. Beispiele dafür sind die Einhaltung der maximalen Stundenzahlen der Lehrer, die Einhaltung der Fähigkeiten der Lehrer, Verfügbarkeiten der Räume und natürlich die Garantierung der Konfliktfreiheit, also dass etwa keine Klassen oder Lehrer zur gleichen Zeit im gleichen Raum Unterricht haben. Dadurch wird der Lösungsraum für die Algorithmen erheblich

verkleinert, was die Laufzeit stark senkt.

### 2.3.2 Architektur

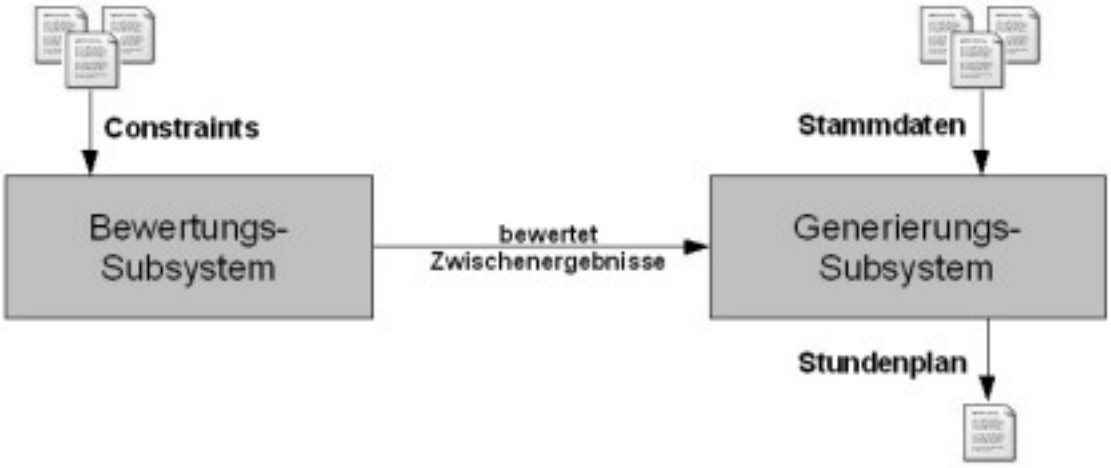


Abbildung 2.3: Ein grober Systemüberblick

Aus den beiden vorangegangenen Entscheidungen, die die grundsätzliche Vorgehensweise in diesem Projekt bedingen, lässt sich nun ein sehr grober, erster Überblick über das implementierte System anstellen. Wie in Abbildung 2.3 dargestellt, zerfällt das Kernsystem hauptsächlich in zwei große Komponenten, jeweils eine zur Bewertung eines Stundenplanes und eine zur Generierung und Suche. Das Subsystem zur Berechnung der Güte eines Planes liest dabei die zugrunde liegenden Bedingungen aus einer Datei ein und gibt auf Anfrage und Übergabe eines Stundenplanes ein Feedback zurück. Das Generierungs-Subsystem verwaltet auch wieder über eine Datei-Struktur die Stammdaten (Informationen über die Klassen, Lehrer, Fächer etc.) und kann aus diesen einen oder mehrere Stundenpläne generieren. Als Schnittstelle zwischen den beiden Komponenten fungiert die Heuristik, welche das Bewertungs-Modul für die Berechnung der Güte eines Planes nutzt.

Die Wahl bezüglich der grundsätzlichen Systemarchitektur fiel dabei auf die Programmiersprache Java. Grundsätzlich wäre auch eine Implementierung in einer beliebigen anderen Programmiersprache möglich gewesen, jedoch gibt es eine Reihe von Gründen, weshalb in diesem Fall Java einige Vorteile bietet:

1. Durch die Plattformunabhängigkeit des generierten Byte-Codes und dem Konzept der *virtuellen Maschinen* ist dieser auf einer Vielzahl von verschiedenen Betriebssystemen lauffähig.

2. Die einfache Gestaltung von grafischen Benutzeroberflächen erlaubt ein schnelles und professionelles Design der Anwendung, was vorallem die Benutzbarkeit stark fördert.
3. Für Standard-Aufgaben ist eine Vielzahl von freien Tools verfügbar, so dass persistente Daten einfach als XML-Dateien gespeichert und wieder ausgelesen werden können.
4. Für Java sind mehrere Parser-Generatoren verfügbar, welche den Aufwand, einen Parser für die vorgesehene Sprache zur Formulierung der Constraints zu schreiben, so gering und wenig fehleranfällig wie möglich halten.

In den beiden folgenden Kapiteln in 3 wird zunächst unsere Vorgehensweise bei der Suche nach dem möglichst optimalen Stundenplan skizziert. Wie die Anforderungen an einen guten Plan mithilfe der Constraints beschrieben werden können, wird ausführlich in Kapitel 4 beleuchtet.

# Kapitel 3

# Das Verfahren zur Generierung der Stundenpläne

Zur Umsetzung von nachbarschaftsbasierten Verfahren benötigt man zunächst gewisse Grundbausteine, welcher sich dann jeder konkrete Algorithmus bedient und welche in Bezug auf das jeweilige Problem definiert werden müssen. Diese lauten wie folgt:

- Zuerst muss eine **Nachbarschaft** über mögliche Lösungen des Problems festgelegt werden. Diese besagt, welche Lösungen ähnlich zueinander sind.
- Mithilfe dieser abstrakten Definition der Nachbarschaft kann man **Iteratoren** implementieren, welche die komplette Nachbarschaft zu einer gegebenen Lösung durchwandern und es so möglich machen, sich im Lösungsraum zu bewegen.
- Um während der Suche überhaupt erst entscheiden zu können, welche Lösung gut oder besser als eine andere ist, wird eine **Heuristik** benötigt, welche einen gegebenen Plan bewertet.
- Ein Durchsuchen des Lösungsraumes benötigt außerdem noch einen Startpunkt bzw. eine **Startlösung**, von der aus die Suche beginnt.

Sind diese Bausteine vorhanden, können ausgehend von der Startlösung weitere Lösungen durchlaufen und nach der besten oder zumindest einer besseren Lösung gesucht werden. Das Zusammenspiel dieser Komponenten zeigt Abbildung 3.1. Die verschiedenen Verfahren unterscheiden sich dann gerade in der Art, wie nach dieser besten Lösung gesucht wird.

Eine Definition der Nachbarschaft in Bezug auf dieses Projekt folgt direkt in Kapitel 3.1; deren praktische Implementierung mittels Iteratoren wird in Kap. 3.2 beschrieben. Die Heuristik wird in diesem Kapitel zunächst als *Black Box* betrachtet und dementsprechend verwendet. Sie wird in Kapitel 3.3 kurz beleuchtet, ehe sich dann Kapitel 4 mit dem konkreten Auswertungsmechanismus befasst. Auf den Generator von Startlösungen wird in Kap. 3.4 eingegangen. Aufbauend auf diesen grundlegenden Einheiten werden

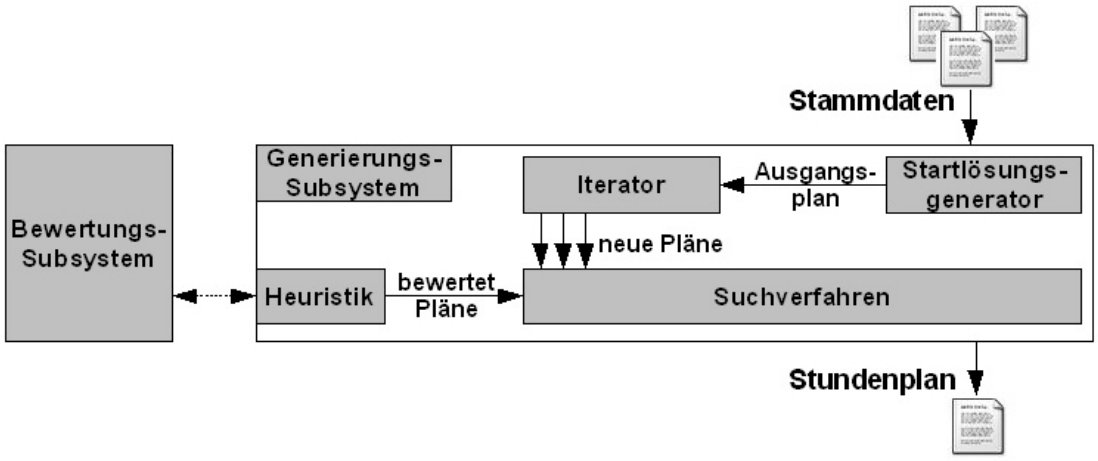


Abbildung 3.1: Komponenten eines nachbarschaftsbasierten Verfahrens

schließlich die Suchalgorithmen in Kapitel 3.5 besprochen. Für alle diese Komponenten wurden in diesem Rahmen Interfaces angelegt, um eine variable Benutzung sowie den einfachen Austausch dieser zu ermöglichen. Prinzipiell können also alle Suchverfahren auch für andere Probleme eingesetzt werden (etwa das des Handlungsreisenden, siehe [ABCC98]), indem entsprechend die Heuristik etc. angepasst wird.

## 3.1 Nachbarschaften

Wie schon erwähnt, drückt eine Nachbarschaft von zwei Lösungen in einem Suchverfahren aus, dass diese Lösungen in gewisser Weise ähnlich bzw. verwandt zueinander sind. Anders formuliert bedeutet dies, dass sich zwei benachbarte Lösungen nur minimal voneinander unterscheiden dürfen (möglichst auch in ihrer Güte). Da dieses Faktum von vielen Suchverfahren ausgenutzt wird, ist auch die Qualität des Suchverfahrens stark von der verwendeten Nachbarschaftsdefinition abhängig. In den meisten bisherigen Lösungsansätzen (z.B. in [SOS05]) bezieht sich diese Nachbarschaftsdefinition nur auf ein Teilproblem des Stundenplanproblems; da hier mehrere Teilprobleme auf einmal gelöst werden müssen, werden hier mehrere verschiedene Arten von Nachbarschaften separat definiert. Die gesamte Nachbarschaft einer Lösung definiert sich dann durch die Vereinigung aller so gebildeten Teilnachbarschaften. Diese werden in den nächsten Abschnitten jeweils zuerst formal festgelegt und danach informell und anhand eines Beispiels erläutert. Für Ersteres wird das folgende Modell eines Stundenplanes benutzt, welches sich an der Implementierung dieses Projektes orientiert.

**Definition 1 (Stundenplan und Stundenplaneintrag)** *Sei $C$ die Menge der Klassen, $T$ die Menge der Lehrer, $S$ die Menge der Fächer, $R$ die Menge der zur Verfügung stehenden Räume und $M$ die Menge der verfügbaren Zeitslots (als geordnetes Paar der Form $(d, h)$, wobei $d$ den Wochentag und $h$ die Unterrichtsstunde darstellt). Dann ist ein **Stundenplaneintrag** $TTE$ ein Tupel der Form $(c, t, s, r, m)$ mit $c \in C, t \in T, s \in S, r \in R$ und $m \in M$. Ein **Stundenplan** $TT$ ist eine Menge von Stundenplaneinträgen. Dieser Plan muss noch nicht zwangsweise gültig sein.*

### 3.1.1 Nachbarschaft in Bezug auf Lehrer

Die Definition der Nachbarschaft zweier Stundenpläne in Bezug auf Lehrer basiert auf der Idee, dass man (a) durch neues Zuordnen eines Lehrers zu einem Stundenplaneintrag oder (b) durch Tauschen der Lehrer zweier Stundeplaneinträge, die zur gleichen Zeit stattfinden, von einem Nachbarplan zum anderen gelangt. Dies motiviert die folgende Definition.

**Definition 2 (Nachbarschaft in Bezug auf Lehrer)** *Seien $TT_1$ und $TT_2$ zwei Stundenpläne. Diese gelten in Bezug auf Lehrer genau dann als benachbart ($(TT_1, TT_2) \in N_T$), wenn es ein $TTE_1 = (c_1, t_1, s_1, r_1, m_1) \in TT_1$ (bzw. $TTE'_1 = (c'_1, t'_1, s'_1, r'_1, m'_1) \in TT_1$) und ein $TTE_2 = (c_2, t_2, s_2, r_2, m_2) \in TT_2$ (bzw. $TTE'_2 = (c'_2, t'_2, s'_2, r'_2, m'_2) \in TT_2$) gibt, so dass entweder*

$$c_1 = c_2 \wedge t_1 \neq t_2 \wedge s_1 = s_2 \wedge r_1 = r_2 \wedge m_1 = m_2 \wedge \\ TT_1 \setminus \{TTE_1\} = TT_2 \setminus \{(TTE_2)\} \tag{3.1}$$

*oder*

$$\begin{aligned} &c_1 = c_2 \wedge t_1 = t'_2 \wedge s_1 = s_2 \wedge r_1 = r_2 \wedge m_1 = m_2 \wedge \\ &c'_1 = c'_2 \wedge t'_1 = t_2 \wedge s'_1 = s'_2 \wedge r'_1 = r'_2 \wedge m'_1 = m'_2 \wedge \\ &\quad TT_1 \setminus \{TTE_1, TTE'_1\} = TT_2 \setminus \{TTE_2, TTE'_2\} \end{aligned} \tag{3.2}$$

*gilt.*

Nach Bedingung 3.1 sind also zwei Stundenpläne in Bezug auf Lehrer dann benachbart wenn es daraus je einen Stundenplaneintrag gibt, welcher sich vom anderen allein durch die zweite Stelle (also den Lehrer-Eintrag) unterscheidet, ansonsten aber gleich ist. Abbildung 3.2 verdeutlicht das Prinzip anhand eines Auszuges eines imaginären Stundenplanes. Die grau dargestellten Stundenplaneinträge sind in beiden (Teil-)Plänen identisch und müssen also nicht weiter betrachtet werden. Klasse, Fach, Raum und Zeitslot der beiden schwarz gedruckten Einträge sind ebenfalls gleich, nur der Lehrer eines Plans wurde im Gegensatz zum anderen geändert. Die beiden Pläne erfüllen also alle Kriterien aus 3.1, sie sind somit benachbart. Praktisch gewinnt man einen Plan aus dem anderen, indem man in einem festgelegten Eintrag den Lehrer tauscht.

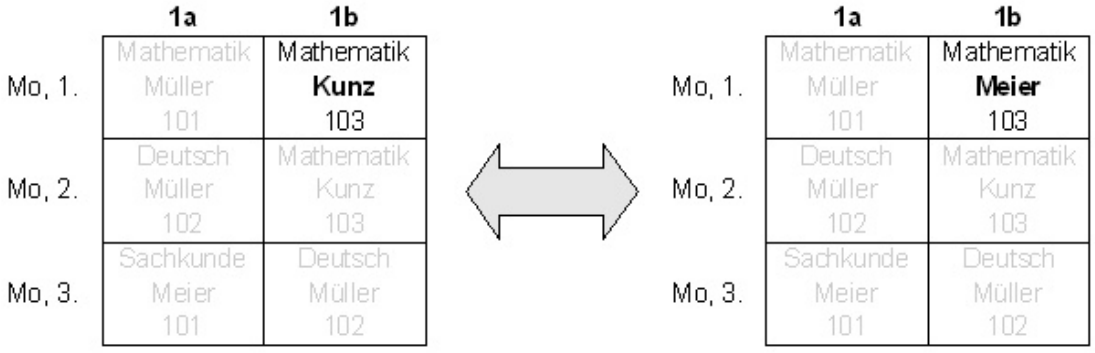


Abbildung 3.2: Nachbarschaft durch neuen Lehrer

Um Gleichung 3.2 gerecht zu werden und somit ebenfalls in Bezug auf Lehrer benachbart zu sein, müssen zwei Stundenpläne jeweils zwei Einträge aufweisen, deren Lehrer kreuzweise getauscht wurden. Der Rest der Pläne muss identisch sein. Abbildung 3.3 zeigt, wie ein Plan aus dem anderen gewonnen werden kann, indem in zwei beliebigen (hier wieder schwarz eingefärbten), jedoch zeitgleichen Einträgen einfach die Lehrer getauscht werden.

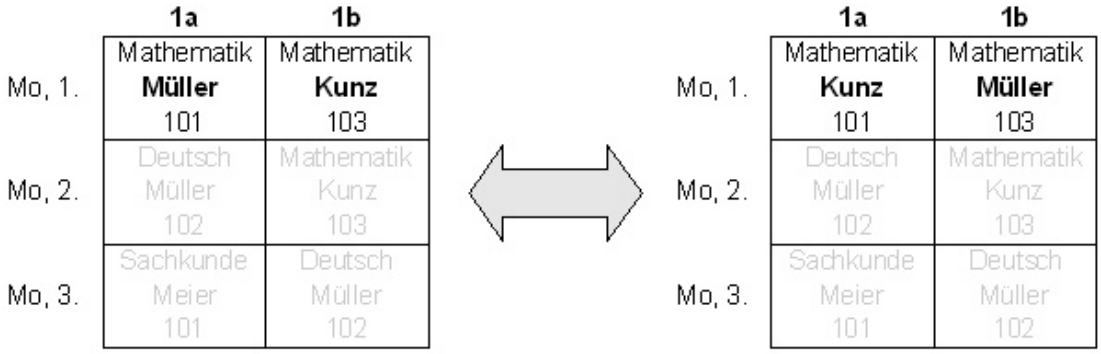


Abbildung 3.3: Nachbarschaft durch getauschten Lehrer

### 3.1.2 Nachbarschaft in Bezug auf Räume

Analog zur Nachbarschaft in Bezug auf Lehrer lässt sich diese auch in Bezug auf Räume definieren.

**Definition 3 (Nachbarschaft in Bezug auf Räume)** *Seien $TT_1$ und $TT_2$ zwei Stundenpläne. Diese gelten in Bezug auf Räume genau dann als benachbart ($(TT_1, TT_2) \in N_R$), wenn es ein $TTE_1 = (c_1, t_1, s_1, r_1, m_1) \in TT_1$ (bzw. $TTE'_1 = (c'_1, t'_1, s'_1, r'_1, m'_1) \in TT_1$) und ein $TTE_2 = (c_2, t_2, s_2, r_2, m_2) \in TT_2$ (bzw. $TTE'_2 = (c'_2, t'_2, s'_2, r'_2, m'_2) \in TT_2$) gibt, so dass entweder*

$$\begin{aligned} c_1 = c_2 \wedge t_1 = t_2 \wedge s_1 = s_2 \wedge r_1 \neq r_2 \wedge m_1 = m_2 \wedge \\ TT_1 \setminus \{TTE_1\} = TT_2 \setminus \{(TTE_2)\} \end{aligned} \tag{3.3}$$

*oder*

$$\begin{aligned} c_1 = c_2 \wedge t_1 = t_2 \wedge s_1 = s_2 \wedge r_1 = r'_2 \wedge m_1 = m_2 \wedge \\ c'_1 = c'_2 \wedge t'_1 = t'_2 \wedge s'_1 = s'_2 \wedge r'_1 = r_2 \wedge m'_1 = m'_2 \wedge \\ TT_1 \setminus \{TTE_1, TTE'_1\} = TT_2 \setminus \{TTE_2, TTE'_2\} \end{aligned} \tag{3.4}$$

*gilt.*

Da die Bedingung 3.3 und 3.4 jeweils den einfachen bzw. kreuzweisen Austausch der Räume anstatt der Lehrer beschreiben, wird hier auf eine weitere Erläuterung verzichtet. Die Abbildungen 3.4 und 3.5 stellen die Nachbarschaft in Bezug auf Räume grafisch dar.

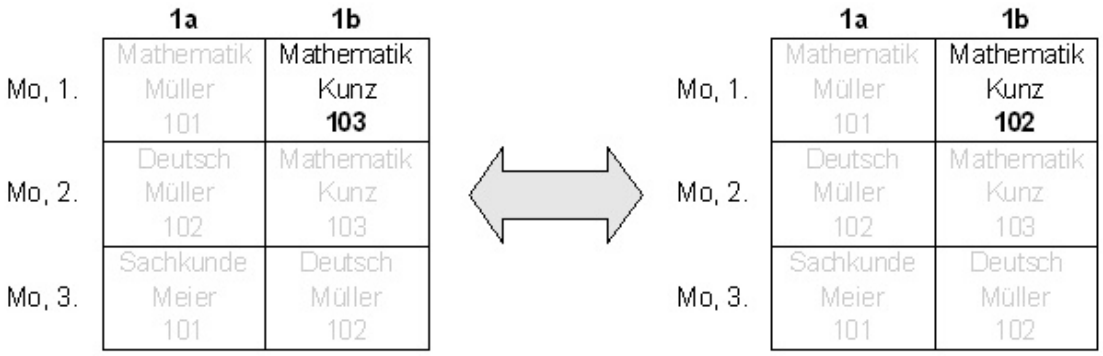


Abbildung 3.4: Nachbarschaft durch neuen Raum

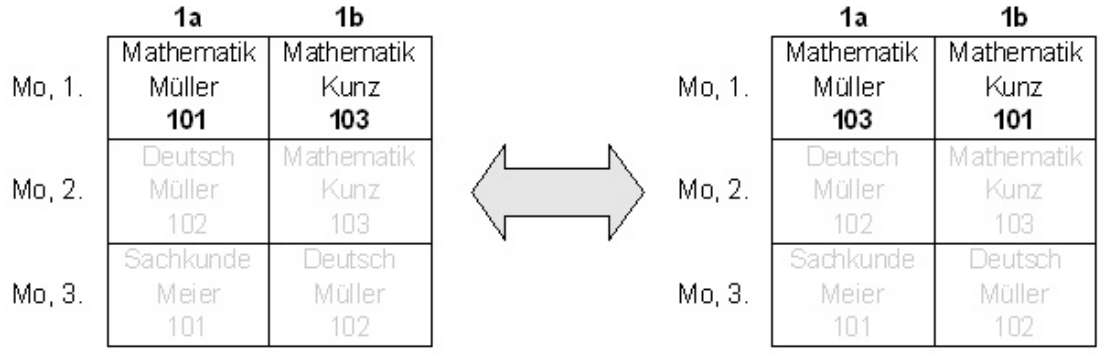


Abbildung 3.5: Nachbarschaft durch getauschten Raum

### 3.1.3 Nachbarschaft in Bezug auf Zeitslots

**Definition 4 (Nachbarschaft in Bezug auf Zeitslots)** *Seien $TT_1$ und $TT_2$ zwei Stundenpläne. Diese gelten in Bezug auf Zeitslots genau dann als benachbart ($(TT_1, TT_2) \in N_M$), wenn es ein $TTE_1 = (c_1, t_1, s_1, r_1, m_1) \in TT_1$ (bzw. $TTE'_1 = (c'_1, t'_1, s'_1, r'_1, m'_1) \in TT_1$) und ein $TTE_2 = (c_2, t_2, s_2, r_2, m_2) \in TT_2$ (bzw. $TTE'_2 = (c'_2, t'_2, s'_2, r'_2, m'_2) \in TT_2$) gibt, so dass entweder*

$$c_1 = c_2 \wedge t_1 = t_2 \wedge s_1 = s_2 \wedge r_1 = r_2 \wedge m_1 \neq m_2 \wedge TT_1 \setminus \{TTE_1\} = TT_2 \setminus \{(TTE_2)\} \tag{3.5}$$

*oder*

$$c_1 = c_2 \wedge t_1 = t_2 \wedge s_1 = s_2 \wedge r_1 = r_2 \wedge m_1 = m'_2 \wedge c'_1 = c'_2 \wedge t'_1 = t'_2 \wedge s'_1 = s'_2 \wedge r'_1 = r'_2 \wedge m'_1 = m_2 \wedge TT_1 \setminus \{TTE_1, TTE'_1\} = TT_2 \setminus \{TTE_2, TTE'_2\} \tag{3.6}$$

*gilt.*

Analog zu den ersten beiden Arten der Nachbarschaft definiert sich diese durch neu zugewiesene (Bedingung 3.5) bzw. getauschte (Bedingung 3.6) Zeitslots eines bzw. zweier Stundenplaneinträge. Abbildungen 3.6 bzw. 3.7 erläutern das Prinzip.

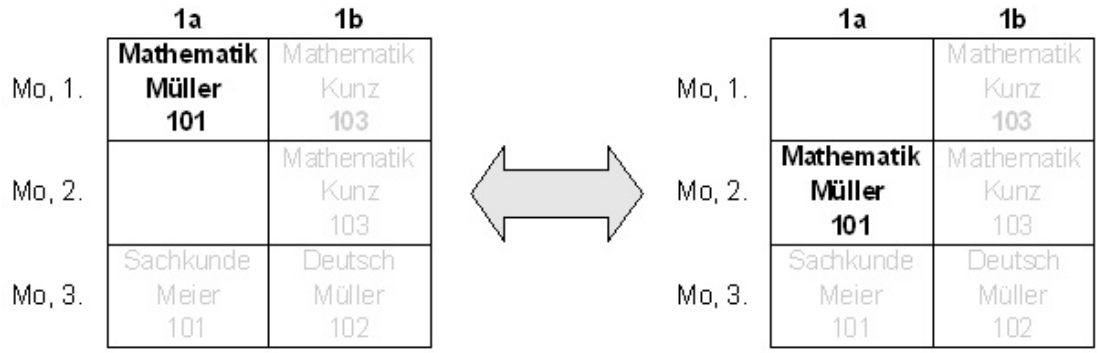


Abbildung 3.6: Nachbarschaft durch neuen Zeitslot

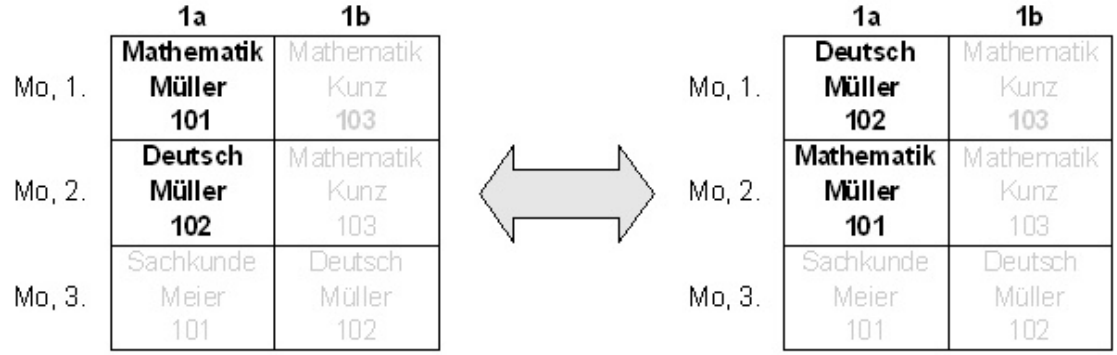


Abbildung 3.7: Nachbarschaft durch getauschten Zeitslot

Nun haben wir alle Nachbarschaften einzeln definiert und können diese einzelnen Teilnachbarschaften für die Definition einer generellen Nachbarschaftsrelation nutzen.

**Definition 5 (Nachbarschaft von Stundenplänen)** *Für die Nachbarschaft zweier Stundenpläne gilt:*

$$\begin{aligned}(TT_1, TT_2) \in N \quad \leftrightarrow \quad & (TT_1, TT_2) \in N_T \vee \\ & (TT_1, TT_2) \in N_R \vee \\ & (TT_1, TT_2) \in N_M \end{aligned} \tag{3.7}$$

*Die gesamte Nachbarschaft eines Planes $TT$ kann man dann mit $N(TT)$ bezeichnen, wobei gilt:*

$$N(TT) = \{TT' \quad | \quad (TT, TT') \in N\} \tag{3.8}$$

Bei dieser Definition mag zunächst auffallen, dass durch diese Nachbarschaften nur 3 der 5 Einträge eines Stundenplanes getauscht werden können; der Fach- und Gruppeneintrag der jeweiligen Stundenplaneinträge bleibt fest. Im Falle der Gruppe ist dies jedoch nicht verwunderlich, da die Gruppen bzw. Klassen in diesem Fall feste Einheiten sind, nach denen die Stundenplaneinträge referenziert werden. Ein reiner Tausch des Faches eines Stundenplaneintrags ist ebenfalls nicht nötig, da man dies durch Tausch des Zeitslots, Tausch des Lehrers und Tausch des Raumes erreichen kann (das Einfügen eines neuen Faches sollte sowieso nicht möglich sein, da dies die Stundenverteilung der einzelnen Fächer in einer Klasse verändern würde). Mithilfe dieser Nachbarschatfsdefinitionen kann man nun Iteratoren implementieren, die über die Nachbarschaft $N(TT)$ eines Stundenplanes $TT$ iterieren.

## 3.2 Die Iteratoren

Um verschiedene mögliche Stundenpläne zu gegebenen Rahmenbedingungen durchlaufen und bewerten zu können, benötigen die später besprochenen Suchverfahren *Iteratoren*, die genau dafür zuständig sind. Diese implementieren effektiv die in Kapitel 3.1 definierten Nachbarschaften der Pläne, indem sie zu einem gegebenen Stundenplan wiederholt bestimmte Lehrer, Räume oder Zeitpunkte ändern, so dass Pläne entstehen, die alle mit dem ursprünglichen Stundenplan benachbart sind (gemäß den gegebenen Definitionen). Da diese Nachbarschaften in verschieden Arten und Teilaspekte unterteilt wurden, liegt es nahe, auch die Iteratoren in der gleichen Struktur zu untergliedern.

An alle diese Iteratoren wird die Forderung gestellt, gültige Lösungen zurückzuliefern. Wie schon in Kapitel 2.3.1 beschrieben, werden aus Performancegründen nicht alle Bedingungen, die an einen Stundenplan gestellt werden, als Constraints formuliert. Die Bedingungen, die nicht variabel und auf jedes Stundenplanproblem zutreffen, die *System-Bedingungen*, werden von allen implementierten Iteratoren von der übergebenen Ausgangslösung (siehe auch Kap. 3.4) als erfüllt erwartet. Die Iteratoren müssen nun dafür sorgen, dass deren zurückgegebenen Lösungen wieder diese *System-Bedingungen* erfüllen, also insbesondere keine doppelten Zuordnungen aufweisen. Der Vorteil davon ist, dass sich der Lösungsraum stark verkleinert und jeder von einem Suchverfahren betrachtete Stundenplan auch eine potentielle Gesamt-Lösung darstellt. Zusätzlich wird von allen Iteratoren verlangt, dass sie die jeweils vollständige Nachbarschaft durchlaufen, welche sich aus Kapitel 3.1 ergibt.

### 3.2.1 Iteratorstruktur

Grundsätzlich lässt sich aus der gegebenen Nachbarschaftsdefinition die Unterteilung der Iteratoren in diejenigen vornehmen, die für Räume, Lehrer und Zeitslots verantwortlich sind. Eine weitere Unterteilung kann bei den Iteratoren, die für die Änderung der Räume und Lehrer zuständig sind, geschehen. Diese werden wiederum in einen Iterator, der einem Stundenplaneintrag einen neuen (zu diesem Zeitpunkt freien) Raum bzw. Lehrer zuordnet, und einen, der die Räume bzw. Lehrer zweier Einträge tauscht, untergliedert. Um die verschiedenen Iteratoren zu einem Gesamt-Iterator zu vereinen, ist es dann von Nöten, *Meta-Iteratoren* zu erstellen, die diejenigen auf einer tieferen Ebene benutzen. Abbildung 3.8 zeigt die Struktur der Iteratoren in diesem Projekt. Zu beachten ist hierbei, dass der Iterator, der die Vertauschung in Zeitslots vornimmt, nicht weiter unterteilt wird, da die Operation der Vertauschung und der Neuzuordnung in diesem Fall identisch ist. Ein jeder Iterator implementiert dabei das selbe Interface, um eine universelle und variable Benutzung der Iteratoren durch eine darüberliegende Instanz zu gewährleisten.

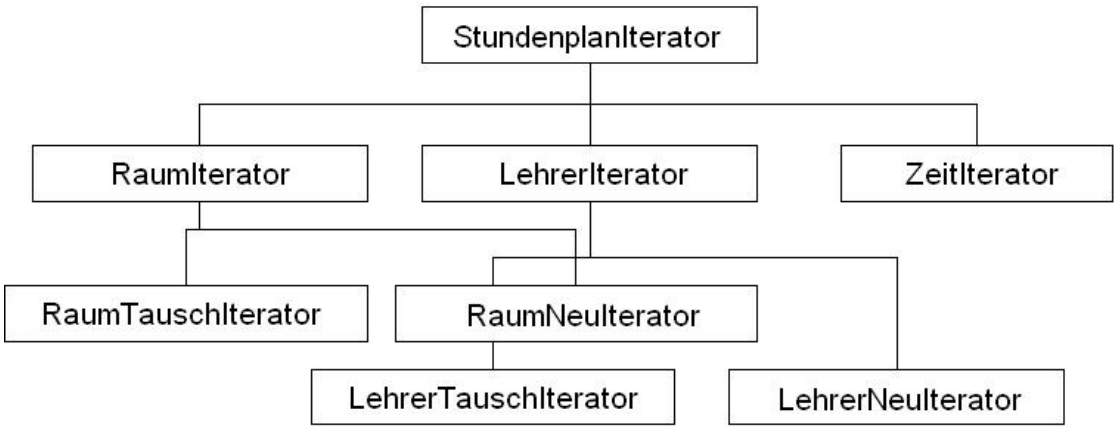


Abbildung 3.8: Die Iteratorstruktur des Projektes

Die Implementierung der Meta-Iteratoren gestaltet sich hierbei denkbar einfach. Ein solcher Iterator kann noch einen weiteren Stundenplan aus der Nachbarschaft zur Verfügung stellen, wenn dies auch für einen der darunter liegenden Iteratoren gilt. Wird ein Plan aus der Nachbarschaft angefordert, so kann einfach ein Plan eines unterhalb liegenden Iterators zurückgeliefert werden.

Die Iteratoren auf den unteren Stufen sind prinzipiell so implementiert, dass bei deren Initiierung eine Liste mit Kombinationen angelegt wird, die die zu tauschenden Einheiten beinhaltet. Im Falle des für die Neuzuordnung von Räumen zuständigen Iterators besteht eine solche Kombination zum Beispiel aus dem Stundenplaneintrag, dem ein neuer Raum zugeordnet werden soll, dem Zeitslot (bestehend aus Tag, Stunde), an welchem dieser Eintrag liegt, und dem neuen Raum, der dort eingesetzt werden soll. Diese Liste enthält nur die Kombinationen, deren Anwendung einen Plan zurückliefert, der nach den *System-Bedingungen* gültig ist, was jeweils beim Erstellen der Liste überprüft wird. Soll dann ein Plan generiert werden, wird der erste Eintrag ausgewählt, der Tausch durchgeführt und dieser aus der Liste gelöscht. Alternativ hätte man die Iteratoren auch so implementieren können, dass erst *on demand* beim Anfordern eines neuen Planes eine entsprechende Kombination erzeugt wird; jedoch wurde der beschriebene Weg bewusst gewählt, um auch andere Reihenfolgen in der Auswahl des nächsten Planes innerhalb einer Nachbarschaft durch einfaches Überschreiben der Klasse zu ermöglichen.

### 3.2.2 Zufällige und Intelligente Iteratoren

Die Liste der Tausch-Kombinationen wird bei jedem Iterator jeweils nach dem gleichen System und insbesondere in der gleichen Reihenfolge initiiert (die Zeitslots werden nach Ihrer zeitlichen Reihenfolge durchlaufen). Das angesprochene sequentielle Abarbeiten der Liste besipielweise bei einer *Lokalen Suche* (siehe Kap. 3.5.1) führt nun dazu, dass zu Anfang der Iteration durch eine Nachbarschaft auch immer die ersten Einträge der Liste bestrachtet werden. Nach einer gewissen Zeit ist es jedoch sehr wahrscheinlich,

dass die Pläne, die durch das Anwenden dieser Einträge am Anfang der Liste entstehen, schon eine hohe Güte in diesem Anfangs-Bereich aufweisen und dort nicht mehr verbessert werden können. Daher ist das Durchlaufen der resultierenden Stundenpläne in den meisten Fällen umsonst. Ein Durchlaufen der Liste in zufälliger Reihenfolge löst dieses Problem und sorgt in diesem Fall für eine nicht unerhebliche Performance-Steigerung. Daher wurden alle Iteratoren zusätzlich in einer randomisierten Variante implementiert, wobei sich eine Hierarchie analog zu Abbildung 3.8 ergibt. Auch die Meta-Iteratoren wählen in diesem Fall zufällig einen unterliegenden Iterator aus.

Die intelligenten Iteratoren gehen noch einen Schritt weiter: Wenn in einem Suchverfahren eine neue (temporär) beste Lösung gefunden wird, wird in der Regel ein neuer Iterator auf dieser Lösung erstellt (zum Beispiel bei der *Lokalen Suche*). Wenn sich die Meta-Iteratoren nun jeweils merken, welcher der letzte angewandte Unter-Iterator war, kann man damit feststellen, welcher Iterator im Verlauf einer Suche zu den meisten Verbesserungen geführt hat. Nun wird mit einer bestimmten Wahrscheinlichkeit dieser Iterator bevorzugt durch die Meta-Iteratoren ausgewählt, um den nächsten Schritt in der Nachbarschaft zu gehen. Mit einer weiteren sehr kleinen Wahrscheinlichkeit werden die Zähler, die speichern, welcher Iterator zu wievielen besseren Lösungen geführt hat, wieder zurückgesetzt, damit sich die Suche nicht auf einen Iterator beschränkt und sich der aktuellen Situation anpassen kann. Durch geschickte Programmierung konnten diese Iteratoren so implementiert werden, dass das Feedback über das Auftreten einer besseren Lösung nicht direkt von der Suche kommt, sondern der Iterator selbst für diese Information verantwortlich ist. So muss für die Benutzung dieser Iteratoren kein Suchverfahren verändert werden und eine transparente Benutzung ist möglich. Diese Iteratoren sind also nicht wirklich „intelligent“, sondern verwalten einfach eine Art *Historie* über vorangegangene Tauschvorgänge.

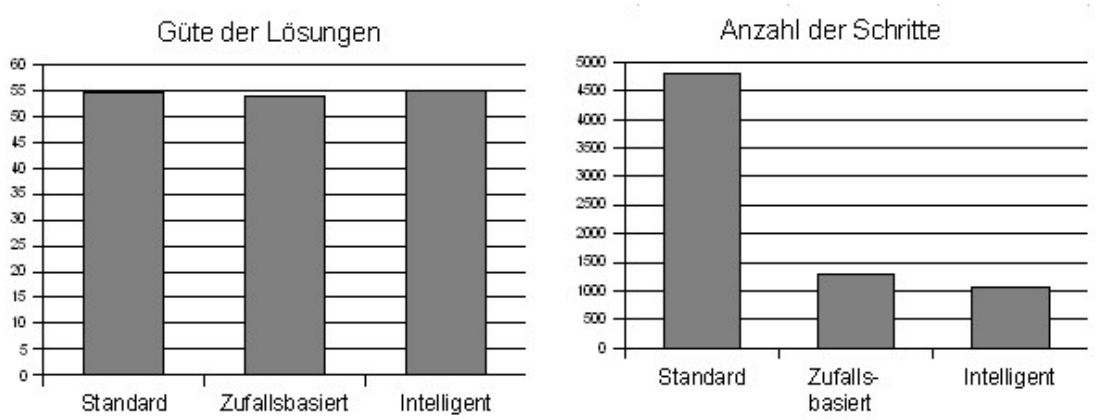


Abbildung 3.9: Vergleich der verschiedenen Iterator-Typen

Abbildung 3.9 zeigt die Unterschiede der drei Iterator-Typen in Bezug auf die Güte der zurückgelieferten Lösungen und der Anzahl der Suchschritte, die damit benötigt wurden.

Benutzt wurde ein kleines Datenset (3 Klassen, je 13 Stunden) und einfache Constraints, die Aussagen über Klassenlehrer und -räume sowie Freistunden machen, wobei 100 Lokale Suchen mit den drei Iterator-Arten gestartet wurden. Die Unterschiede in der Güte der Lösung sind dabei auf die randomisierte Komponente der zufallsbasierten und intelligenten Iteratoren zurückzuführen und vernachlässigbar gering. Größere Unterschiede treten dagegen bei den Laufzeiten auf, woran man sieht, dass sich der Einsatz der beiden zufälligen Arten durchaus lohnt. Die Verwendung der intelligenten Iteratoren führt dabei trotz optimierter Parameter (die Wahrscheinlichkeit zum Bevorzugen eines „guten“ Iterators und die zum Rücksetzen der entsprechenden Zähler) nur noch zu einer geringen Verringerung der Laufzeiten, was grundsätzlich die Qualität komplett zufallsbasierter Verfahren aufzeigt.

## 3.3 Die Heuristik

Zur Entscheidungsfindung in den jeweiligen Suchverfahren wird eine Instanz benötigt, die die Güte einer gültigen Lösung, also in diesem Fall eines Stundenplans, bestimmt. In der Regel wird diese Güte durch einen Zahlenwert repräsentiert, wobei wir hier ohne Beschränkung der Allgemeinheit annehmen, dass kleine Zahlenwerte eine hohe Güte beschreiben. Mit anderen Worten steht diese Zahl für eine Bestrafung (*Penalty*), die ein Plan durch die Heuristik erhält, wobei eine höhere Bestrafung natürlich einen schlechteren Plan kennzeichnet.

Wie schon in Kapitel 2.3.1 erwähnt, wird eine solche Bewertungsfunktion normalerweise fest im System einprogrammiert und entspricht meist der gewichteten Summe der Ergebnisse mehrerer einzelner Bewertungsroutinen. Ein Beispiel für eine solche Implementierung ist in [FCMR99] zu finden. Aus den schon genannten Gründen haben wir uns hier für einen dynamischeren Weg entschieden, welcher entsprechend aufwändiger zu implementieren ist. Kapitel 4 beschäftigt sich mit der Umsetzung der benutzten Heuristik. Das Wissen um die genaue Vorgehensweise ist für dieses weitere Kapitel nicht von Nöten, die Heuristik kann sozusagen als *Black Box* betrachtet werden, welche zu einem gegebenen Stundenplan eine Zahl zurückliefert, die dessen Güte ausdrückt. Oft wird diese Heuristik auch *Kostenfunktion* genannt, da man davon ausgeht, dass eine schlechte Lösung auch hohe Kosten verursacht, welche mit diesem Zahlenwert ausgedrückt werden sollen.

## 3.4 Generierung der Startlösung

Wie schon zu Anfang in Kapitel 3 erwähnt wurde, versuchen nachbarschaftsbasierte Suchverfahren das Optimierungsproblem zu lösen, also eine möglichst gute Lösung zu finden. Dies setzt eine Lösung des Generierungsproblems voraus, wobei in diesem Fall eine nach den *Systembedingungen* gültige Lösung als Startpunkt für die Suche zu finden ist (siehe auch Kap. 3.2). Üblicherweise benutzt man für diese Methoden ein zufallsbasiertes Verfahren für die Generierung einer Startlösung, um die Gefahr möglichst klein zu halten, in ein lokales Optimum zu laufen. Diese Technik wurde auch hier angewandt. Grundsätzlich ist bei der Implementierung des Startlösungsgenerators darauf zu achten, dass dieser zwar ein ausreichend zufälliges Ergebnis produziert, trotzdem jedoch während dieses Vorganges darauf geachtet wird, dass der resultierende Stundenplan nach den *Systembedingungen* gültig ist. Das grundsätzliche Vorgehen des Generators soll hier kurz skizziert werden.

Zunächst wird eine Liste der durch die Stammdaten definierten Klassen erstellt. Danach wird für jede Klasse die Liste der zu besuchenden Fächer mit den entsprechenden Stundenanzahlen bestimmt. Nun werden zuerst die schwer zu verteilenden Fächer zugeteilt, wie zum Beispiel diejenigen, die in Bandstunden organisiert sind, einen speziellen Raum benötigen oder in Blöcken verteilt werden sollen (etwa sollte Sport zum Beispiel immer zweistündig abgehalten werden). Dieses Vorgehen sorgt dafür, dass die Wahrscheinlichkeit, einen gültigen Plan zu finden, erheblich steigt, da für diese problematischen Fächer so noch mehr Möglichkeiten zur Platzierung zur Verfügung stehen. Danach werden dann die „normalen“ Fächer verteilt und abhängig von deren Stundenzahl eine Prozedur aufgerufen, welche für eine gegebene Gruppe dieses Fach einmalig plaziert.

Zu Anfang dieser Methode wird ein Zeitslot zufällig gewählt, an dem das Fach plaziert werden soll. Danach wird eine Liste der Lehrer und Räume erstellt, die zu diesem Zeitpunkt für dieses Fach verfügbar sind, wobei zum Beispiel bei den Lehrern auch deren maximale Stundenanzahl und Qualifikation berücksichtigt wird. Aus diesen Listen wird dann wieder zufällig ein Raum und ein Lehrer ausgewählt, aus den gesammelten Daten ein Stundenplaneintrag generiert und dieser in den Stundenplan geschrieben. Natürlich kann es vorkommen, dass zu einem Zeitpunkt entweder kein Lehrer oder kein Raum zur Verfügung steht, so dass eine der Listen leer sein kann. In diesem Fall startet die Prozedur von vorne und generiert einen neuen Zeitslot. Falls es in dieser Prozedur zu einer bestimmten Anzahl an Fehlversuchen kommt, bricht die Generierung durch die Auslösung einer *Exception* ab. Die prinzipielle Vorgehensweise wird in Abbildung 3.10 beschrieben.

Die beschriebene Methode beschränkt sich dabei auf das Platzieren der einfacheren Fächerkombinationen ohne weitere Nebenbedingungen; die vorher beschriebenen Methoden zum Legen der Bandstunden oder der Blockstunden gehen prinzipiell ähnlich vor, nur dass hier komplexere Gültigkeitsüberprüfungen anzuwenden sind. Da es von vorne-

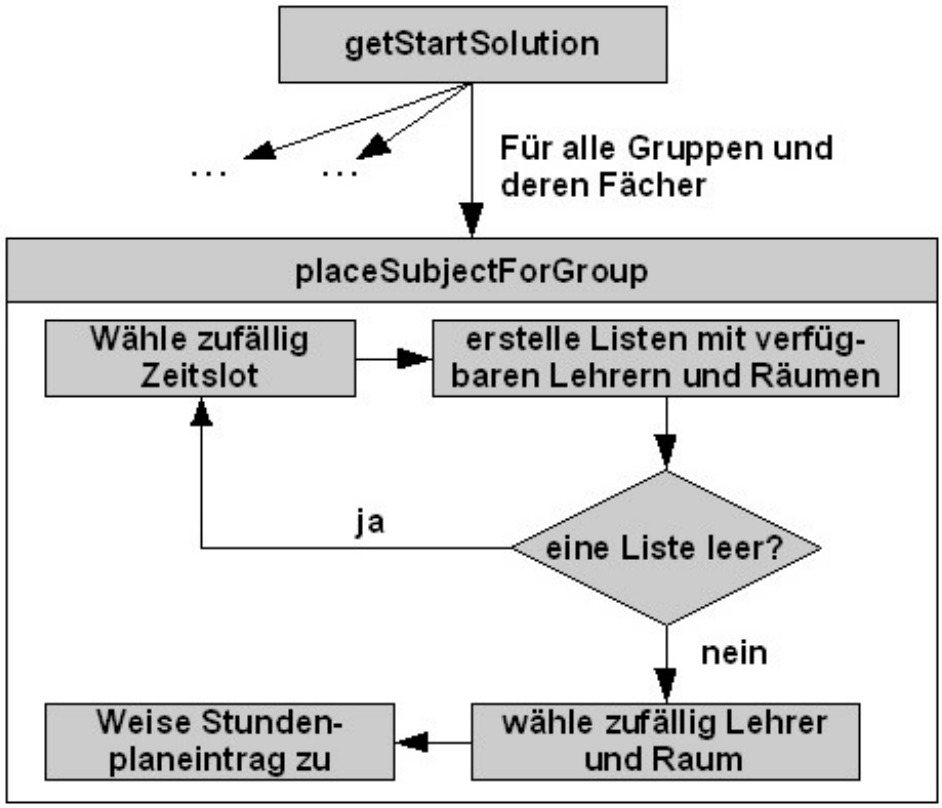


Abbildung 3.10: Zuweisung eines Faches beim Generieren der Startlösung

herein möglich ist, den Klassen spezielle Klassenlehrer und -räume zuzuordnen, ist es durch Angabe je einer Wahrscheinlichkeit auch möglich, den Startgenerator dahingehend zu steuern, dass er mit dieser Wahrscheinlichkeit einer Gruppe deren Klassenlehrer und -raum bevorzugt zuzuordnet. Mit einer Angabe von 90% für die Klassenlehrer-Wahrscheinlichkeit wird einer Klasse zum Beispiel nur im Schnitt in 10% ihrer Stunden ein anderer Lehrer zugewiesen, wobei hier natürlich auch wieder die Kompetenzen und Verfügbarkeiten der Lehrer berücksichtigt werden.

Die zufällige Generierung einer Startlösung schafft so gute Vorraussetzungen für eine nachbarschaftsbasierte Suche. Auf der anderen Seite kann man jedoch gerade durch die Zufälligkeit keine Aussagen über die Qualität der Startlösung machen. In einzelnen Fällen kann es sogar dazu kommen, dass die Generierung fehlschlägt. Für diesen Fall wurde ein weiterer Startgenerator implementiert, der den eigentlichen, oben beschriebenen Startgenerator mehrmals nacheinander ausführt und das beste Ergebnis zurückliefert. Dieser fungiert dann als *Wrapper* um den eigentlichen Generator, indem dieser das gleiche Interface implementiert und so in Kombination mit jedem anderen Generator und jedem Suchverfahren benutzt werden kann.

## 3.5 Die Suchverfahren

Mit den bis jetzt besprochenen Modulen haben wir nun alle Instrumente in der Hand (siehe auch Abb. 3.1), die wir benötigen, um ein konkretes Suchverfahren zu implementieren. Die *Lokale Suche* bildet den einfachsten und zugleich die Basis der im Folgenden dargestellten Algorithmen. Darauf bauen in veschiedener Weise das *Simulated-Annealing* und die *Tabusuche* auf. Als Meta-Suche über der Lokalen Suche kann man die nachfolgende Implementierung einer *evolutionsbasierten Suche* sehen. Einen komplett anderen Ansatz verfolgt das *Greedy-Verfahren*, das in diesem Unterkapitel zum Schluss kurz angesprochen wird.

### 3.5.1 Lokale Suche

Hat man eine Startlösung, eine Bewertungs-Instanz und einen Iterator gegeben, der die Nachbarschaft von Lösungen zurückliefern kann, so liegt es zunächst nahe, diese Nachbarschaft der Startlösung zu durchlaufen bis man auf eine bessere Lösung als die Startlösung trifft. Genau dieses Prinzip verfolgt die lokale Suche. Bei deren Start wird als momentan beste Lösung die Startlösung festgesetzt (eine andere kennt man zu diesem Zeitpunkt ja nicht). Nun benutzt man den Iterator, um andere Lösungen in deren Nachbarschaft zu finden. Hat man nun eine bessere Lösung gefunden, setzt man diese wiederum als die momentan beste fest. Nun beginnt man wieder in deren Nachbarschaft nach einer besseren Lösung zu suchen usw. Der Vorgang bricht ab, wenn die komplette Nachbarschaft einer Lösung durchsucht wurde und keine Verbesserung mehr erzielt werden konnte. Kleinere Variationen dieses Verfahrens werden in der Regel auch als lokale Suche bezeichnet, eine mögliche Änderung wäre zum Beispiel, dass zu einer Lösung jeweils die komplette Nachbarschaft durchlaufen wird und die beste Lösung aus der gesamten Nachbarschaft als Ausgangspunkt für die nächste Suche genommen wird. Beide Verfahren wurden in diesem Projekt implementiert.

Listing 3.1 stellt die Lokale Suche in Pseudo-Code dar. Die Funktion *cost* liefert hierbei die Güte einer Lösung in Form einer Zahl zurück. Diese entspricht in der Praxis dem Penalty, das durch die Verletzung von Constraints entsteht, kleine Werte sind also besser als größere.

```
1 Beginne mit einer gültigen Lösung s*
2 Initiiere Nachbarschaftsiterator it(s*)
3 Solange it(s*) noch weitere Lösungen hat
4    Sei s die nächste Lösung von it(s*)
5    Wenn cost(s) < cost(s*)
6       s* = s
7       Reinitiiere Iterator it(s*) auf neuem s*
8 Gib s* zurück
```

Listing 3.1: Lokale Suche

Der Schwachpunkt der lokalen Suche ist, dass diese nur lokale Optima im Lösungsraum erkennen und zurückliefern kann. Befindet sich die Suche in einem solchen lokalen Optimum, so gibt es in der unmittelbaren Nachbarschaft der momentan besten Lösung keine bessere, die Nachbarschaft wird erfolglos komplett durchlaufen und die Suche bricht ab. Nun kann ein Schritt zu einer vermeintlich schlechteren Lösung in deren Nachbarschaft jedoch wieder zu einer insgesamt besseren Lösung führen, welche in diesem Fall unentdeckt bleibt. Abbildung 3.11 zeigt diese Problematik. Der Lösungsraum ist in Wirklichkeit natürlich sehr viel höher dimensional, das Prinzip ist jedoch das gleiche.

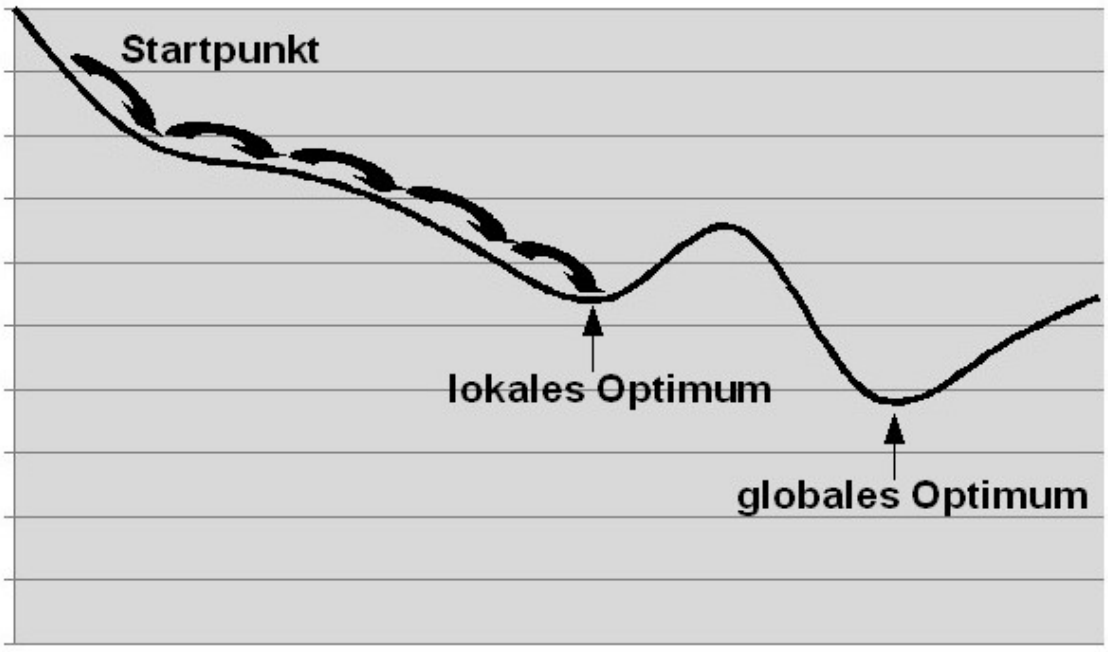


Abbildung 3.11: Lokales vs. globales Optimum

### 3.5.2 Simulated Annealing

Die Technik des Simulated Annealing versucht die Schwäche der lokalen Suche, in einem lokalen Minimum festzulaufen, durch eine zufallsbasierte Komponente auszugleichen. Konkret wird mit einer bestimmten Wahrscheinlichkeit auch dann eine Lösung als momentan beste angesehen, wenn diese eigentlich gar nicht besser ist. In Listing 3.2 sieht man den prinzipiellen Algorithmus in Pseudo-Code.

```
Beginne mit einer gültigen Lösung s*
Initiiere Nachbarschaftsiterator it(s*)
Solange it(s*) noch weitere Lösungen hat
   Sei s die nächste Lösung von it(s*)
   Wenn cost(s) < cost(s*)
      s* = s
      Reinitiiere Iterator it(s*) auf neuem s*
   Ansonsten
      Treffe zufällige Ja/Nein-Entscheidung
      Falls Ergebnis = Ja
         s* = s
         Reinitiiere Iterator it(s*) auf neuem s*
Gib s* zurück
```

Listing 3.2: Simulated Annealing

Mithilfe der Wahrscheinlichkeit, mit der die Ja/Nein-Entscheidung getroffen wird, kann man den Suchvorgang in gewisser Weise steuern. Während es am Ende einer Suche eher hinderlich ist, zufällig auch schlechtere Lösungen für die weitere Suche auszuwählen, ist dieses Verhalten am Anfang nicht problematisch und sogar gewollt. Deshalb lässt man diese Wahrscheinlichkeit von Schritt zu Schritt immer kleiner werden (daher auch der Name „simuliertes Abkühlen“), wobei man diese auch von der Güte der momentanen Lösung in Relation zur besten bis dahin gesehenen abhängig macht. Man kann beweisen, dass dieses Verfahren bei theoretisch unendlicher Laufzeit die optimale Lösung eines (Stundenplan-) Problems findet (Details in [Aze87] oder [Gid85]). Die verwendete Formel ist:

$$P = e^{\frac{cost(s*)-cost(s)}{T}} \qquad T = \frac{C}{\log step} \tag{3.9}$$

Hierbei ist $C$ eine Konstante, deren optimaler Wert empirisch bestimmt wurde, und *step* der aktuelle Suchschritt. Die Einbeziehung der Kostendifferenz zur besten bisher gesehenen Lösung stellt sicher, dass der Sprung zu einer schlechteren Lösung nicht zu besonders ungünstigen Lösungen führt. Mithilfe der genannten Veränderungen stellt das Verfahren des Simulated-Annealing eine Verbesserung der lokalen Suche dar, die jedoch mit Vorsicht zu genießen ist. Wählt man den Parameter $C$ für die Suche zu hoch, erhält man im Extremfall eine lokale Suche (die Wahrscheinlichkeit $P$ ist dann sehr klein oder gleich null); wählt man den Parameter zu klein, springt die Suche oft zu schlechten Lösungen, was eine Verlängerung der Laufzeit und eine schlechtere Qualität der Ergebnisse zur Folge haben kann. Ein längeres Test- und Optimierungsverfahren ist daher unabdinglich. Dieses Suchverfahren wurde zum Beispiel in [ADK99] implementiert, wo auch Analysen zu verschiedenen Abkühlungsmechanismen zu finden sind.

### 3.5.3 Tabu-Suche

Die Tabu-Suche ist ein weiterer Ansatz, durch den das Problem der Lokalen Suche, nur in einem lokalen Minimum zu enden, behoben werden soll. Falls hierbei ein solches während der Traversierung erreicht wird, wird dieses zunächst gespeichert. Die verschiedenen Strategien, die diesen Ansatz verfolgen, verwenden in diesem Fall eine Tabu-Liste, um sich von dem gefundenen lokalen Minimum zu entfernen und eventuell eine bessere Lösung zu finden.

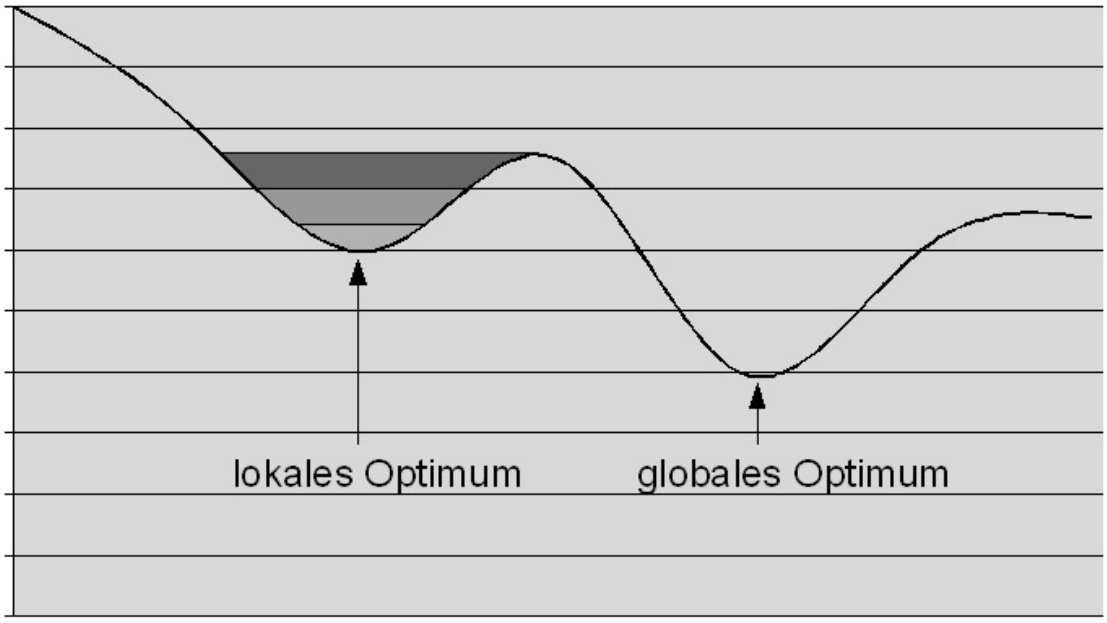


Abbildung 3.12: Verhalten der Tabusuche in einem lokalen Optimum

In dieser Tabuliste werden sogenannte Features gespeichert, aus welchen die Lösungen des Optimierungs-Problems bestehen. Im Falle der Stundenplanoptimierung wären die Features also die einzelnen Einträge eines Stundenplanes. Nach der Auffindung eines lokalen Minimums werden dann Features gesammlt und in die Tabuliste geschrieben, so dass eine zyklische Traversierung verhindert wird. Die Suche wird fortgesetzt, wobei analog zur Lokalen Suche wieder die beste Lösung einer Nachbarschaft ausgewählt wird. Der Unterschied ist, dass Lösungen, die ein in der Tabuliste gespeichertes Feature enthalten, hierbei nicht betrachtet und auf diese Weise vom Suchprozess ausgeschlossen werden. Auf diese Weise wird die Tabu-Suche von diesem lokalen Minimum weggeführt und kann ein potentiell besseres finden. Wie in Abbildung 3.12 illustriert ist, wird dieses lokale Optimum durch diesen Prozess „zugemauert“, und ist für die Tabu-Suche dann als solches nicht mehr erkennbar. Denkt man wiederum an die Stundenplanoptimierung, kann es also nicht dazu kommen, dass der selbe Eintrag über mehrere Suchschritte hinweg hin und her geschoben wird.

Als zu optimierenden Parameter ist die festgelegte Tabu-Dauer eine wichtige Einstellung der Tabu-Suche. Diese definiert die Anzahl der Suchschritte, die ein Feature in der Tabu-Liste gespeichert werden soll. Während des Suchprozesses wird weiterhin jedes gefundene lokale Minimum gespeichert, und nach Abschluss der Suche das beste als Ergebnis zurückgegeben. Als Abbruchkriterium kann eine festgelegte Anzahl der zu findenden Minima oder eine maximale Anzahl von Suchschritten dienen. Für die Generierung von Stundenplänen wurde die Tabu-Suche in verschiedenen anderen Systemen eingesetzt, siehe z.B. [SOS05].

### 3.5.4 Evolutionssuche

Als Evolutionsstrategien werden solche Suchverfahren bezeichnet, die sich des durch Darwin geprägten Prinzips des „Survival of the fittest“ (siehe [Dar59]) bedienen und somit in gewisser Weise natürliche Prozesse nachahmen. Wie in der Natur existiert hier eine Menge von verschiedenen Lösungen, die *Population*. Nun versucht man, einen Fortschritt bzw. eine Verbesserung dadurch zu erlangen, dass die schlechtesten Lösungen „sterben“ bzw. aus dieser Menge gelöscht werden, während sich die besten Lösungen in irgendeiner Weise reproduzieren. Je nach Ausprägung verwendet man zusätzlich oft noch das Prinzip der Mutation, bei dem sich zufällig gewisse Lösungen minimal ändern.

Wie schon in Kapitel 2.2 angesprochen, ist dies eine generische Vorgehensweise, die man auf verschiedene Arten konkret umsetzen kann. Die Variante, die sich am nähesten an der Natur orientiert, sind die *genetischen Algorithmen*, bei dem jede Lösung in geeigneter Weise als Menge von Chromosomen, den Genen, repräsentiert wird. In diesem Fall könnte das für einen Plan zum Beispiel eine immer gleich sortierte Liste der Stundenplaneinträge sein. Die Reproduktion erfolgt dann wie in der Natur durch das Kreuzen der Gene zweier Lösungen (mittels verschiedener Crossover-Verfahren), wobei daraus wieder zwei Kind-Lösungen entstehen. Dabei nimmt man an, dass das Kreuzen zweier guter Lösungen wie in der Natur mit hoher Wahrscheinlichkeit auch wieder eine gute Lösung oder gar bessere Lösung hervorbringt. Versuche, die in diese Richtung gehen, wurden im Rahmen dieser Arbeit unternommen, jedoch waren diese aus folgenden Gründen unergiebig.

Zum einen produziert die Kreuzung zweier Lösungen durch ein Crossover-Verfahren (in diesem Fall ein Uniform-Crossover, siehe z.B. [FCMR99]) ungültige Lösungen (da hierdurch wieder Systembedingungen verletzt werden), die dann erst wieder repariert werden müssen. Dieser Vorgang ist in diesem Fall so komplex, dass er auf der einen Seite eine vergleichsweise hohe Laufzeit benötigt und andererseits die Pläne so weit verändert, dass diese mit den Eltern-Plänen nicht mehr viel gemein haben. Damit ist eines der Grundprinzipien dieser Verfahren in Frage gestellt. Ein weiterer Grund für den Verzicht auf diese Verfahren war, dass man diese Gen-Repräsenatition ja auch wieder bewerten muss. Nun müsste das komplette Verfahren zur Bewertung der Pläne dahingehend erweitert werden, dass dies nicht nur die bisherige Repräsentation eines Stundenplanes

bewerten kann, sondern auch dessen genetische Darstellung. Aufgrund der Komplexität des Bewertungs-Subsystems stellt dies einen nicht lohnenswerten Ansatz dar.

Stattdessen wurde die Evolutionssuche als *Metasuche* über andere Suchverfahren implementiert. Die Population bilden in diesem Fall keine konkreten Lösungen sondern verschiedene Suchen, welche parallel schrittweise vorangetrieben werden. Im Prinzip kann man hierfür jedes andere Suchverfahren verwenden, aus Performance-Gründen wurde hier die Lokale Suche verwendet. Dabei wird jeder $n$-te Schritt als sogenannter *Key-Step* angesehen, in dem dann eine gewisse Anzahl von schlechten Lösungen aus der Population genommen wird und die gleiche Anzahl guter Lösungen repliziert wird. Durch die Verwendung randomisierter Iteratoren (siehe 3.2) ist sichergestellt, dass diese Replikate nicht alle die gleiche Richtung bei der Suche einschlagen. Da eine Suche in diesem Fall ohnehin schon einen relativ hohen zeitlichen Aufwand benötigt, muss hierbei die Population relativ klein gehalten werden, was die Effektivität dieses Verfahrens natürlich etwas einschränkt. Einen umfassenden Überblick über evolutionsbasierte Suchverfahren findet sich in [Wei07], eine Implementierung im Stundenplan-Kontext wird in [FCMR99] beschrieben.

### 3.5.5 Das Greedy-Verfahren

Generell wird durch ein *Greedy-Verfahren* ein Suchverfahren beschrieben, dass zu jedem Zeitpunkt die Entscheidung trifft, die gerade als beste erscheint. Dabei werden getroffene Entscheidungen nie rückgängig gemacht, auch wenn sie sich im Nachhinein als schlecht herausstellen. Im Kontrast zu den nachbarschaftsbasierten Suchverfahren, welche die Lösung des Generierungsproblems vorraussetzen, wurde zusätzlich ein Greedy-Verfahren implementiert, dass sowohl das Generierungsproblem als auch das Optimierungsproblem löst. Dieses zusätzliche Verfahren wurde entwickelt, um eventuellen, später auftretenden Nachteilen der nachbarschaftsbasierten Verfahren entgegenzuwirken. Im Gegensatz zu diesen wird hier nur ein einziger Plan erstellt und schon bei dessen Generierung darauf geachtet, dass dieser möglichst gut ist. Um dies zu bestimmen wird wieder die Heuristik (siehe 3.3) benötigt.

Die grobe Vorgehensweise des *Greedy-Generators* ist dabei folgende: Angefangen mit einem komplett leeren Stundenplan erstellt der Generator eine Liste mit allen Kombinationen, die beschreiben, auf welche Weise ein erster Stundenplaneintrag gesetzt werden kann. Jede dieser Kombinationen enthält einen zu setzenden Zeitslot (repräsentiert durch **Tag** und **Stunde**), eine **Klasse** oder Gruppe, ein zu unterrichtendes **Fach**, einen **Lehrer** und einen **Raum**. Da die Anzahl dieser Kombinationen sehr hoch ist (durch die 6 Einträge im Bereich von $O(n^6)$), wird schon hier eine Vorauswahl nach den *System-Bedingungen* vorgenommen. So werden Kombinationen, deren Lehrer das enthaltene Fach gar nicht unterrichtet, oder solche, bei denen der angegebene Raum zu dieser Zeit gar nicht verfügbar ist, usw. gar nicht erst in diese Liste eingetragen. Anschließend wird

jede Kombination einzeln dem momentanen Plan hinzugefügt und das Ergebnis mittels der Heuristik bewertet. Diese Bewertung stellt jedoch nur eine Abschätzung dar, da der komplette Stundenplan zu dieser Zeit noch gar nicht verfügbar ist. Nun wählt der Generator diejeninge Kombination, die zur besten Bewertung geführt hat aus, setzt diese im Plan fest und löscht sie aus der Liste der Kombinationen. Anschließend werden die Kombinationen, die durch das Setzen der letzten Kombination ungültig geworden sind (z.B. weil die Klasse nun genug Stunden für ein Fach hat oder weil dieser Zeitslot für eine Klasse nun belegt ist), aus der Liste entfernt. Die übrig gebliebenen Kombinationen werden nun wieder bewertet, die beste davon ausgewählt und so weiter.

Der Prozess endet hierbei, wenn die Liste der Kombinationen leer ist, was immer dann der Fall ist, wenn alle Fächer für alle Klassen korrekt verteilt wurden (positiver Ausgang) oder keine Kombination zum Platzieren eines Faches mehr gefunden werden konnte (negativer Ausgang). Da die Liste der Kombinationen trotz einer frühen Filterung immer noch sehr groß ist, führt dies zu einer sehr hohen Laufzeit, da ja jede Kombination mehrfach bewertet werden muss. Abhilfe schaffte hier ein Vorgehen in mehreren Runden, wobei in den ersten Runden nur die schwer zu platzierenden Fächer verteilt werden und in der letzten Runde die regulären. Jede Runde kann hierbei wieder klassenweise ablaufen, was die Liste zusätzlich enorm verkleinert. Auf die beschriebene Weise funktioniert der Greedy-Generator ähnlich wie der Startlösungsgenerator (siehe Kapitel 3.4), nur dass die Einträge in diesem Fall nicht zufällig gesetzt werden, sondern aufgrund der aktuellen Information immer der beste ausgewählt wird.

# Kapitel 4

# Das Verfahren zur Bewertung der Stundenpläne

## 4.1 Anforderungen

In Kapitel 3 wurde von einer Heuristik ausgegangen, welche den Suchalgorithmen zur Bewertung eines Stundenplans zur Verfügung steht. Dabei wird diese Bewertung durch einen Zahlenwert repräsentiert, welcher für eine Bestrafung des Plans steht. Je höher dieser ist, desto schlechter ist also der bewertete Plan. Bei dieser Bewertung stellt sich die Frage, welche Kriterien einen guten bzw. schlechten Stundenplan auszeichnen. Ein Kriterium für einen schlechten Stundenplan wäre beispielsweise, dass er viele Freistunden für die Schüler aufweist. Ein Gutes wäre hingegen, dass die Stundenanzahl für eine Klasse an allen Tagen etwa gleich ist.

Doch kann man nicht im Allgemeinen sagen, was einen guten Stundenplan auszeichnet, da sich die speziellen Kriterien von Schule zur Schule unterscheiden können. Auch innerhalb einer Schule können diese sich im Laufe der Zeit verändern. Ein Beispiel für eine Veränderung der Bewertungskriterien wäre die folgende Situation: Angenommen ein Lehrer möchte aus persönlichen Gründen Montags und Dienstags erst zur dritten Stunde mit dem Unterricht beginnen. Nun kann kann man ihm entgegenkommen, indem man die Bewertungskriterien so ändert, dass Pläne, die dem Wunsch des Lehrers nicht entsprechen, etwas schlechter bewertet werden als ansonsten gleichwertige Pläne, die diesem beachten.

Es sollte also möglich sein, die Bewertungskriterien mit deren Hilfe ein Stundenplan von der Heuristik bewertet wird, zu verändern, damit das gesamte System flexibel einsetzbar bleibt. Eine Möglichkeit, dies zu gewährleisten, wäre es, eine sorgfältige Anforderungsanalyse zu betreiben, um festzustellen welche Kriterien sich üblicherweise ändern bzw. ändern könnten. Im Hinblick auf die obige Situation könnte man dem Benutzer beispielsweise eine Oberfläche anbieten, welche es ihm erlaubt, Zeiten festzulegen, zu denen

bestimmte Lehrer nicht verfügbar sind. Zusätzlich müssten Verstöße gegen diese Verfügbarkeitsbedingungen entsprechend geahndet werden. Es ist jedoch unwahrscheinlich, dass ein Entwickler alle denkbaren Bewertungskriterien und deren dynamische Änderung antizipieren kann. Aus diesem Grund haben wir ein System entwickelt, dass es erlaubt, die Bewertungskriterien anhand von variablen Constraints zu definieren.

**Definition 6 (Constraints)** *Mit Constraints bezeichnen wir in unserem Fall Bedingungen für den zu generierenden Stundenplan. Diese Bedingungen können mit Hilfe einer von uns erstellten Sprache definiert werden. Wir unterscheiden zwischen zwei verschiedenen Arten von Constraints:*

- *Ein **hartes Constraint** ist eine Bedingung, die ein Stundenplan erfüllen muss. Wird ein hartes Constraint durch einen Plan verletzt, so nennen wir diesen Plan **ungültig**.*
- *Ein **weiches Constraint** ist eine Bedingung, die ein Stundenplan erfüllen sollte, aber nicht muss. Jedem weichen Constraint ist eine Zahl, das **Penalty**, zugeordnet, welche für die Bestrafung im Falle einer Nicht-Einhaltung des Constraints steht.*

*Ein gültiger Plan ist umso besser, je geringer seine Gesamt-Bestrafung ist.*

Angenommen, es soll ein Stundenplan für eine Grundschule erstellt werden. Ein Beispiel für ein hartes Constraint wäre dann, dass es keine Freistunden für die Schüler geben darf, da diese immer beaufsichtigt werden müssen. Ein Beispiel für ein weiches Constraint wäre hingegen, dass Mathematik möglichst immer zu Anfang des Tages liegen sollte, da dieses Fach die Konzentration der Schüler mehr beansprucht als andere Fächer.

Die zur Definition der Constraints verwendete Sprache ähnelt der Prädikatenlogik und wird in Abschnitt 4.3 näher erläutert. Zunächst wird jedoch in Abschnitt 4.2 ein Überblick über das System zur Behandlung von Constraints gegeben. In Abschnitt 4.4 wird auf die Implementierung dieses Systems eingegangen.

## 4.2 Überblick

Abbildung 4.1 gibt eine Übersicht über das Bewertungs-Subsystem und den grundsätzlichen Ablauf bei der Auswertung eines Stundenplanes anhand von vorher definierten Constraints.

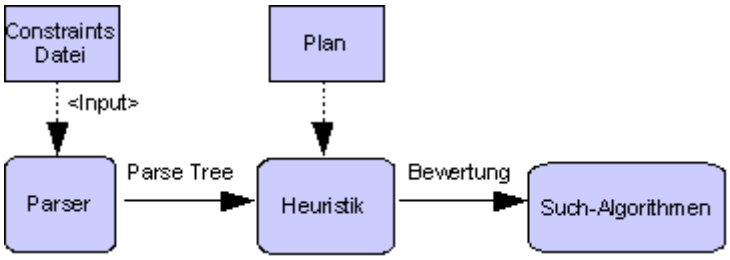


Abbildung 4.1: Das Constraint-System

Die Constraints können in einer Datei definiert werden, die dem Parser als Eingabe zur Verfügung gestellt wird. Die Sprache zur Definition der harten und weichen Constraints in dieser Datei wird im Folgenden als *Constraints-Sprache* bezeichnet. Der Parser liest die Datei mit den Constraint-Definitionen ein und erzeugt daraus einen abstrakten Parse-Tree. Dies ist ein Baum, welcher die syntaktische Struktur eines eingelesenen Strings bezüglich einer formalen Grammatik repräsentiert (analog zu einem Compiler, siehe z.B. [UAS99]). Die Grammatik der *Constraints-Sprache* wird in Abschnitt 4.3 erläutert.

Anhand des Parse-Tree kann die Heuristik Stundenpläne bewerten, worauf genauer in Abschnitt 4.4 eingegangen wird. Die Suchalgorithmen haben damit die Möglichkeit, einen Stundenplan anhand der harten Constraints auf Gültigkeit zu überprüfen (liefert einen Wahrheitswert) oder anhand der weichen Constraints zu bewerten (gibt eine Zahl zurück).

## 4.3 Grammatik

In diesem Abschnitt wird die Grammatik vorgestellt, die die Syntax der **Constraints-Sprache** definiert. Dabei werden jeweils logisch zusammenhängende Ausschnitte dieser Sprache nacheinander präsentiert und erläutert. Nicht alle Produktionsregeln werden dabei bis auf die unterste Ebene vorgestellt, um den Leser nicht unnötig zu irritieren. Die Ausführungen beziehen sich dabei nicht nur auf die Syntax, sondern auch auf die Semantik der Sprache. Dem Leser soll insgesamt ein Eindruck vermittelt werden, wie sie zur Beschreibung der Constraints verwendet werden kann. Die Grammatik wird in der *Backus-Naur-Form* (BNF) dargestellt. Die BNF ist eine Metasprache zur Darstellung kontextfreier Grammatiken (siehe [Gar98]).

```
<Constraints>    ::= (<HardConstraint> | <SoftConstraint>)*
<HardConstraint> ::= <Hard> ":" <Expression> [<Comment>]
<Hard>           ::= "hard" | "HARD"
<SoftConstraint> ::= <SOFT> "["<Float\_Literal>"]:" <Expression> [<Comment>]
<Soft>           ::= "soft" | "SOFT"
```

Constraints sind also entweder harte oder weiche Constraints. Bei letzteren muss in eckigen Klammern das Penalty dieses Constraints angegeben werden. Beispielsweise würde ein Penalty von 5 bedeuten, dass für jede Verletzung des Constraints die Bewertung des Stundenplans um 5 schlechter wird. Für harte Constraints wird kein Penalty angegeben, da es hier nur um die Gültigkeit des Plans geht. `<Float_Literal>` steht für eine Float-Zahl und `<Comment>` steht für einen Kommentar, der optional hinter das Constraint geschrieben werden kann, um es genauer zu erklären. Ein weiches Constraint hätte also die folgende Form, wobei hier `<Expression>` als Platzhalter für die im Folgenden definierte Produktionsregel steht:

**Beispiel 4.1**

```
SOFT[15]: <Expression> /* Test-Constraint */
```

```
<Expression>          ::= <Quantor> | <BracketedExpression> | <UnaryExpression>
<Quantor>             ::= <ImpliesExpression> | (<QUANTOR> ":" <Quantor>)
<ImpliesExpression>   ::= <OrExpression> ["->" <ImpliesExpression>]
<OrExpression>        ::= <AndExpression> ["|" <OrExpression>]
<AndExpression>       ::= <Negation> ["\&" <AndExpression>]
<BracketedExpression> ::= "("<Expression>")"
<Negation>            ::= ["!"] (<BracketedExpression> | <UnaryExpression>)
```

`<Expression>` definiert die Syntax des Ausdrucks, der die eigentliche Bedingung, die der Plan erfüllen soll, darstellt. Hierbei handelt es sich um logische Formeln die mit

∀-, ∃-*Quantoren*, *Negationen*, *Impliziert-*, *Oder-* sowie *Und*-Verknüpfungen erstellt werden. Die Ausdrücke, die mit diesen logischen Operatoren verknüpft werden können, sind durch `<UnaryExpression>` definiert. Dabei muss es sich um Ausdrücke handeln, die zum Wahrheitswert `true` oder `false` ausgewertet werden.

Die Produktionsregel `<QUANTOR>` definiert die Syntax für einen Quantor. Bei einem solchen kann es sich entweder um einen ∀- (beschrieben durch `forall`) oder ∃-Quantor (dargestellt durch `exists`) handeln. In Klammern folgen dann die an diesen gebundenen Variablen mit ihren Typen[1]. Ein Beispiel für die Anwendung eines Quantors in einem harten Constraint wäre:

**Beispiel 4.2**

```
HARD: FORALL(g IN group): <UnaryExpression> & <UnaryExpression>
```

Bei einer `<UnaryExpression>` handelt es sich entweder um die einfachen Wahrheitswerte `true` oder `false`, um Relationen oder um Vergleichsausdrücke.

```
<UnaryExpression> ::= <BooleanExp> | <Relation> | <CompareExp>
<BooleanExp>      ::= "TRUE" | "true" | "FALSE" | "false"
```

Relationen sind semantisch im gleichen Sinne wie die Relationen der Prädikatenlogik zu verstehen. Im unserem Falle wäre `TEACHERCLASS(teacher, group)` ein einfaches Beispiel für eine Relation. An die erste Stelle der Relation kann eine Variable vom Typ `teacher`, an die zweite Stelle eine vom Typ `group` gesetzt werden. Die Relation ist genau dann wahr, wenn der entsprechende Lehrer der Klassenlehrer dieser Klasse ist. In eine Relation können an Quantoren gebundene Variablen oder Konstanten (z.B. `1a` für `group`) eingesetzt werden und so der Wahrheitswert der Relation in Bezug auf diese Variablen abgefragt werden. Desweiteren können Stellen in der Relation leer gelassen werden, was bedeutet, dass hier bei der Auswertung der Relation der Wert an dieser Stelle keine Rolle spielt. Beispielsweise würde `TEACHERCLASS(Müller,)` zu `true` ausgewertet werden, wenn der Lehrer `Müller` der Klassenlehrer einer beliebigen Klasse ist. Jetzt sind wir in der Lage, ein erstes vollsändiges Constraint zu formulieren. Beispielsweise sollte jede Klasse einen Klassenlehrer zugeordnet haben:

**Beispiel 4.3**

```
HARD: FORALL(g IN group): TEACHERCLASS(,g)
```

```
<CompareExp>        ::= <CExpression> <Op\_Comp> <CExpression>
<Op\_Comp>          ::= "=" | "!=" | ">" | "<" | ">=" | "<="
<CExpression>       ::= <AdditiveExp> | <BracketedCExp> | <UnaryCompExp>
```

[1] Die verfügbaren Typen sind: `group`, `grade teacher`, `subject`, `room`, `roomtype`, `day` und `hour`

```
<BracketedCExp>     ::= "(" <CExpression> ")"
<AdditiveExp>       ::= <MultiplicativeExp> [("+" | "-") <AdditiveExp>]
<MultiplicativeExp> ::= <UnaryCompExp> [("*" | "/") <MultiplicativeExp>]
<UnaryCompExp>      ::= <Function> | <Identifier> | <Float\_Literal>
                      | <BracketedCExp>
```

Mit Vergleichsausdrücken (`<CompareExp>`) können jeweils zwei Bezeichner (`<Identifier>`), Funktionen (`<Function>`) und Zahlen (`<Float_Literal>`) untereinander verglichen werden. Es stehen sechs Vergleichsoperatoren zur Verfügung (`=`, `!=`, `>`, `<`, `>=`, `<=`). Innerhalb eines Vergleichsausdrucks ist Arithmetik erlaubt, deren Nutzung für die Definition mancher Constraints unerlässlich ist.

Funktionen geben für eine Variablenbelegung einen Zahlenwert zurück. Ein Beispiel für eine Funktion wäre `TEACHERHOURS[teacher]`, welche die Anzahl der Stunden zurückgibt, die ein Lehrer in einer Woche unterrichten muss. Syntaktisch unterscheiden sich Funktionen von Relationen dadurch, dass ihre Belegung in eckigen statt in runden Klammern steht.

Vergleiche von Zahlen sowie anderen Variablentypen wie `group` oder `teacher` sind möglich. Desweiteren können auch Konstanten durch Bezeichner dargestellt und verglichen werden (z.B. `Müller`). So können auch Vergleiche der folgenden Art angestellt werden: ...t = Müller ... (t sei eine Variable vom Typ teacher). Bei dieser Art von Vergleichen macht es natürlich nur Sinn, die Vergleichsoperatoren `=` und `!=` zu verwenden.

Um die Anwendung der Constraints-Sprache zu verdeutlichen, definieren wir im Folgenden das Constraint, welches in Abschnitt 4.1 dieses Kapitels als Beispiel verwendet wurde. Hier wurde gesagt, man könne den Wunsch eines Lehrers, Montags und Dienstags erst ab der dritten Stunde zu unterrichten, als weiches Constraint formulieren. Um dieses zu definieren, benötigen wir eine Relation, die Informationen aus dem übergebenen Stundenplan repräsentiert. Diese Relation ist mit dem Namen `LESSON(day,hour,teacher,group,room,subject)` vorgegeben und kann zu diesem Zweck benutzt werden. Ein Eintrag in dieser Relation bedeutet, dass die Klasse (oder zusammengesetzte Gruppe) `group` am Tag `day`[2] zur Stunde `hour`[3] in Raum `room` bei Lehrer `teacher` im Fach `subject` unterrichtet wird.

Das Constraint kann nun folgendermaßen formuliert werden (der besagte Lehrer heiße Müller):

[2]Die Zählung der Tage beginnt bei 0 (also 0 = Montag, 1 = Dienstag usw.)
[3]Die Zählung der Stunden beginnt ebenfalls bei 0

**Beispiel 4.4**

```
soft[1]: forall(d IN day, h IN hour): LESSON(d,h,Müller,,,) & (d=0 | d=1)
-> h >= 2
```

In Worten besagt dieses Constraint: Wenn der Lehrer Müller Unterricht gibt und dieser Unterricht Montags oder Dienstags stattfindet, dann darf diese Stunde frühestens die dritte sein. Zu beachten ist weiterhin, dass in die letzten drei Stellen der Abfrage der Relation nichts eingesetzt werden muss, da es nicht relevant ist, in welchem Fach, in welcher Klasse oder in welchem Raum der Unterricht stattfindet. Über das Penalty kann der Stellenwert des Constraints gesteuert werden.

## 4.4 Implementierung

In Abschnitt 4.2 wurde gezeigt, dass die Datei mit den Constraints von einem Parser eingelesen wird. Dieser Parser erzeugt dann einen Parse-Tree auf Basis der in Abschnitt 4.3 vorgestellten Grammatik. Die Implementierung dieses Parsers und die Umsetzung der einzelnen Knoten im Parse-Tree ist Gegenstand dieses Kapitels.

### 4.4.1 JavaCC und JJTree

Zur Entwicklung des Parsers wurde der Parsergenerator JavaCC[4] (mit der Erweiterung JJTree) verwendet. JavaCC kann einen Parser in der Programmiersprache Java[5] für eine gegebene Grammatik erzeugen, welche in der *extended Backus-Naur-Form* angegeben wird (siehe [MW04]). JJTree ist ein Präprozessor für JavaCC, der den JavaCC-Code um Aktionen, die einen Parse-Tree erstellen und zurückliefern können, erweitert. Durch JJTree wird also eine Datei erzeugt, die JavaCC als Eingabe dient, um einen Parser zu erstellen, der einen Parse-Tree für die eingelesenen Constraints aufbaut. Für jeden Knoten-Typ im Parse-Tree wird hierbei eine Java-Klasse erzeugt, die diesen repräsentiert.

Wie man in JJTree spezifizieren kann, dass für eine Produktionsregel eine enstprechende Knoten-Klasse erzeugt wird, soll anhand eines Beispiels für die Produktionsregel von `<SoftConstraint>` verdeutlicht werden. Diese war in Abschnitt 4.3 folgendermaßen definiert:

```
<SoftConstraint> ::= <SOFT> "["<Float_Literal>"]:" <Expression> [<Comment>]
```

Die Spezifikation für diese Produktionsregel innerhalb von JJTree sieht folgendermaßen aus:

```
ASTSoftConstraint SoftConstraint() #SoftConstraint :
{
    Token t = new Token();
    Token comment = new Token();
}
{
    <SOFT> "[" t = <FLOAT_LITERAL> "]: " Expression() [comment = <COMMENT>]
    {
        jjtThis.penalty = t.toString();
```

[4] siehe http://javacc.dev.java.net
[5] siehe http://www.java.com

```
        jjtThis.setComment(comment.toString());
        return jjtThis;
    }
}
```

`ASTSoftConstraint` ist der Name der Klasse, die generiert wird und den Knoten für die Produktionsregel `<SoftConstraint>` repräsentiert. Variablen vom Typ Token können definiert werden, um bestimmte Teile der eingelesenen Zeichenkette zu speichern. In diesem Fall wird das Penalty des Constraints (`<Float_Literal>`) in die Varaible `t` und der Kommentar für dieses Constraint(`<COMMENT>`) in die Variable `comment` gespeichert. Die Variable `jjtThis` bezeichnet das aktuelle Knotenobjekt analog zum Keyword `this` in Java. So können mit Hilfe von `jjtThis` und den Tokens eingelesene Informationen in den erzeugten Objekten gespeichert werden. In diesem Fall hat die Klasse `ASTSoftConstraint` die Attribute `penalty` und `comment`, in die das Penalty des Constraints sowie dessen Kommentar gespeichert werden.

Die in JJTree definierte Regel enspricht also einem weichem Constraint in der vorgestellten Grammatik. Man sieht, dass für den Ausdruck `<Expression>` der Grammatik wiederum ein Knoten-Objekt erzeugt wird (`Expression()`), welches im Parse-Tree dem `ASTSoftConstraint`-Objekt als Kindknoten zugeordnet wird. Für die Produktionsregeln `<SOFT>`, `<FLOAT_LITERAL>` und `<COMMENT>` werden keine Knoten benötigt, sie können also in JJTree als Terminalsymbole definiert werden. Für die Regeln `<SOFT>` und `<HARD>` sieht das folgendermaßen aus:

```
TOKEN: /* Hard- und Soft-Constraints */
{
   < SOFT: "soft" | "SOFT" >
|  < HARD: "hard" | "HARD" >
}
```

Das Beispiel soll allgemein verdeutlichen, wie man eine Knoten-Klasse für eine Produktionsregel erzeugen kann und wie man den Parser so definiert, dass dieser beim Einlesen der Zeichenkette ein entsprechendes Objekt generiert und gewonnene Informationen darin speichert. Diese Informationen können dann bei der Auswertung des Parse-Tree verwendet werden, um einen Wahrheitswert dieses Constraints in Bezug auf den aktuellen Stundenplan zu ermitteln. Im späteren Verlauf des Kapitels wird auf die Implementierung der wichtigsten Knoten des Parse-Tree eingegangen.

### 4.4.2 Evaluierung des Parse-Tree

Um den erzeugten Parse-Tree auf einem übergebenen Stundenplan auszuwerten, wurde in jeder Knoten-Klasse eine `evaluate`-Methode implementiert, welche einen boolschen

Wahrheitswert zurückliefert. Diese Methode bekommt als Parameter den Stundenplan und ein Objekt, das eine Variablenbelegung speichert, übergeben. Da in unserer Sprache zur Beschreibung der Constraints nur gebundene Variablen erlaubt sind, wird die Zuweisung von Werten an Variablen in diesem Belegungsobjekt nur von den Quantor-Knoten vorgenommen, allen darunter liegenden Knoten stehen diese dann darin zur Verfügung. Beispielsweise können die Knoten, die eine Relation repräsentieren, diese Informationen dann nutzen, um die in der Abfrage benutzten Variablen durch konkrete Werte zu ersetzen.

### 4.4.3 Die Constraints-Knoten

Wie in Kapitel 4.3 angegeben, ist der Wurzelknoten eines Parse-Tree vom Typ `<Constraints>`. Die Knoten für die Produktionsregeln `<SoftConstraint>` sowie `<HardConstraint>` sind direkte Nachfolgerknoten dieses Wurzelknotens. Diese beinhalten wiederum einen Verweis auf den zugeordneten Ausdruck vom Typ `<Expression>`, durch den das jeweilige Constraint beschrieben wird. Der Parse Tree einer eingelesenen Zeichenkette mit zwei weichen Constraints und einem harten Constraint würde also folgendermaßen aussehen:

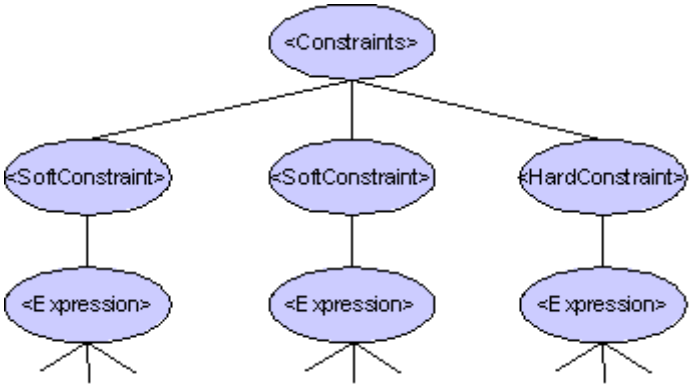


Abbildung 4.2: Die ersten drei Ebenen eines Parse Trees

Anhand des Parse-Tree kann dann die Heuristik erzeut werden, die von den Suchalgorithmen zur Bewertung des Plans genutzt wird. Hierbei werden für alle Constraints Objekte erzeugt, welche in der Heuristik gespeichert werden. Diese können dann zur der Bewertung eines Planes herangezogen werden.

Zur Auswertung eines Constraints muss der jeweilige Unterbaum ab dem Knoten evaluiert werden, der den Ausdruck, welcher dem Constraint zugeordnet ist, darstellt (`<Expression>`). Handelt es sich um ein hartes Constraint, wird ein Wahrheitswert zurückgegeben, der besagt, ob das Constraint erfüllt ist oder nicht. Handelt es sich um ein weiches Constraint, wird anhand des Penalties und der Anzahl der Verletzungen ein

enstprechender Zahlenwert errechnet. Wie festgestellt wird, wie oft ein weiches Constraint verletzt ist, wird in Abschnitt 4.4.4 genauer beschrieben.

Ein Constraint-Objekt speichert also den angesprochenen Ausdruck in Form eines Parse-Tree, welcher zur Evaluierung benötigt wird. Die in Abschnitt 4.4.2 vorgestellte `evaluate`-Methode muss also für den Knoten der Produktionsregel `<Expression>` sowie für alle seine möglichen Nachfolgerkonten implementiert werden, damit diese bei der Bewertung eines Plans herangezogen werden kann.

In den folgenden Abschnitten wird näher auf die Implementierung der wichtigsten Knoten im Parse-Tree eingegangen.

### 4.4.4 Quantoren

Im Laufe dieses Projektes stellte sich schnell heraus, dass die Auswertung der Quantoren die Laufzeit einer Bewertung extrem beinflusst, weswegen diese speziell optimiert wurden. Zunächst wird nun auf die ursprüngliche Version der Quantoren eingegangen; später werden die Maßnahmen zur Laufzeitoptimierung näher beleuchtet.

#### Implementierung der Quantoren

Zur Definition der meisten Constraints müssen Qunatoren angewandt werden. In diesem Abschnitt wird die Implementierung der Klasse für einen Quantorknoten beleuchtet. Es gibt dabei zwei Typen von Quantoren: $\forall$- und $\exists$-Quantoren. Diese werden in der *Constraints-Sprache* mit `forall` bzw. `exists` bezeichnet. In den Klammern hinter dem Quantor werden die Variablen, die an diesen gebunden sind, sowie deren Typen angegeben.

Ein Quantorknoten hat immer nur einen direkten Nachfolgerknoten. Dieser Knoten ist der Wurzelknoten des Teilausdrucks, auf den sich dieser bezieht. In dessen `evaluate`-Methode muss also der Baum unterhalb des Quantors für jede Belegungskombination der an diesen gebundenen Variablen ausgewertet werden. Handelt es sich um einen $\forall$-Quantor, so gibt die `evaluate`-Methode `true` zurück, wenn der Unterbaum für jede mögliche Belegungskombination ebenfalls zu `true` ausgewertet wird. Entsprechend muss der Unterbaum eines $\exists$-Quantors nur für eine mögliche Belegungskombination wahr sein, damit die `evaluate`-Methode insgesamt wahr wird.

Ein Quantor erzeugt also eine Belegungskombination der an ihn gebundenen Variablen, indem er zu jedem Typ einen möglichen Wert bestimmt. Diese werden in das bereits angesprochene Objekt zur Repräsentation der Variablenbelegung geschrieben und der

`evaluate`-Methode übergeben. Nach einer Auswertung des Unterbaums wird eine neue Belegungskombination generiert und der Unterbaum anhand dieser erneut ausgewertet. Dies geschieht solange, bis alle Kombinationen erschöpft sind und ein Gesamtergebnis vorliegt. Handelt es sich um einen ∀-Quantor, kann der Bewertungs-Vorgang natürlich abgebrochen werden, wenn schon die Auswertung des Unterbaums für eine Belegungskombination `false` ergibt. Ein analoger Weg wird auch bei einem ∃-Quantor gewählt.

Bei der Auswertung des Quantors muss immer unterschieden werden, ob hier ein hartes oder um ein weiches Constraint vorliegt. Handelt es sich um ein hartes Constraint, so genügt das oben genannte Vorgehen, da nur ein Wahrheitswert zurückgegeben werden muss. Handelt es sich jedoch um ein weiches Constraint, so muss eine Zahl ermittelt werden, die der Bewertung des Plan bezüglich des entsprechenden Constraints entspricht. Eine Möglichkeit, weiche Constraints zu behandeln wäre, dass das Constraints-Objekt einfach den gespeicherten Baum auswertet und den Wert des Penalties zurückgibt, wenn die Auswertung `false` ergibt. Anderfalls könnte der Wert 0 zurückgegeben werden. Ein solches Verfahren wäre allerdings in der Praxis nicht zweckmäßig, was im Folgenden erläutert werden soll.

Angenommen, man hat ein weiches Constraint definiert, welches besagt, dass Montags kein Unterricht vor der dritten Stunde stattfinden soll. Wir betrachten die Ausschnitte von zwei unterschiedlichen Stundenplänen für zwei Klassen, in denen dieses Constraint verletzt ist.

| | 1a | 1b |
|---|---|---|
| Mo, 1. | | |
| Mo, 2. | Deutsch<br>Müller<br>102 | |
| Mo, 3. | Sachkunde<br>Meier<br>101 | Deutsch<br>Müller<br>102 |

| | 1a | 1b |
|---|---|---|
| Mo, 1. | Mathematik<br>Müller<br>101 | Mathematik<br>Meier<br>103 |
| Mo, 2. | Deutsch<br>Müller<br>102 | Mathematik<br>Kunz<br>103 |
| Mo, 3. | Sachkunde<br>Meier<br>101 | Deutsch<br>Müller<br>102 |

Abbildung 4.3: Beispiel Stundenpläne

Betrachtet man die beiden Pläne, so sieht man leicht, dass der linke Plan bezüglich des Constraints intuitiv besser ist als der rechte. Im linken Plan wird das Constraint durch eine Stunde verletzt, während es im rechten Plan durch vier Unterrichtseinheiten verletzt wird. Trotzdem würden beide Pläne von der Heuristik gleich bewertet werden, wenn man die oben geschilderte Lösung anwendet. Dieses Verfahren liefert zu undifferenzierte Bewertungen, mit denen in einem realistischen Szenario nicht gearbeitet werden kann.

Aus diesem Grund haben wir uns entschieden, den äußeren Quantor eines weichen Constraints die Anzahl der Belegungskombinationen, für die das Constraint verletzt ist, zählen zu lassen. Dieses Zählen macht nur Sinn, wenn der äußere Quantor entweder ein $\forall$- oder ein negierter $\exists$-Quantor ist. Handelt es sich bei den äußeren Quantoren um einen negierten $\forall$- oder um einen $\exists$-Quantor, ist ein Zählen der Belegungskombinationen nicht von Nöten. Das folgende Beispiel zeigt, warum.

**Beispiel 4.5**
Betrachtet werden folgende vier Constraints, deren Bedeutung durch den dahinterstehenden Kommentar zusätzlich erläutert wird:

```
soft[1]: forall(d IN day, g IN group): LESSON(d,0,,g,,)
/* Alle Klassen sollten an jedem Tag in der ersten Stunde Unterricht haben */

soft[1]: !forall(d IN day, g IN group): LESSON(d,0,,g,,)
/* Nicht alle Klassen sollten an jedem Tag in der ersten Stunde
Unterricht haben */

soft[1]: exists(d IN day, g IN group): LESSON(d,0,,g,,)
/* Es existiert mindestens eine Klasse, die an einem Tag in der
ersten Stunde Untrerricht hat */

soft[1]: !exists(d IN day, g IN group): LESSON(d,0,,g,,)
/* Es soll keine Klasse existieren, die an einem Tag zur ersten
Stunden Unterricht hat */
```

Bei dem ersten und dem vierten Constraint wird eine Aussage darüber getroffen, wie oft eine Klasse zur ersten Stunde Unterricht haben sollte. Nämlich im ersten Fall immer oder im vierten Fall nie. Dies bedeutet: je öfter gegen das jeweilige Constraint verstoßen wird, desto schlechter sollte ein Plan bewertet werden, da hier jeder Verstoß einzeln Verschlechterung darstellt.
Bei dem zweiten und dem dritten Constraint wird nur eine Aussage darüber gemacht, dass es mindestens einen Fall geben sollte, in dem eine Klasse nicht zur ersten Stunde bzw. zur ersten Stunde Unterricht hat. Bei diesen Constraints macht es also aufgrund ihrer Sematik keinen Sinn, alle Belegungskombinationen zu zählen, um den Plan entsprechend schlechter zu bewerten, da hier nur gefragt ist, ob ein solcher Fall überhaupt exisitiert oder nicht.

### Optimierung der Quantoren

Die Funktionsweise eines Quantors wurde im vorherigen Abschnitt beschrieben. Ein solcher muss zur Auswertung seinen Unterbaum für jede mögliche Variablenkombination evaluieren. Angenommen, es sind drei Variablen an den Quantor gebunden, die jeweils mit 10 verschiedenen Werten belegt werden können. Dann muss dieser Unterbaum 1000 mal evaluiert werden, da es genau so viele Belegungskombinationen gibt. Weil diese Evaluierung relativ viel Zeit benötigt, haben wir eine Strategie entwickelt, durch die ein Quantor ausgewertet werden kann ohne den Unterbaum für jede Belegungskombination auszuwerten.

Da in diesem Projekt hauptsächlich nachbarschaftsbasierte Verfahren zur Suche eines möglichst guten Planes eingesetzt werden, sind die nacheinander beurteilten Pläne aufgrund der Arbeitsweise der Iteratoren (siehe 3.2) sehr ähnlich. Durch die starke Ähnlichkeit der Pläne führt die Evaluierung des Ausdrucks unter dem Quantor für die meisten Belegungskombinationen zum selben Ergebnis. Zur Veranschaulichung dient das folgende Constraint:

**Beispiel 4.6**

```
soft[1]: forall(d IN day): !LESSON(,0,,,)
/* An jedem Tag sollte zur ersten Stunde kein Unterricht stattfinden */
```

Tauscht ein Iterator nun am dritten Tag einen Stundenplaneintrag von der zweiten in die erste Stunde, ändert sich die Auswertung des Ausdrucks unter dem ∀-Quantor nur für die Belegung: `d = 2`. Hätte man also eine Strategie, die nur Belegungskombinationen auswertet, an denen sich auch etwas geändert hat, so würde man sich in diesem Beispiel 4 Evaluierungen[6] sparen. Offensichtlich würde diese Strategie im Vergleich zum normalen Verfahren umso effektiver werden, je mehr Belegungskombinationen es gibt.

Um diese Strategie umzusetzten, erzeugt der Quantor bei der ersten Evaluierung eine Tabelle mit den Auswertungsergebnissen für alle Belegungskombinationen. Bei den späteren Auswertungen werden nur noch die Belegungskombinationen neu ausgewertet, deren Wahrheitswert sich möglicherweise geändert hat. Hierzu muss der Quantor natürlich wissen, welche Kombinationen dies sind. Daher speichert ein Stundenplan in zwei separaten Mengen die gelöschten und neu hinzugefügten Stundenplaneinträge (Tauschen wird von den Iteratoren hier als Kombination von Löschen und Hinzufügen benutzt). Diese Information ist dann einem Quantor zugänglich, welcher anhand dieser Daten überprüfen und entscheiden kann, welche Kombinationen neu geprüft werden müssen.

Im obigen Beispiel kann der Quantor anhand dieser Informationen erkennen, dass an Tag 2 eine Stunde gelöscht sowie eine Stunde eingefügt wurde. Also muss der Unter-

[6] nämlich die der Tage 0,1,3,4

baum auch nur für die Belegung `d = 2` neu ausgewertet werden. Es kann allerdings spezielle Situationen geben, in denen es bei Anwendung dieser Strategie zu fehlerhaften Auswertungen kommt. Eine solche Situation wird im Folgenden kurz aufgezeigt. Betrachtet werden zwei benachbarte Pläne, die nacheinander bewertet werden sollen.

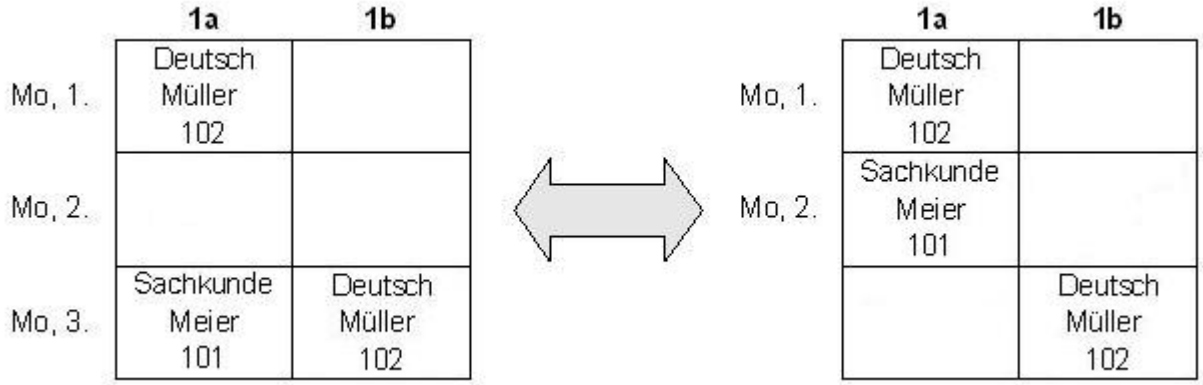


Abbildung 4.4: Beispiel Stundenpläne

Die beiden Pläne werden anhand des folgenden Constraints bewertet.

**Beispiel 4.7**

```
soft[1]: forall(g IN group, d IN day, h IN hour): exists(h1 IN hour):
           LESSON(d,h,,g,,) -> (LESSON(d,h1,,g,,) & h1 = h + 1.0)
                | !(exists(h2 IN hour): LESSON(d,h2,,g,,) & h2 > h)
/* Keine Freistunden */
```

Dieses Constraint beschreibt die Bedingung, dass eine Klasse keine Freistunden haben soll. Genauer besagt dieses, dass an jedem Tag nach jeder Stunde entweder noch eine weitere Stunde, oder aber keine weitere Stunde folgen sollte.

Der linke Plan in Abb. 4.4 würde bezüglich dieses Constraints mit dem Wert 1 bewertet werden, da das Constraint einmal verletzt ist. Wird jetzt der zweite Plan, der aus dem zweiten durch Iteration hervorgegangen ist, nach der optimierten Strategie bewertet, kommt es zu einem Fehler. Der Quantor erkennt anhand der zusätzlichen Informationen über gelöschte und eingefügte Stunden, dass ein Eintrag in der Klasse 1a Montags in der dritten Stunde gelöscht wurde und ein Eintrag in der Klasse 1a Montags in die zweite Stunde eingefügt wurde. Es werden also die beiden Kombinationen: `g = 1a`, `d = 0`, `h = 1` sowie `g = 1a`, `d = 0`, `h = 2` neu bewertet. Die Auswertung dieser Kombinationen ergibt aber keinen Unterschied, da das Constraint durch die Kombination `g = 1a`, `d = 0`, `h = 0` verletzt ist und nicht durch die beiden neu überprüften. Zu beachten ist hier, dass die Bewertung nicht falsch wäre, wenn in der Klasse `1a` der Eintrag der ersten Stunde in die zweite Stunde getauscht werden würde. Hier würde aufgrund der Formulierung des Constraints erkannt werden, dass es im rechten Plan keine Freistunde mehr gibt.

Zusammenfassend kann man sagen, dass Constraints problematisch sein können, deren Aussagen sich nicht nur auf einen Stundenplaneintrag sondern auf eine Kombination von Einträgen beziehen. Bei diesem optimierten Verfahren handelt es sich also um eine Abschätzung. In der Praxis hat sich herausgestellt, dass dessen Anwendung kaum Qualitätseinbußen mit sich bringt, jedoch die Performanz erheblich verbessert. Tabelle 4.1 belegt diese Aussage. Benutzt wurde hier ein kleines Datenset mit 3 Klassen und je 13 zu verteilenden Schulstunden, intelligente Iteratoren und eine lokale Suche (jeweils 100 Versuche). Die Güte der Lösungen beschreibt die durchschnittliche Bewertung der generierten Pläne.

| Verfahren | Güte der Lösungen | Laufzeit |
|---|---|---|
| Standard-Quantoren | 52,17 | 35716 ms |
| optimierte Quantoren | 52,66 | 5084 ms |

Tabelle 4.1: Performancesteigerung durch optimierte Quantoren

### 4.4.5 Operatoren

In diesem Abschnitt geht es um die Auswertung der logischen Operatoren. In der definierten Sprache sind Formeln mit $\rightarrow$-, $\wedge$-, $\vee$-Verknüpfungen sowie Negationen möglich; zusätzlich ist es erlaubt, Klammern zu setzen. Dabei muss ein Knoten eines Operators nicht unbedingt zwei Kinder haben, denn die Produktionsregeln, für die die Knoten stehen, können auch einfach Zwischenregeln sein, welche für die korrekte Auswertungsreihenfolge benötigt werden. So ist der Vaterknoten der $\vee$-Verknüpfung immer eine $\rightarrow$-Verknüpfung, da die Produktionsregel `<OrExpression>` innerhalb der Regel `<ImpliesExpression>` steht. Der Baum für das Constraint `hard: true` sieht beispielsweise folgendermaßen aus.

```
Start
 Constraints
  HardConstraint
   Expression
    Quantor
     ImpliesExpression
      OrExpression
       AndExpression
        Negation
         UnaryExpression
          BooleanExp
```

An diesem Baum kann man gut die Bindungsstärke der Operatoren erkennen, die durch die Grammatik definiert ist. Die →- Verknüpfung bindet beispielsweise schwächer als die ∧-Verknüpfung, da sie im Parse-Tree immer oberhalb dieser steht, es sei denn sie wird durch einen Klammer-Ausdruck gebunden. In diesem Fall würde diese unter dem Knoten für `<BracketedExpression>` stehen, welcher wiederum unter dem Knoten für die ∧-Verknüpfung stehen kann.

Bei der Evaluierung muss ein Knoten, der für einen boolschen Operator steht, also zunächst testen, ob er über ein oder zwei Kinder verfügt. Im Falle nur eines Kindes steht er für eine Zwischenregel und ruft lediglich dessen `evaluate`-Methode auf. Ansonsten wertet er den rechten sowie den linken Kind-Knoten aus und gibt ein Ergebnis entsprechend seines Operators zurück.

Ein Knoten, der eine Negation repräsentiert, hat hierbei immer nur ein Kind, egal ob es sich um eine Zwischenregel handelt oder nicht. Daher wird bei der Erzeugung eines solchen Knotens darin gespeichert, ob es sich bei ihm um eine "echte" Negation (angegeben durch ein `!`) handelt. Dies wird durch den Parser beim Einlese-Vorgang bestimmt.

### 4.4.6 Relationen

Dieses Kapitel beschäftigt sich mit der Auswertung der Relationen als Grundbausteine der Grammatik und somit auch des Parse-Tree. Als solche werden diese relativ oft (gerade in Kombination mit darüber liegenden Quantoren) abgefragt und lohnen sich daher besonders zur Optimierung. Wie im letzten Kapitel wird hier zunächst die Grundversion der Implementierung beschrieben und später auf die angewandten Optimierungen eingegangen.

#### Implementierung der Relationen

Wie auch alle anderen Bausteine wird auch eine Relation im Parse-Tree durch einen Knoten repräsentiert. Innerhalb der Klasse für solchen Relationsknoten werden alle Belegungskombinationen gespeichert, die diese Relation enthält und für die deren Auswertung somit `true` ergeben soll. Wie man schon an einigen Beispielen gesehen hat, kann man bei der Abfrage einer Relation auch Stellen einfach frei lassen; diese sind dann bei der Abfrage schlicht nicht relevant. Die Auswertung einer Relation läuft dann konkret so ab, indem alle gespeicherten Belegungskombinationen durchgegangen werden und für jede überprüft wird, ob die zu testende Kombination dieser entspricht oder eine Untermenge davon ist. Das folgende Beispiel illustriert dieses Vorgehen.

**Beispiel 4.8**
Betrachtet wird der Stundenplan aus Abbildung 4.4. Die `LESSON`-Relation repräsentiert die darin enthaltenen Stundenplaneinträge und hat daher folgende Elemente:

| DAY | HOUR | TEACHER | GROUP | ROOM | SUBJECT |
|---|---|---|---|---|---|
| 0 | 0 | Müller | 1a | 102 | Deutsch |
| 0 | 2 | Meier | 1a | 101 | Sachkunde |
| 0 | 2 | Müller | 1b | 102 | Deutsch |

Tabelle 4.2: gespeicherte Belegungskombinationen

Beispielsweise wird bei der Auswertung von `LESSON(,,Müller,,,)` wird also erkannt, dass die Kombination `Müller` eine Untermenge der ersten Zeile in Tabelle 4.2 ist, so dass die Auswertung der Relation `true` ergibt.

Relationen kann man in drei verschiedene Klassen unterteilen:

- dynamische Relationen
- interne Relationen
- externe Relationen

Interne Relationen sind solche, die intern vom System generiert werden und nicht direkt vom Benutzer beeinflusst werden können. Die Relation `CLASSROOM`, welche angibt, ob ein Raum der Klassenraum einer Klasse ist, gehört zum Beispiel in diese Gruppe, da diese aus den vorgegebenen Stammdaten intern erzeugt wird. Interne Relationen werden als dynamisch bezeichnet, wenn Sie sich ändern, falls sich auch der Stundenplan ändert. In unserem System betrifft dies die `LESSON`-Relation. Diese muss für jeden Stundenplan neu initiiert werden, da sie sich von Plan zu Plan unterscheidet. Relationen, die nicht dynamisch sind, ändern sich nur bei einer Modifikation der Stammdaten und müssen nicht für jeden Stundenplan neu erstellt werden. Ein Beispiel hierfür ist wieder die obige Relation `CLASSROOM`. Für den Benutzer stehen vielfältige vorgefertigte Relationen zur Verfügung, die in Constraints eingebaut und so abgefragt werden können[7].

Da es jedoch von Fall zu Fall sein kann, dass zur Bildung von Constraints auch andere als die vorgefertigten Relationen benutzt werden müssen (und um das Programm so variabel wie möglich zu halten), wurden externe Relationen eingeführt. Diese können durch den Benutzer innerhalb einer Datei definiert werden, wofür dem Benutzer eine GUI zur Verfügung steht. Dort kann tabellenartig angegeben werden, welche Einträge diese Relation enthalten soll (ähnlich wie in Tabelle 4.2).
Ein Beispiel für eine externe Relation, könnte die Definition von

[7]Eine Übersicht über die vorhandenen Relationen im System steht im Constraint-Editor unter dem Punkt Relationen zur Verfügung

`VERTRAUENSLEHRER(teacher, group)` sein, welche angibt, ob ein Lehrer Vertrauenslehrer für eine Klasse ist.

### Optimierung der Relationen

Mit Hilfe eines Profilers wurde festgestellt, dass das System bei der Bewertung eines Plans die meiste Zeit mit dem Auswerten von Relationen beschäftigt ist. Daher ist die Optimierung dieser Auswertung ein lohnenswerter Ansatzpunkt und führte zu einer erheblich besseren Performanz des Generierungsvorgangs insgesamt.

Betrachtet man beispielsweise die `LESSON`-Relation in einem Szenario realistischer Problemgröße, so erhält man im Falle von 4 Stufen á 3 Klassen mit jeweils 25 Wochenstunden eine Gesamtanzahl von $3 * 4 * 25 = 300$ Unterrichtsstunden und somit 300 Einträge in dieser Relation. Im schlimmsten Fall müssen zur Auswertung also alle 300 Belegungskombinationen zum Testen durchlaufen werden. Dies ist z.B. immer der Fall, wenn die Relation zu `false` ausgewertet wird, weil die geforderte Abfrage keinem Eintrag in der Relation entspricht.

Das Ziel der Optimierunsansätze war es also, zu verhindern, dass bei jeder Auswertung einer Relation alle Kombinationen durchlaufen werden müssen, die in dieser gespeichert sind. Der erste Optimierungsversuch basierte auf einer Indexstruktur, die zusätzlich in der Relation gespeichert wird. In dieser Indexstruktur wird für jeden möglichen Eintrag in jeder Spalte die Zeile gespeichert, in welcher dieser vorkommt. Abbildung 4.5 soll das Konzept verdeutlichen. Hier kann man z.B. erkennen, das der Eintrag `Meyer` der Spalte `Lehrer` in den Zeilen 1,6 und 8 vorkommt. Wird nun eine Anfrage an die Relation ge-

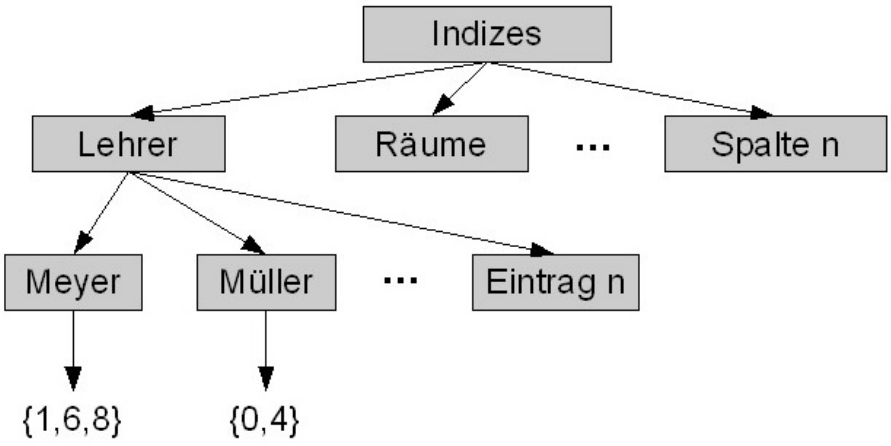


Abbildung 4.5: Speicherstruktur zur Abfrage von Relationen

stellt, wird zum ersten Eintrag dieser Anfrage (der nicht leergelassen wurde) die Menge der Zeilen erstellt, in denen dieser auftritt. Für jeden weiteren Eintrag wird diese Menge ebenfalls erstellt und sukzessive die Schnittmenge dieser Mengen gebildet. Ist diese Schnittmenge beim letzten Eintrag in der Anfrage nicht leer, so entspricht die Anfrage mindestens einem Eintrag in der Datenbank und kann somit mit `true` beantwortet werden, ansonsten mit `false`. Dies senkt die Komplexität einer Datenbankanfrage von $O(n)$ auf $O(e)$, wobei $e$ die Anzahl der verschiedenen Einträge einer Spalte und $n$ die Anzahl der Datenreihen ist. Die Verwendung solcher oder ähnlicher Indexstrukturen sind in Datenbanksystemen sehr weit verbreitet (siehe z.B. [BBK01]).

Die Grundiedee der zweiten Optimierungsstrategie ist, dass in einem HashSet jede mögliche Belegungskombination gespeichert ist, für deren Abfrage die Relation `true` zurück gibt und so die Vorteile dieser Datenstruktur in Bezug auf die Laufzeit genutzt werden können. Angenommen wir hätten eine `TEACHERCLASS`-Relation mit zwei Einträgen wie in Tabelle 4.3. Die Auswertung von `TEACHERCLASS(,1b)` würde beispielsweise `true`

| TEACHER | GROUP |
|---|---|
| Müller | 1a |
| Meyer | 1b |

Tabelle 4.3: gespeicherte Belegungskombinationen

ergeben. Um das Ergebnis zurückliefern zu können, müssen beide Einträge durchlaufen und jeweils getestet werden, ob sie die Konstante `1b` enthalten. In der optimierten Version werden nun **alle** Kombinationen, für die eine Abfrage `true` zurückgibt (auch die mit nicht belegten Stellen), in einem HashSet als Strings gespeichert. Dieses würde in diesem Beispiel folgendermaßen aussehen, wobei das Zeichen □ für einen nicht zugewiesenen Eintrag in einer Stelle der Abfrage steht.

| EINTRAG-NR. | WERT |
|---|---|
| 1 | Müller□ |
| 2 | □1a |
| 3 | Müller1a |
| 4 | Meyer□ |
| 4 | □1b |
| 5 | Meyer1b |

Tabelle 4.4: gespeichertes HashSet

Aus der Abfrage `TEACHERCLASS(,1b)` wird der String □`1b` erzeugt. Nun kann überprüft

werden, ob dieser String im HashSet gespeichert ist, welches mit einer Komplexität von $O(1)$ geschehen kann.

Zu beachten ist, dass beide vorgestellten Ansätze zusätzliche Zeit bei der Initiierung der Relationen benötigen. Beim ersten Ansatz muss die Indexstruktur erzeugt werden, während beim zweiten Ansatz das HashSet erstellt werden muss, was sich als besonders aufwändig herausgestellt hat. Dies bedeutet gerade für dynamische Relationen einen Nachteil, da diese für jeden neu zu bewertenden Plan wieder initiiert werden müssen. Für die Bewertung eines einzelnen Plans müssen jedoch so oft Relationen ausgewertet werden, dass beide Ansätze immer noch einen erheblichen Geschwindigkeitsvorteil gegenüber der nicht optimierten Variante aufweisen.

Eine weitere Optimierung kann zusätzlich bei der Initialisierung der Relationen vorgenommen werden, welches den Nachteil der vorherigen Techniken in Bezug auf dynamische Relationen wieder schmälert. Wie auch bei den Quantoren (siehe 4.4.4) wird hier die Tatsache ausgenutzt, dass zwei benachbarte Pläne, die nacheinander bewertet werden, zum größten Teil identisch sind. Für die Bewertung eines neuen Plans werden die Relationen also nicht komplett neu initialisiert, sondern nur die Einträge aktualisiert, die sich geändert haben. Hierzu kann die Relation auf die geänderten Einträge des Stundenplans zugreifen, analog zur Vorgehensweise bei den Quantoren (siehe Abschnitt 4.4.4).

Insgesamt wurde durch den zweiten Ansatz zusammen mit der verbesserten Aktualisierung der Relationen die größte Zeitersparnis erreicht (Details siehe Abschnitt 5.2.1).

### 4.4.7 Vergleichsausdrücke

Vergleichsausdrücke sind wie die Relationen Grundbausteine der verwendeten Grammatik und geben ebenfalls den Wahrheitswert `true` oder `false` zurück. Sie ermöglichen den Vergleich von Zahlen, Variablen und Funktionen, wobei `==`, `!=`, `>`, `>=`, `<`, und `<=` als Vergleichsoperatoren erlaubt sind. Ein Beispiel für die Anwendung eines solchen Ausdrucks ist der Teilausdruck `h >= 2`. Hier wird eine Variable, der ein Zahlenwert zugewiesen ist, mit einer float-Zahl verglichen.

Innerhalb einer *CompareExpression* ist dabei zusätzlich Arithmetik erlaubt. Es können dafür die Operatoren `+`, `-`, `*` und `/` verwendet und Klammern beliebig gesetzt werden. Die Produktionsregeln sind hierbei ähnlich aufgebaut wie die der Operatoren für logische Ausdrücke, wodurch die Bindungsstärke der Operatoren bei der Erzeugung des Parse-Tree beachtet wird (siehe 4.4.5). Wiederum gibt es für alle Operatoren eine entsprechende Klasse, die den Knoten für den Operator innerhalb des Parse-Tree repräsentiert. Die Auswertung dieser Operatorklassen erfolgt dann analog zu der Auswertung der logischen Operatoren, wobei es auch hier wieder Zwischenknoten gibt.

Die Auswertung der Knoten für Zahlen, Variablen und Funktionen wird in der Knoten-Klasse `ASTCompareExpression` vorgenommen, welche diese dann bezüglich des jeweiligen Operators evaluiert. Zu beachten ist, dass es sich bei den zu vergleichenden Variablen nicht um Zahlen handeln muss; es können auch Einträge eines anderen Typs eingesetzt werden (z.B. `teacher`). Die Klasse für eine Variable bietet dafür eine Möglichkeit herauszufinden, ob der Typ der Variablen eine Zahl ist oder nicht. Im letzten Fall werden lediglich die Zeichenketten der zugewiesenen Werte verglichen. Hierbei können nur die beiden Operatoren `=` sowie `!=` verwendet werden.

# Kapitel 5

# Die Benutzung des Systems

## 5.1 Die Benutzeroberfläche

Dieser Abschnitt beschäftigt sich mit der Entwicklung und dem Gebrauch der Benutzeroberfläche des Systems. Zwei Gesichtspunkte sind für deren Erstellung besonders zu beachten. Zum einen muss durch eine sorgfältige Analyse zunächst festgestellt werden, welche Funktionalitäten dem Benutzer angeboten werden sollen. Falls an dieser Stelle etwas vergessen wird, kann dies dazu führen, dass das System für die Praxis unbrauchbar wird. Wichtig ist auch die Gebrauchstauglichkeit (Usability) des Systems, denn entscheidend ist die subjektive Benutzerzufriedenheit, welche letztendlich den Ausschlag für die Akzeptanz des Systems gibt[1]. Die Benutzerfreundlichkeit wird vor allem durch eine einfache Erlernbarkeit der Funktionalitäten des Systems sowie durch eine schnelle und angenehme Ausführbarkeit der Aufgaben bestimmt.

### 5.1.1 Funktionen des Systems

Für die Entwicklung der Benutzeroberfläche müssen zunächst die Funktionen erkannt werden, die für den alltäglichen und praktischen Einsatz des Systems unerlässlich sind. Die primäre Funktionalität des Systems ist dabei eindeutig die Generierung von Stundenplänen. Gerade hier sollen dem Benutzer viele Möglichkeiten geboten werden, diese Generierung individuell zu gestalten, um ihm so ein hohes Maß an Flexibilität und Komfort zu bieten. So sollte es einerseits möglich sein, die Stundenplangenerierung komplett dem System zu überlassen und den erstellten Plan danach zu bearbeiten, andererseits muss auch die Möglichkeit bestehen, aktiv in den Generierungsprozess einzugreifen. Während die Erstellung läuft, ist es hierbei von Vorteil, sich jederzeit den bis jetzt besten gefundenen Plan anzeigen zu lassen, an welchem dann Manipulationen vorgenommen werden

[1] siehe zum Beispiel http://www.fit-fuer-usability.de

können. Das System sollte danach auf Basis des neuen Plans fortfahren können und versuchen, diesen zu verbessern.

Eine weitere sinnvolle Vorgehensweise ist, dass der Anwender zunächst damit beginnt, einen Stundenplan manuell zu erstellen und das System diesen dann vervollständigen lässt. Dabei sollen bestimmte Einträge festsetzbar sein, so dass diese bei der Optimierung nicht verschoben werden dürfen. Dies ist sinnvoll wenn zum Teil bereits genaue Vorstellungen über die Lage bestimmter Einträge vorhanden sind. Beispielsweise könnte man den Zeitpunkt des Schwimmunterrichtes auf diese Weise vorher fixieren, falls das benötigte Schwimmbad der Schule nur genau dann zur Verfügung steht. So kann die Generierungs-Funkion sowohl vollautomatisch als auch zur Unterstüzung einer manuellen Erstellung verwendet werden.

Für den alltäglichen Gebrauch muss weiterhin eine Funktionalität zum Drucken der erstellten Stundenpläne vorhanden sein. Auch hier wollen wir ein breites Spektrum an Möglichkeiten zur maßgeschneiderten Ausgabe bieten. Neben einer umfassenden Übersicht sollte man auch die Stundenpläne einzelner Klassen oder Lehrer sowie Raumpläne ausdrucken können.

Um überhaupt die Rahmenbedingungen für eine Stundenplangenerierung zu ermöglichen, müssen die der Schule zu Grunde liegenden Daten eingegeben und komfortabel editierbar sein. Wenn der Schule zum Beispiel eine neue Lehrkraft zugeteilt wird, sollte diese neue Situation leicht und schnell erfassbar sein. Bei diesen zu Grunde liegenden Daten, im Folgenden auch Stammdaten genannt, handelt es sich um die folgenden Informationen:

- Grundsätzliche Angaben zum Stundenplan (Wie viele Stunden bzw. Tage gibt es?)
- Informationen zu Jahrgangsstufen mit deren zugeordneten Unterrichtsplänen
- Daten von unterrichtenden Lehrern
- Angabe der Fächer
- Informationen zu Klassen und zusammengesetzten Gruppen
- Daten der Räume und der Raumtypen

Weiter muss der Benutzer in der Lage sein, die Constraints für das System anzupassen, was so einfach wie möglich umsetzbar sein sollte. Mehrere Stundenpläne sollten definierbar und speicherbar sein, um bei der Erstellung des endgültigen Plans Alternativen abwägen und unabhängig voneinander auswählen zu können. Alle einer Schule zugeordneten Daten, also die Stammdaten, die Constraints und die gespeicherten Stundenpläne sollten hierbei jeweils in Projekten verwaltet werden, damit auch mehrere komplett verschiedene Ausgangs-Situationen modelliert werden können (z.B. für jedes Schuljahr ein separates Projekt), wobei es von Vorteil ist, diese Projekte aus Gründen der Austauschbarkeit auch exportieren und wieder importieren zu können.

Anhand dieser Anforderungen wurde eine grafische Oberfläche entwickelt, welche diese Funktionalitäten in einer möglichst komfortablen Weise anbietet. Zur Übersicht folgt eine kurze Liste der Programmfunktionen, die sich damit für den Nutzer ergeben und den Bedürfnissen der meisten Schulen entsprechen sollten. Für eine detaillierte Beschreibung der Programmfunktionen sei auf das Handbuch verwiesen.

- Stammdatenverwaltung bestehend aus
  - Verwaltung der Jahrgangsstufen
  - Management der Fächer
  - Bearbeitung der Klassen
  - Organisation der Räume und Raumtypen
  - Klassenübergreifende Zuordnung in Gruppen
- verschiedene Suchverfahren zur Stundenplangenerierung auswählbar
- manuelle Verteilung der Stunden und Zuordnung der Räume/Lehrer jederzeit möglich
- Verwaltung beliebig vieler Stundenpläne in einem Projekt möglich
- Ansicht eines Stundenplans (jeweils einzeln oder gesamt)
  - nach Klassen
  - nach Lehrern
  - nach Räumen
- Festsetzung von Stundenplaneinträgen
- Vervollständigung eines bereits teilweise spezifizierten Planes
- Grafischer Editor zum Bearbeiten der Constraints und der darin verwendeten Relationen
- Batchmodus zum Generieren mehrerer Pläne nacheinander
- Export eines Projektes im zip-Format
- Import eines Projektes im zip-Format
- Export eines Stundenplanes im HTML-Format
- Individuell konfigurierbare Druck-Ansicht

### 5.1.2 Überblick über die grafische Oberfläche

Zunächst soll dem Leser in diesem Abschnitt ein Eindruck von der Oberfläche vermittelt werden, indem deren Bedienung anhand der wichtigsten Funktionen erläutert wird. Im darauffolgenden Abschnitt erfolgt eine Analyse des Systems anhand der Kriterien der Gebrauchstauglichkeit.

Abbildung 5.1, in welcher ein generierter Stundenplan zu sehen ist, zeigt die Oberfläche des Systems. Aus Übersichtsgründen wird in der folgenden Darstellung lediglich ein kleiner Abschnitt des Stundenplans angezeigt; im Normalfall ist in dieser Ansicht ein Großteil des aktuellen Plans zu sehen, wodurch das Arbeiten sehr übersichtlich wird.

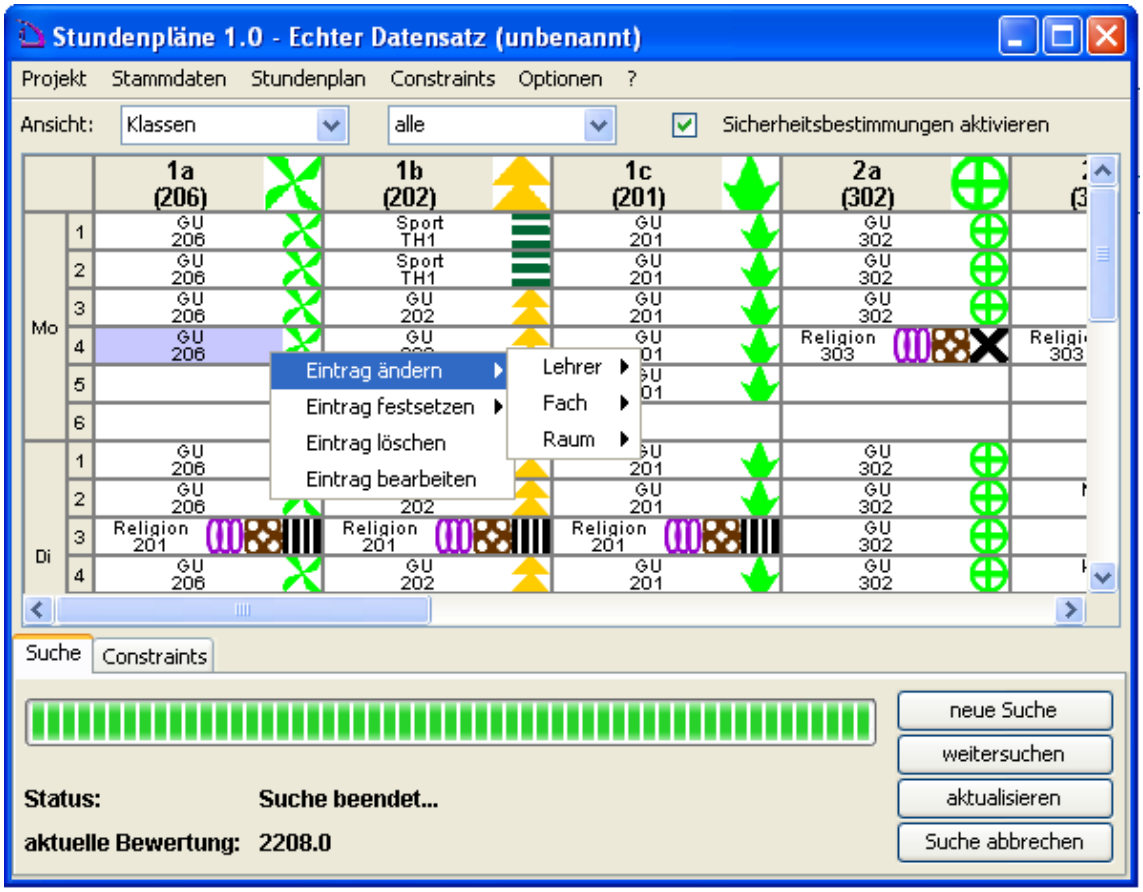


Abbildung 5.1: Bearbeitung des Stundenplans

Neben der automatischen Generierung ist die manuelle Bearbeitung des Stundenplans die wichtigste Funktion des Systems. Der Benutzer wird die meiste Zeit damit verbringen, den Plan zu verändern und ihn seinen Wünschen anzupassen. Dies sollte also möglichst angenehm und intuitiv möglich sein. Daher lassen sich die Stundenplan-Einträge einfach von einer Zeiteinheit zur anderen schieben. Falls beide Stellen belegt sind, so werden die Einträge getauscht, wobei im Falle einer Bandstunde der Unterricht der gesamten Stufe gleichzeitig verschoben wird.

Ferner existiert ein sogenannter **Sicherheitsmodus**, welcher über die Checkbox **Sicherheitsbestimmungen aktivieren** eingeschaltet werden kann. Ist dieser aktiv, kann man die Einträge nur verschieben, wenn der Plan danach auch weiterhin gültig ist. Es ist dann nicht mehr möglich, Einträge so zu verschieben, dass ein Lehrer zwei Klassen gleichzeitig

unterrichten müsste oder dass in einem Raum zwei Klassen gleichzeitig Unterricht hätten. Kann ein Eintrag gültig an einen anderen Zeitslot verschoben werden, wird dieser grün hinterlegt; ist dies aus oben genannten Gründen nicht möglich, so wird dieser rot markiert. Bestehende Einträge können über ein Kontextmenü geändert, gelöscht oder festgesetzt werden. Öffnet man dieses an einer leeren Position, hat man die Möglichkeit einen neuen Unterrichts-Eintrag anzulegen.

Über die Option **Stammdaten** im Navigationsmenü können die der Schule zu Grunde liegenden Daten bearbeitet werden. In Abbildung 5.2 ist die Oberfläche zur Bearbeitung der Stammdaten zu sehen.


Abbildung 5.2: Bearbeitung der Stammdaten

Zu erkennen sind hier die einzelnen Bereiche der Stammdaten, welche über Registrierkarten erreichbar sind. Jeder Bereich zeigt eine Tabelle mit den existierenden Daten an, wobei es möglich ist, diese Daten zu bearbeiten, zu löschen oder neue Daten anzulegen. Eine Änderung wird dabei sofort propagiert und andere Daten, die von dieser abhängen, ändern sich entsprechend. Wird beispielsweise ein Fach gelöscht, so wird dieses bei jedem Lehrer, der es vorher unterrichtet hat, ebenfalls entfernt. Um die geänderten Stammdaten komplett zu übernehmen, muss auf **speichern** geklickt werden; falls der Benutzer einzelne Daten ändert und lediglich auf **beenden** klickt, erscheint ein Dialog, der ihn hierauf hinweist und ihm die Möglichkeit bietet, seine Änderungen zu speichern oder eben nicht.

Die dritte wichtige Grundfunktionalität der grafischen Oberfläche ist die Verwaltung der Constraints. Ein Klick auf den Menüpunkt **Constraints** öffnet die in Abbildung 5.3 gezeigte Übersicht, von welcher eine weitere Bearbeitung der Constraints erfolgen kann.


Abbildung 5.3: Übersicht über die Constraints

Hier wird eine Liste mit allen aktivierten Constraints angezeigt, die das System bei der Stundenplan-Generierung berücksichtigen soll. Der Benutzer kann jedes Constraint löschen oder sich dessen Details anzeigen lassen. Innerhalb dieser Details sieht man unter anderem den zugeordneten Ausdruck, welcher dort bearbeitet werden kann. Desweiteren besteht die Möglichkeit, ein neues Constraint anzulegen, wobei sich hier ein Fenster öffnet, in dem man dessen Grunddaten angeben kann. Erzeugt man ein weiches Constraint, so kann das Penalty über einen Schieberegler bestimmt werden. Weiterhin kann man hier zwischen zwei verschiedenen Assistenten zur Erzeugung von Constraints wählen. Ein Assistent stellt hierbei häufig gebräuchliche Constraints in einer Liste zur Verfügung, welche einfach ausgewählt und übernommen werden können. Zum anderen kann man mit Hilfe des Constraint-Editors eigene Ausdrücke für diese formulieren. Dies sollte aber nur in seltenen Fällen nötig sein und von einem Experten durchgeführt werden, der die Sprache zur Formulierung der Constraints beherrscht. Dabei werden dem Benutzer einige Hilfestellungen geboten; so können einfache Konstrukte (z.B. Quantoren oder Und-Verknüpfungen) direkt über Schaltflächen erzeugt werden, so dass man sie nicht selbst eingeben muss. Relationen und Funktionen können aus einer Liste gewählt werden, um diese in den bestehenden Ausdruck einzufügen. Eine ähnliche Auswahlmöglichkeit exisitert für Konstanten, welche nach Angabe eines Typs aufgelistet (z.B. alle Lehrer oder Räume) und integriert werden können.

Wählt man in der Übersicht aus Abbildung 5.3 die Option **Relationen** bzw. **Funktionen**, so werden alle vorhandenen Relationen bzw. Funktionen angezeigt. Der Benutzer kann diese löschen, bearbeiten oder neue erzeugen. Möchte man beispielsweise eine neue Relation anlegen, öffnet sich ein Fenster in dem der Name der Relation und ihre Stellen angegeben werden können, worauf in einem zweiten Schritt alle Belegungen eingepflegt werden, für die die Relation wahr sein soll.

### 5.1.3 Gebrauchstauglichkeit des Systems

Die Tauglichkeit eines Systems für die alltägliche Anwendung ist von entscheidender Bedeutung für dessen Akzeptanz. Im Bereich der Usability wurden daher verschiedene Kriterien entwickelt (siehe [Sch05]), die für die Benutzung eines Systems von essentieller Bedeutung sind. Im Folgenden wollen wir unser System hinsichtlich dieser Kriterien analysieren.

**Vorhersagbarkeit** Dieser Punkt bezieht sich auf zwei verschiedene Teilaspekte. Zum einen sollte der Benutzer vor der Ausführung eine Vorstellung davon haben, was passiert, wenn er eine bestimmte Aktion vornimmt. Zum anderen sollte er auch nur die in diesem Moment durchführbaren Aktionen zu Gesicht bekommen und auswählen können. In unserem System sind die einzelnen Aktionen sinnvoll und intuitiv benannt, zum Beispiel kann ein Stundenplan **verbessert** oder **vervollständigt** werden. Optionen, die zu einem Zeitpunkt nicht durchführbar sind, werden immer deaktiviert und sind so nicht ausführbar. Beispielsweise wird während der Stundenplangenerierung die komplette Navigationsleiste deaktiviert, damit beim Benutzer nicht der Eindruck entsteht, dass er während der Suche andere Aktionen durchführen könnte.

**Synthetisierbarkeit** Dieser Aspekt beschreibt, inwiefern der Benutzer die Auswirkungen seiner Aktionen auf den Systemzustand beurteilen kann, was in unserem System immer gegeben ist. Ändert man den Stundenplan, die Constraints oder die Stammdaten, so werden diese Änderungen sofort sichtbar.

**Bekanntheit** Bekanntheit beschreibt, inwiefern der Benutzer Wissen aus der realen Welt bzw. seinem Arbeitsbereich auf das vorhandene System übertragen kann. Daher wurde die Gesamtansicht des Stundenplans der bisher in der Schule praktizierten Methode (in Form eines Steckbrettes) nachempfunden. Der durch das System generierte Stundenplan wird genau in dieser Ansicht dargestellt, so dass keine Umstellung für die Lehrkräfte nötig ist. Für die Kennzeichnung der verschiedenen Lehrer wurden außerdem schon vorher in der betroffenen Schule Symbole verwendet. Diese wurden digitalisiert

und für die Darstellung der Lehrer in unserem System eingebunden, wodurch wir einen guten ersten Eindruck beim Nutzer zu erzeugen konnten.

**Generalisierbarkeit** Der Benutzer sollte das Wissen aus anderen Programmen direkt auf unser System übertragen können. Dieser Aspekt wird als Generalisierbarkeit bezeichnet. Ähnlich wie beim Kriterium der Bekanntheit soll hierdurch ermöglicht werden, dass der Benutzer sich schnell im System zurechtfindet. Daher ist dieses System wie ein Office-ähnliches Programm aufgebaut und weist ein dafür typisches Look-and-Feel auf. Innerhalb einer Navigationsleiste befinden sich die wichtigsten Funktionsgruppen und ein Stundenplan kann über ein Kontextmenü direkt bearbeitet werden.

**Konsistenz** Konsistenz bedeutet, dass das System sich bei ähnlichen Aufgaben auch ähnlich verhalten soll. Da dieser Aspekt die Eingewöhnungszeit in das Programm deutlich verkürzen soll, wurde auch hierauf großen Wert gelegt. So sind beispielsweise die Stammdaten alle in ähnlicher Weise erreichbar und editierbar, indem die einzelnen Bereiche alle auf die selbe Art in Tabellenform dargestellt sind mit den gleichen Optionen zum Löschen, Einfügen und Bearbeiren der Einträge.

**Multithreading** Multithreading beschreibt, inwiefern der Benutzer mehrere Aufgaben gleichzeitig bearbeiten kann. Grundsätzlich ist dies im entwickelten System an vielen Stellen möglich (zum Beispiel können gleichzeitig die Stammdaten und die Constraints bearbeitet werden), jedoch verbessert dieser Punkt die Benutzung des Systems nicht wesentlich und spielt in diesem Kontext daher keine große Rolle.

**Migrationsmöglichkeiten** Hiermit ist gemeint, dass der Benutzer entscheiden kann, ob er bestimmte Aufgaben selbst erledigen möchte oder ob das System dies für ihn übernimmt. Unser Programm bietet vielfältige Möglichkeiten, sowohl zur manuellen als auch komplett automatischen Generierung eines Stundenplanes. Wie bereits erwähnt bietet sich dem Benutzer grundsätzlich auch ein semiautomatischer Weg, indem er beispielsweise Fächer schon vorher verteilt und den Plan dann vom System vervollständigen lässt. Zwischenergebnisse können jederzeit angezeigt und bearbeitet werden.

**Dialog-Initiative** Mit Dialog-Initiative ist gemeint, dass die Initiative bei der Verwendung des Programms vom Benutzer ausgehen sollte. Im besten Fall sollte das System dem Benutzer also nicht immer genau vorgeben, welche Aktion er auszuführen hat. Grundsätzlich wurde bei der Entwicklung des Systems darauf geachtet, den Benutzer bei seiner Arbeit nicht einzuschränken und ihm so die Initiative bei der Ausführung seiner Aufgaben zu überlassen.

**Ersetzbarkeit** Dem Anwender sollten sich mehrere verschiedene Wege anbieten, ein und die selbe Aufgabe zu erledigen. Auf diese Weise kann jeder Benutzer das System so bedienen wie es seinen Vorlieben entspricht. In unserem System sind wichtige Operationen sowohl über die Navigationsleiste als auch über Schaltflächen anwählbar. Beispielsweise kann eine Suche direkt in der Stundenplanansicht oder auch über das Navigationsmenü gestartet werden. Desweiteren kann man einen Stundenplan-Eintrag direkt über das Kontextmenü ändern oder den Dialog "Eintrag bearbeiten" zu diesem Zweck aufrufen.

**Konfigurierbarkeit** Konfigurierbarkeit beschreibt, inwiefern der Benutzer die Oberfläche des Systems seinen Wünschen anpassen kann. Dafür bieten wir die Möglichkeit, verschiedene "Look & Feels" einzustellen. Außerdem können für die Symbole der Lehrer beliebige Bilder eingefügt werden.

**Beobachtbarkeit** Beobachtbarkeit bedeutet, dass der Benutzer viele Möglichkeiten haben sollte, den Zustand des Systems zu erforschen. Dies ist besonders in diesem Kontext wichtig, da genau überprüft werden muss, ob der erstellte Stundeplan auch allen Anforderungen gerecht wird. Besonders hilfreich ist hier die Möglichkeit, sich die Eigenschaften des Stundenplans anzuschauen. Hier werden u.a. Informationen über die Anzahl der Stunden von Klassen, Lehrern und Räumen angezeigt. So kann auf einen Blick bestimmt werden ob auch jede Klasse und jeder Lehrer an genügend Unterrichtsstunden teilnimmt. Desweiteren erlauben die verschiedenen Ansichten eines Plans[2] dessen Erforschung aus mehreren Perspektiven.

**Recoverbility** Dieser Aspekt beschreibt die Möglichkeit des Benutzers, Fehler rückgängig zu machen. Wenn ein Benutzer beispielsweise versehentlich einen Lehrer aus den Stammdaten löscht, kann er den dafür vorgesehenen Editor einfach ohne einen Speichervorgang schließen, so dass dies keine Auswirkungen hat. Auch kann ein Projekt jederzeit für eine Datensicherung exportiert und später wieder eingespielt werden.

**Ansprechbarkeit** Die Antwortzeiten des Systems sollten im Erwartungsbereich des Benutzers liegen. Dies ist besonders von Bedeutung, wenn die Generierung eines Stundenplans gestartet wird, was einige Zeit benötigt und vom Benutzer auf keinen Fall als Systemabsturz fehlinterpretiert werden darf. Daher zeigt ein Balken den Fortschritt der Suche, die Bewertung des aktuell besten Stundenplans sowie eine Abschätzung der noch für die Suche benötigten Zeit an.

[2] Wie bereits erwähnt kann der Plan geordnet nach Klassen, Lehrern und Räumen (jeweils zusammen oder einzeln) angezeigt werden

**Aufgabenanpassung** Das System muss alle geforderten Aufgaben unterstützen. Dieser Aspekt wurde bereits in Abschnitt 5.1.1 diskutiert.

## 5.2 Evaluierung

Im Verlauf des Projektes wurde relativ früh klar, dass das benutzte System zur Formulierung von Constraints zwar hochgradig variabel, aber leider dadurch auch entsprechend unperformant ist. Daher wurden verschiedene Optimierungen durchgeführt, welche vorher an den entsprechenden Stellen schon vorgestellt wurden. Diese sollen im Folgenden in Bezug auf ihren Nutzen verglichen werden. Danach folgt dann ein Vergleich der Ergebnisse der verschiedenen Suchverfahren und eine Evaluierung aus Benutzersicht.

### 5.2.1 Optimierungen

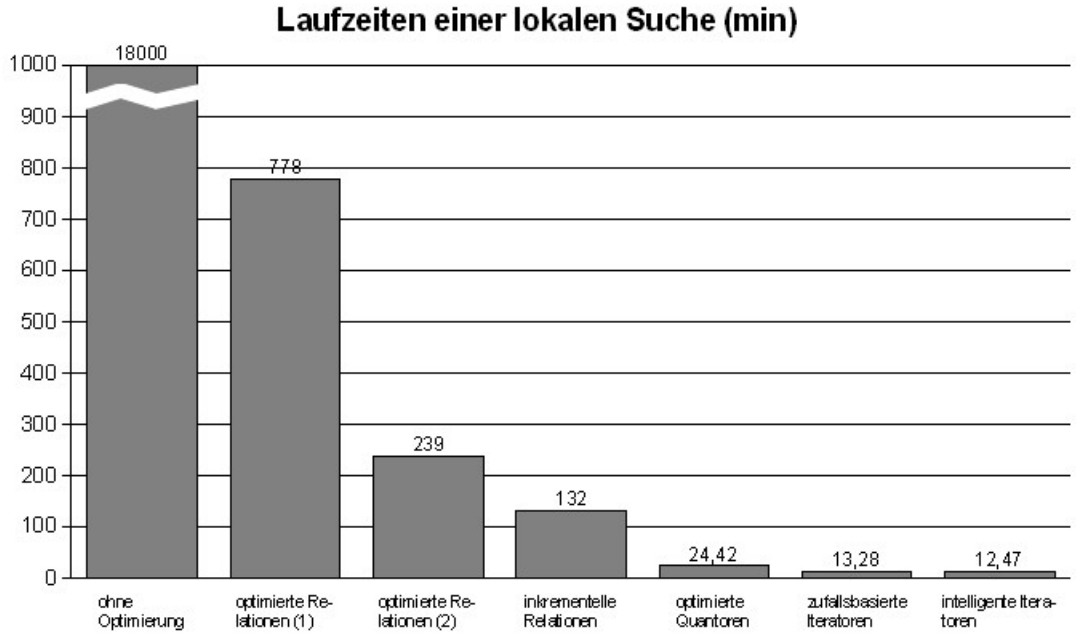


Abbildung 5.4: Verbesserungen der Laufzeit durch Optimierungen

Eine Analyse der Laufzeit eines Suchvorganges wurde sowohl mit gewöhnlichen Analysemaßnahmen sowie mit aufwändigeren *Profilern*[3] durchgeführt. Nicht verwunderlich war deren Ergebnis, nämlich dass die Bewertung der Stundenpläne den mit Abstand größten Teil der Laufzeit einer Suche ausmacht. Deshalb wurde dieser Vorgang besonders optimiert, wobei eine große Verbesserung erzielt werden konnte. Die einzelnen Maßnahmen zur Optimierung wurden schon detailliert an den entsprechenden Stellen erläutert (in Bezug auf die Relationen siehe Kap. 4.4.6, für die Quantoren siehe 4.4.4, zu den Iteratoren siehe 3.2); an dieser Stelle werden diese noch einmal kurz miteinander in Beziehung

[3] z.B. dem YourKit Java Profiler (siehe http://www.yourkit.com)

gesetzt. Abbildung 5.4 zeigt die durchgeführten Optimierungen in chronologischer Reihenfolge. Diesen Messungen liegt jeweils der Mittelwert aus mehreren lokalen Suchen auf einem Problem realistischer Größe (12 Klassen mit je 23-27 Stunden, 15 Constraints) zu Grunde. Tabelle 5.1 zeigt, dass die ersten Maßnahmen, die ergriffen wurden, darauf abzielten, die Bewertungs-Zeit an sich zu reduzieren, während die Optimierungen der Iteratoren deren Anzahl reduzieren sollten. Insgesamt konnte eine Suche zeitlich um mehr als den Faktor 1.000 verbessert werden. Dabei gab es nur bei der Optimierung der Quantoren einen (sehr geringen) Qualitätsverlust in Bezug auf die Güte der Lösungen (siehe Tabelle 4.1); alle anderen Optimierungsmaßnahmen führten zu keinerlei Verschlechterung der Ergebnisse.

| | Zeit pro Bewertung | Suchschritte | Laufzeit |
|---|---|---|---|
| ohne Optimierung | 10.311 ms | 104.712 | $\sim$ 300 h |
| optimierte Relationen (1) | 412 ms | 104.712 | 12 h 58 min |
| optimierte Relationen (2) | 103 ms | 104.712 | 3 h 59 min |
| inkrementelle Relationen | 76 ms | 104.712 | 2 h 12 min |
| optimierte Quantoren | 14 ms | 104.712 | 24 min 25 s |
| zufallsbasierte Iteratoren | 14 ms | 56.943 | 13 min 17 s |
| intelligente Iteratoren | 14 ms | 53.517 | 12 min 28 s |

Tabelle 5.1: Vergleich der Optimierungen

### 5.2.2 Vergleich der Algorithmen

Für die automatische Generierung der Stundenpläne wurden verschiedene Algorithmen implementiert. Diese weisen verschiedene Vor- und Nachteile auf (siehe dazu Kap. 3.5), woraus eine unterschiedliche Qualität der enstehenden Pläne sowie verschiedene Laufzeiten resultieren. Ein Vergleich aller verwendeten Verfahren wurde auf jeweils vier verschiedenen Datensets durchgeführt. Diese repräsentieren jeweils eine Schule, welche den Unterricht ein-, zwei-, drei oder vierzügig gestaltet, wobei dies jeweils der Anzahl der Klassen in einer Jahrgangsstufe entspricht. Dafür wurde jeweils ein handelsübliches Notebook mit einer Prozessorleistung von 1,73 GHz und einem Hauptspeicher mit 512 MB benutzt. Die Abbildungen 5.5 und 5.6 zeigen die Ergbenisse.

Auf den ersten Blick erkennt man hier, dass die Bewertung eines Planes schlechter wird, je größer die Anzahl der Klassen insgesamt ist. Dies ist jedoch nicht verwunderlich, da hieraus auch eine größere Menge an Unterrichtsstunden resultiert, die gegen die gestellten Bedingungen verstoßen können. Insgesamt fällt auf, dass die Algorithmen in Bezug

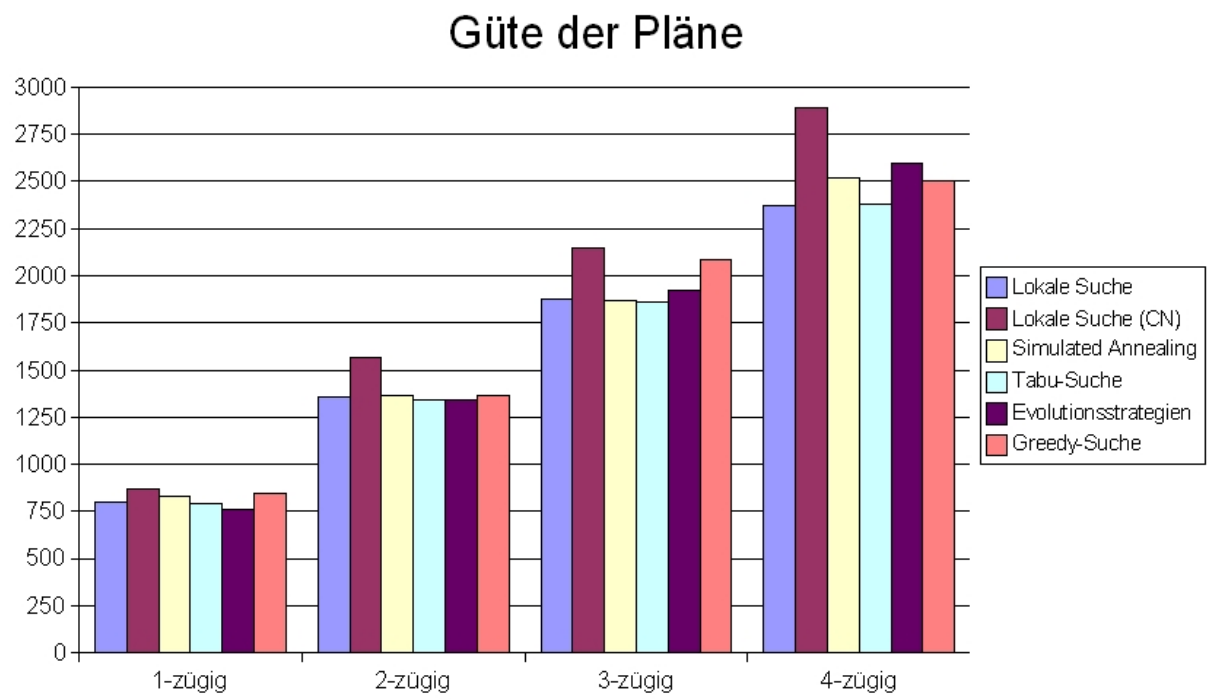


Abbildung 5.5: Qualität der gefundenen Stundenpläne

auf die Güte der Lösungen gerade bei kleinen Datensets sehr nahe beieinander liegen. Lediglich die Lokale Suche mit kompletter Durchsuchung der Nachbarschaft (Lokale Suche CN) liefert hier Lösungen, die durchweg etwas schlechter als die anderen Algorithmen sind, was vermutlich darauf zurückzuführen ist, dass dieses Verfahren schneller in einem lokalen Minimum endet. Dabei benötigt dieser Algorithmus zusätzlich mehr Zeit als die Lokale Suche im Standard-Verfahren, welche sowohl in der Laufzeit als auch in der Qualität sehr gute Ergebnisse liefern konnte.

Die beiden anderen nachbarschaftsbasierten Verfahren, also die Tabu-Suche und das Simulated Annealing, liegen hierbei nahe beeinander und etwa auf dem Niveau der Lokalen Suche. Zeitlich gesehen schneidet die Tabu-Suche etwas schlechter als die anderen Methoden ab, da diese mehrere lokale Optima betrachtet. Aufgrund dessen kann man in den ersten beiden Datensätzen minimale Vorteile der Tabu-Suche gegenüber der Lokalen Suche feststellen, wohingegen erstere bei großen Datensets keine Vorteile mehr bietet. Die liegt wohl daran, dass die Größe der Tabuliste für diesen Fall zu klein eingestellt ist und die Suche nicht mehr aus dem ersten betrachteten Minimum herauslaufen kann.

In kleinen Datenmengen schneidet die Evolutions-Suche am besten ab. Die Verschlechterung auf größeren Informations-Mengen ist auf die limitierte Anzahl der Suchschritte zurückzuführen, welche in diesem Fall nicht mehr ausreichend ist. Zeitlich gesehen liegt diese Suche immer weit über allen anderen Verfahren und kann so nur verwendet werden,

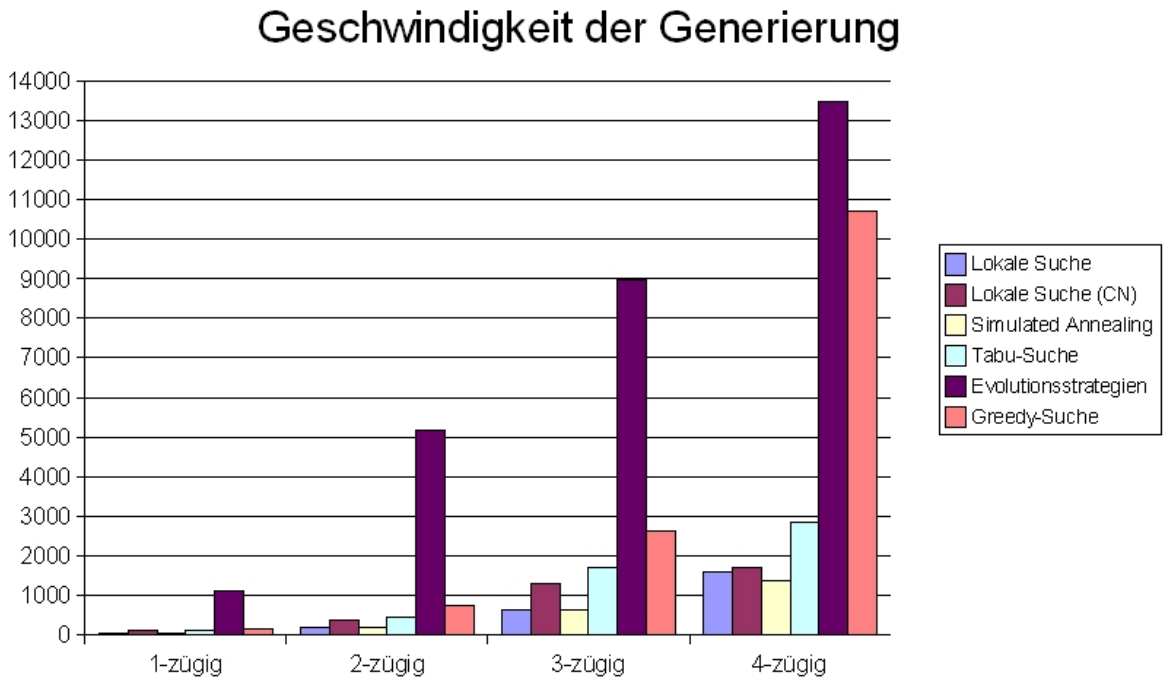


Abbildung 5.6: Geschwindigkeit bei der Erzeugung

| | 1-zügig | | 2-zügig | | 3-zügig | | 4-zügig | |
|---|---|---|---|---|---|---|---|---|
| Lokale Suche | 800 | 36 s | 1354 | 3 min 8 s | 1876 | 10 min 29 s | 2374 | 26 min 15 s |
| Lokale Suche (CN) | 867 | 1 min 51 s | 1568 | 6 min 25 s | 2145 | 21 min 27 s | 2889 | 28 min 27 s |
| Simulated Annealing | 832 | 33 s | 1361 | 3 min 3 s | 1866 | 10 min 48 s | 2518 | 22 min 50 s |
| Tabu-Suche | 791 | 2 min 4 s | 1341 | 7 min 29 s | 1862 | 27 min 15 s | 2382 | 47 min 14 s |
| Evolutions-Suche | 761 | 18 min 38 s | 1340 | 86 min 2 s | 1919 | 149 min 46 s | 2598 | 224 min 46 s |
| Greedy-Suche | 844 | 2 min 25 s | 1364 | 12 min 36 s | 2086 | 44 min 3 s | 2500 | 178 min 10 s |

Tabelle 5.2: Vergleich der Lösungsverfahren

wenn sehr viel Rechenzeit zur Verfügung steht. Das auf einem komplett anderen Ansatz basierende Greedy-Verfahren liefert in diesem Kontext ähnlich gute Ergebnisse wie die nachbarschaftsbasierten Verfahren, jedoch verhält sich dessen Laufzeit exponentiell zur Problemgröße und wird dadurch für größere Datensets nicht mehr anwendbar.

Insgesamt stellt die einfache Version der Lokalen Suche den besten Trade-Off zwischen Ausführungsgeschwindigkeit und der Qualität der Lösungen dar. Tabelle 5.2 stellt die Ergebnisse noch einmal exakt dar.

### 5.2.3 Der Nutzen für die Schule

Wie in einer typischen Bildungseinrichtung muss auch in dieser Schule für jedes Halbjahr ein neuer Stundenplan erstellt werden. Dieser wurde bislang auf einem Blatt Papier mit „Bleistift und Radiergummi“ erstellt und anschließend auf eine große Übersichtstafel in Form eines Steckbrettes im Lehrerzimmer übertragen. Ständig mussten Einträge wegradiert und wieder neu geschrieben werden, was sehr lästig war und die Lesbarkeit des Plans beeinträchtigte. Aus Gesprächen mit Frau Topaloglu[4] ging hervor, dass eine solche Erstellung von Hand bislang ungefähr eine Woche benötigte. Für die alltägliche Verwendung und Verteilung der Pläne wurden diese dann in Excel-Tabellen eingetragen und ausgedruckt, was wiederum einige Zeit in Anspruch nahm. Durch die Verwendung unseres Systems soll die benötigte Zeit für die Anfertigung eines Stundeplans erheblich verkürzt werden.

Eine erste Vorstellung und Vorführung unseres Systems in der betreffenden Grundschule verlief dabei sehr positiv. Durch die der bisherigen Form angepassten Stundenplan-Ansicht fand sich Frau Topaloglu auf Anhieb gut zurecht und ein guter erster Eindruck konnte erreicht werden. Besonders die bekannte Anordnung der Stundenplan-Einträge und die Darstellung der Lehrer durch die gewohnten Symbole sorgten hier für eine schnelle Einarbeitungszeit.

Die Funktionalität, einen Plan komplett manuell erstellen zu können, stellte sich bereits als großer Verteil heraus. Während eine Änderung im Stundenplan vorher durch das Entfernen und erneute Einfügen des entsprechenden Eintrags an anderer Position auf dem Papier bzw. im Steckbrett vorgenommen werden musste, kann der Plan nun durch kompfortables Hin- und Herschieben der Einträge editiert werden. Zusätzlichen Komfort bietet hier die Druckfunktion, da ein erstellter Plan mit unserem System nun direkt ausgedruckt werden kann und kein fehleranfälliges und langwieriges Übertragen der Daten in eine Excel-Tabelle mehr nötig ist. Die diversen Druckansichten können hierbei flexibel für Klassen-, Lehrer- und Raumpläne genutzt werden.

Auch die Hauptfunktion des Programmes, nämlich die automatische Generierung eines Stundenplanes anhand der gegebenen Stammdaten, lieferte qualitativ zufriedenstellende Ergebnisse. Die durch die Optimierungen erreichte Laufzeit von ca. 15 Minuten (für eine in diesem Fall geltende Problemgröße) wurde hier nicht als störend und für diesen Vorgang durchaus angemessen empfunden. Selbst bei mehrmaliger Ausführung mit anschließender manueller Anpassung bleibt die Bearbeitungszeit zur Anfertigung des engültigen Planes weit hinter der bisherigen Arbeitszeit von etwa einer Woche zurück. Es stellte sich ferner heraus, dass von Seiten der Schule nicht erwartet wird, dass der erstellte Plan bei erstmaliger Generierung sofort „perfekt“ ist. Von einer späteren manuellen Bearbeitung der resultierenden Pläne wurde von vornherein ausgegangen, welche unser Programm in vielfältiger Form zur Verfügung stellt.

[4] Die mit der Stundenplanerstellung beauftragte Lehrkraft

Wie erwartet fehlt Frau Topaloglu aufgrund ihrer zusätzlichen Tätigkeit als Klassenlehrerin die Zeit, sich mit der zur Formulierung der Constraints benötigten Sprache vertraut zu machen. Als Grund für diese schwierige Einarbeitung ist wohl zu sehen, dass man nicht davon ausgehen kann, dass der normale Benutzer über Kenntnisse der Prädikatenlogik verfügt. Da die meisten gebräuchlichen Bedingungen bereits als Standard-Constraints vorformuliert und im Editor zur Verfügung stehen, kann eine eingeschränkte Anpassung des Systems hier trotzdem relativ einfach geschehen. An Schulen, in denen eine Person mehr Zeit in die Stundenplanerstellung investieren kann oder enstsprechende Vorkenntnise mitbringt, können dann alle Aspekte des Systems vollständig genutzt werden. Einen sehr großen Vorteil bietet die Verwendung von Constraints jedoch nicht nur dem Nutzer, sondern auch den Entwicklern des Systems. Auf diese Weise kann das Programm durch einfache Umformulierung der Constraints auf jede andere Schule angepasst und so als Speziallösung eingesetzt werden. Beispielsweise erläuterte Frau Topaloglu während der Vorstellung der Stundenplangenerierung die zusätzliche Bedingung, dass der Unterricht jeden Montag in jeder Klasse beim Klassenlehrer beginnen sollte. Diese konnte innerhalb kürzester Zeit als Coinstraint formuliert und im System integriert werden.

Insgesamt kann festgestellt werden, dass das System in der betreffenden Schule tatsächlich eingesetzt wird und diese daraus mit hoher Wahrscheinlichkeit einen großen Nutzen ziehen kann. Im Vergleich zu anderen bisher getesteten Programmen ist die einfache Bedienbarkeit ein entscheidender und sogar kostenbestimmender Faktor. So mussten für bisherige Testläufe aufgrund der weniger intuitiven Benutzbarkeit kostspielige Schulungen wahrgenommen werden, welche die Effektivität in der alltäglichen Benutzung jedoch kaum verbessern konnten. Andere Schulen standen vor ähnlichen Hindernissen. Aus den genannten Gründen stellt das von uns erstellte Programm eine sinnvolle Alternative für die betreffende Schule dar, weswegen der nächste Stundenplan damit erststellt wird.

# Kapitel 6

# Zusammenfassung

Im Rahmen dieser Bachelorarbeit sollte ein System zur automatischen Generierung von Stundenplänen für die Ludwig-Schwamb-Schule erstellt werden. Hierbei wurden besonders zwei Ziele verfolgt. Zum einen sollte das System einfach und angenehm bedienbar und auf der anderen Seite sehr flexibel und auch für andere Schulen anpassbar sein. Zur Erfüllung dieser Anforderungen wurde eine aufwändige Analyse des bestehenden Sachverhaltes durchgeführt und auf deren Basis ein komplexes Gesamt-Programm erstellt. Dieses enhält in der momentanen Form etwa 28.650 Zeilen Code, welche in 325 Klassen in 43 Packages organisiert sind. Der für die Entwicklung benötigte Aufwand und die investierte Zeit wurde dabei nicht genau gemessen; die Anzahl der Mannstunden liegt jedoch schätzungsweise im vierstelligen Bereich, hinzu kommt der Aufwand zur Erstellung dieses Dokumentes.

Zu Anfang dieser Bachelorarbeit wurden die speziellen Anforderungen der Schule eingehend betrachtet und verschiedene Alternativen zur Lösungsfindung verglichen (siehe Kapitel 2). In diesem Abschnitt kristallisierten sich bestimmte Entscheidungen und Vorgehensweisen für die Implementierung heraus, wie etwa die Verwendung einer eigenen Sprache zur Formulierung der Constraints oder die Nutzung von nachbarschaftsbasierten Verfahren für den Generierungs-Prozess. In den darauf folgenden Kapiteln wurde detailliert auf die konkrete Umsetzung des Systems zur Stundenplanerstellung (Kapitel 3) und zur Bewertung der einzelnen Pläne anhand der durch die Grammatik definierten Constraints (Kapitel 4) eingegangen. Eine anschließende Betrachtung des Systems aus Benutzersicht (in Kapitel 5) beschäftigte sich mit der grafischen Oberfläche des Systems als Schnittstelle zwischen dem Menschen und dem Generierungssystem. Danach wurde das erstellte Programm in Bezug auf die Güte der Lösungen, der Laufzeit und dem Nutzen für die betreute Schule evaluiert.

Obwohl wir versucht haben, alle wichtigen Funktionen für den alltäglichen Gebrauch des Programmes bei der Implementierung zu berücksichtigen, exisitieren einige denkbare Erweiterungen, welche im Folgenden kurz dargstellt werden sollen.

**Lehrer-Historie** Wie die vorherigen Kapitel gezeigt haben, wird generell durch das verwendete Bewertungssystem versucht, nicht nur einen gültigen Stundenplan zu schaffen, sondern vor allem einen, der möglichst viele gestellte Nebenbedingungen erfüllt. Eine denkbare Bedingung könnte zum Beispiel sein, dass die Schüler möglichst von Lehrern unterrichtet werden, die ihnen aus vorangegangenen Schuljahren bekannt sind. Eine Möglichkeit, diese Bedingung zu realisieren, ist, eine externe Relation zu erzeugen, in welcher gespeichert wird, welche Lehrer welchen Klassen bereits bekannt sind. Mittels dieser Relation könnte dann ein entsprechendes Constraint formuliert werden. Diese manuell zu pflegen und zu aktualisieren ist allerdings sehr aufwändig und könnte automatisch geschehen. Daher wäre eine Erweiterung der aktuellen Stammdatenverwaltung denkbar, welche zu jeder Klasse eine Historie über die bisherigen Unterrichtseinheiten mitführt. Diese Daten könnten dann vom System bei der Planung entsprechend berücksichtigt werden[1].

**Export- und Importmöglichkeiten** Das vorhandene System bietet zum Austausch der Daten nur die Option an, diese im zip-Format zu exportieren. Dieses enthält die verschiedenen Informationen zu einem erstellten Projekt in einer Form, die nur von unserem System interpretierbar ist. Um die Interoperabilität auch mit anderen Systemen (etwa LUSD[2], welches für alle hessischen Schulen vorgeschrieben ist) zu gewährleisten, sind auch andere Exportformate (wie etwa das `csv`-Format) sinnvoll.

**Neue Assistenten zur Formulierung der Constraints** Da die Sprache zur Formulierung der Constraints zwar eine hohe Ausdrucksfähigkeit besitzt, gleichzeitig jedoch für Nicht-Informatiker schwer zu erlernen ist, könnten dem Anwender in dieser Hinsicht weitere Hilfestellungen gegeben werden. Dies kann in Form von Assistenten geschehen, die dann etwa die Formulierung von Ausdrücken über eine grafische Schnittstelle erlauben oder eine umgangssprachliche Beschreibung einer Bedingung in ein Constraint umwandeln können. Diese Möglichkeit wurde bereits vorgesehen, indem der Constraint-Editor es erlaubt, Assistenten in Form von Plugins einzubinden und dem Benutzer zur Verfügung zu stellen.

Abschließend möchten wir uns bei allen beteiligten Personen bedanken, die uns bei der Erstellung dieser Bachelorarbeit unterstützt haben. Besonderer Dank gilt dabei Herrn Dr. Gunter Grieser, der uns von Seiten der Technischen Universität Darmstadt betreute und sich viel Zeit für unsere Fragen genommen hat. Ebenso möchten wir Frau Topaloglu von der Ludwig-Schwamb-Schule unseren Dank aussprechen, die uns erforderliche Testdaten zur Verfügung stellte und uns als ständiger Ansprechpartner zur Verfügung stand.

[1] Dieses Feature wird zeitnah implementiert und der Schule in einer neuen Version des Programmes zur Verfügung gestellt.

[2] siehe http://www.lusdportal.hessen.de

# Literaturverzeichnis


[ABCC98] Applegate, David, Robert Bixby, Vasek Chvatal und William Cook: *On the Solution of Traveling Salesman Problems.* Documenta Mathematica, Band 3:645–656, 1998.

[ADK99] Abramson, D., H. Dang und M. Krisnamoorthy: *Simulated Annealing Cooling Schedules for the School Timetabling Problem*, 1999.

[Aze87] Azencott, Robert: *Simulated annealing.* Seminaire N. Bourbaki, (697):223 – 237, 1987.

[Bar96] Bardadym, Victor A.: *Computer-Aided School and University Timetabling: The New Wave.* In: *Selected papers from the First International Conference on Practice and Theory of Automated Timetabling*, Seiten 22–45, London, UK, 1996. Springer-Verlag.

[BBK01] Böhm, Christian, Stefan Berchtold und Daniel A. Keim: *Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases.* ACM Comput. Surv., 33(3):322–373, 2001.

[BE72] Burkard, R.E. und H. Enge: *Algorithmus 25 Verfahren zur gemischtganzzahligen, konvexen Optimierung.* Computing, 14(4):389–396, 1972.

[Bea97] Beaumont, Nicholas: *Scheduling Staff using mixed integer programming.* European Journal of Operational Research, 98:473–484, 1997.

[CK96] Cooper, Tim B. und Jeffrey H. Kingston: *The Complexity of Timetable Construction Problems.* In: *Selected papers from the First International Conference on Practice and Theory of Automated Timetabling*, Seiten 283–295, London, UK, 1996. Springer-Verlag.

[Dar59] Darwin, Charles: *On the Origin of Species.* John Murray, sechste Auflage, November 1859.

[DMV89] Dinkel, J. J., J. Mote und M. A. Venkataramanan: *An efficient decision support system for academic course scheduling.* Operations Research, 37(6):853–864, 1989.

[EIS76] Even, S., A. Itai und A. Shamir: *On the Complexity of Timetable and Multicommodity Flow Problems.* SIAM Journal on Computing, 5(4):691–703, 1976.

[FCMR99] Fernandes, C., J. P. Caldeira, F. Melicio und A. Rosa: *High school weekly timetabling by evolutionary algorithms.* In: *SAC '99: Proceedings of the 1999 ACM symposium on Applied computing*, Seiten 344–350, New York, NY, USA, 1999. ACM Press.

[Gar98] Garshol, Lars Marius: *BNF and EBNF: What are they and how do they work?*, Juni 1998.

[Gid85] Gidas, Basilis: *Nonstationary Markov chains and convergence of the annealing algorithm.* Journal of Statistical Physics, 39(1–2):73–131, 1985.

[ISO91] ISO/IEC: *ISO 9126 (Information Technology - Software Product Evaluation)*, 1991.

[Löh98] Löhnertz, Martin: *Theorie und Praxis der automatischen Stundenplanerstellung.* Diplomarbeit, Universität Bonn, 1998.

[MW04] McFlynn, Daniel und Peter F. Weissman: *Using JavaCC and SableCC.* 4UPress, 2004.

[NT74] Neufeld, G. A. und J. Tartar: *Graph coloring conditions for the existence of solutions to the timetable problem.* Commun. ACM, 17(8):450–453, 1974.

[OdW83] Ostermann, R. und D. de Werra: *Some experiments with a timetabling system.* OR Spektrum, 3:199–204, 1983.

[Red04] Redl, Timothy Anton: *A Study of University Timetabling that Blends Graph Coloring with the Satisfaction of Various Essential and Preferential Conditions.* Doktorarbeit, Rice University, 2004.

[Sch99] Schaerf, Andrea: *A Survey of Automated Timetabling.* Artificial Intelligence Review, 13(2):87–127, 1999.

[Sch05] Schiele, Bernd: *Human Computer Systems.* Folien zur Vorlesung, 2005.

[SOS05] Santos, Haroldo G., Luiz S. Ochi und Marcone J.F. Souza: *A Tabu search heuristic with efficient diversification strategies for the class/teacher timetabling problem.* J. Exp. Algorithmics, 10:2.9, 2005.

[SS79] Schmidth, G. und T. Strohlein: *Timetable construction - an annotated bibliography.* The Computer Journal, 23(4):307 – 316, 1979.

[TS74] Trotter, L. E. und C. M. Shetty: *An Algorithm for the Bounded Variable Integer Programming Problem.* J. ACM, 21(3):505–513, 1974.

[UAS99] Ullman, Jeffrey, Alfred Aho und Ravi Sethi: *Compilerbau.* Oldenbourg, 1999.

[USK69] Uhlemann, K.H., K.H. Schöllkopf und B.A. Kanuer: *Untersuchungen zum Stundenplanproblem.* Elektron. Datenverarbeitung, 11:119 – 131, 1969.

[Wei99] Weingärtner, Jan: *Stundenplan-Erstellung mit evolutionären Algorithmen - ein Softwareprojekt.* Diplomarbeit, Universität Potsdam, 1999.

[Wei07] Weise, Thomas: *Gobal Optimization Algorithms - Theory and Application.* Thomas Weise, 2007.

- Benutzerhandbuch -

# CATS - Ein System zur Stundenplanerstellung

(Constraint Aided Timetabling System)

**Timo Bozsolik**
**Lars Meyer**

September 2007

Fachbereich Informatik
der Technischen Universität Darmstadt

# Inhaltsverzeichnis

# Kapitel 1

# Projekte

Sie haben die Möglichkeit, mehrere verschiedene Projekte anzulegen. Jedes Projekt besitzt eigene **Projekteigenschaften**, **Stammdaten** (welche Lehrer existieren usw ...) und **Constraints** (also die Bedingungen, nach denen das System den Stundenplan generiert). In einem Projekt können Sie beliebig viele Stundenpläne speichern.

## 1.1 Projektauswahl


Abbildung 1.1: Die Projektauswahl

Beim Starten der Anwendung erscheint zunächst der **Projektauswahldialog**, welcher in Abbildung 1.1 dargestellt ist. Hier können Sie ein neues Projekt erstellen oder vorhandene Projekte löschen. Klicken Sie auf **Go**, wird die Anwendung mit dem augewählten Projekt gestartet.

Möchten Sie ein neues Projekt anlegen, erscheint ein Dialog analog zu Abbildung 1.3, in dem Sie den Namen und die Einstellungen Ihres Projektes festlegen können. Haben Sie

ein neues Projekt angelegt oder ein ein Projekt ausgewählt, erscheint das Hauptfenster der Anwendung.

## 1.2 Menüpunkt Projekt

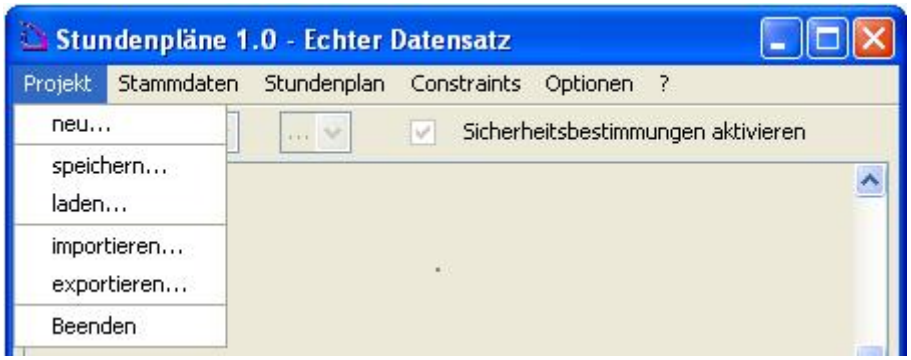


Abbildung 1.2: Menüpunkt **Projekt**

Abbildung 1.2 zeigt die möglichen Aktionen unter dem Menüpunkt **Projekt** in der Navigationsleiste.

- **neu...**: Kehrt wieder zum **Projektauswahldialog** zurück und öffnet gleichzeitig ein Fenster um ein neues Projekt zu erzeugen.
- **speichern...**: Speichert das gesamte Projekt, also die **Projekteinstellungen**, **Stammdaten**, **Constraints** und eventuelle **Stundenpläne**.
- **laden...**: Der **Projektauswahldialog** erscheint wieder, in dem das Projekt zum Laden ausgewählt werden kann.
- **importieren...**: Importiert ein Projekt aus dem *zip*-Format.
- **exportieren...**: Exportiert das momentan geöffnete Projekt in das *zip*-Format (Die Import- bzw. Export-Funktion ist dafür geeignet, um Projekte zwischen zwei Systemen auszutauschen oder um Projekte zu archivieren).
- **Beenden**: Beendet die Anwendung.

## 1.3 Menüpunkt Optionen

Unter diesem Menüpunkt können Sie die Eigenschaften des Projektes festlegen. Bei diesen handelt es sich um die **Projekteigenschaften** und die **Einstellungen**.


Abbildung 1.3: Die Projekt-Optionen

Bei den **Projekteigenschaften** können Sie zu jedem Projekt dessen Namen, den Namen der Schule und die Anzahl der Schultage bzw. -stunden bestimmen.

Bei den **Einstellungen** gibt es zwei Registrierkarten: In den **Drucken**-Optionen lässt sich für jede Druckansicht bestimmen, welche Informationen gedruckt werden sollen (siehe Abb. 1.4). Die Einstellungen der **Suche** beziehen sich auf das für die Stundenplangenerierung angewandte Such- bzw. Iterationsverfahren und sollte nur von Experten vorgenommen werden. Die mit einem * markierten Einträge sind die empfohlenen Verfahren.


Abbildung 1.4: Druckoptionen

# Kapitel 2

# Stammdaten

Mit **Stammdaten** werden die Daten der Schule bezeichnet, welche für die Generierung des Stundenplans benötigt werden. Abbildung 2.1 zeigt das Fenster zur Bearbeitung von Stammdaten, das über den Menüpunkt **Stammdaten** der Navigationsleiste geöffnet werden kann.


Abbildung 2.1: Bearbeitung der Stammdaten

Die einzelnen Kategorien der Stammdaten sind über Registrierkarten erreichbar. In jedem Bereich wird eine Liste mit den existierenden Daten in dieser Kategorie angezeigt. Hier kann einen neuer Eintrag hinzugefügt oder bereits vorhandene Einträge bearbeitet bzw. gelöscht werden.

## 2.1 Bearbeitung von Klassen


Abbildung 2.2: Bearbeitung von Klassen

Abbildung 2.2 zeigt das Fenster zur Bearbeitung von **Klassen**, welches sich öffnet, wenn man eine neue Klasse hinzufügen oder eine bereits bestehende Klasse bearbeiten möchte. Klickt man auf **übernehmen**, werden die Einstellungen gespeichert und das Fenster schließt sich. Bei einem Klick auf **abbrechen** schließt sich das Fenster ohne dass die Einstellungen übernommen werden.

## 2.2 Bearbeitung von Jahrgangsstufen


Abbildung 2.3: Bearbeitung von Stufen

Im Dialog zur Bearbeitung von Stufen gibt es zwei Registrierkarten. Unter **Allgemein** kann man den Namen einer Stufe festelegen (z.B. 3 für die 3. Jahrgangsstufe) und sieht die Klassen, die zu dieser Stufe gehören. Unter **Fächer** (siehe Abb. 2.3) kann eingestellt werden, welche Fächer zu jeweils wievielen Stunden in der Jahrgangsstufe unterrichtet werden sollen.

## 2.3 Bearbeitung von Gruppen

Gruppen dienen dazu, den Unterricht von zusammengelegten Klassen zu organisieren. Speziell können damit Bandstunden[1] wie z.B. Religion realisiert werden.


Abbildung 2.4: Bearbeitung von Gruppen

Unter **Allgemein** können die folgenden Einstellungen definiert werden:

- **Name**: Der Name der Gruppe (z.B. reli1_ev für evangelische Religion in der 1. Jahrgangsstufe)
- **Stufe**: Die Jahrgangsstufe der Klassen, die dieser Gruppe angehören sollen.
- **Fach**: Das Fach, für das die Gruppe erstellt wird.

Unter **Klassen** können die Klassen der entsprechenden Jahrgangsstufe markiert, die dieser Gruppe zugeordnet sein sollen.

[1] Als Bandstunde bezeichnet man den Unterricht, der in allen Klassen einer Jahrgangsstufe zur gleichen Zeit abgehalten werden soll

## 2.4 Bearbeitung von Fächern


Abbildung 2.5: Bearbeitung von Fächern

Abbildung 2.5 zeigt das Fenster zur Bearbeitung von Fächern. Die **Blockgröße** bedeutet, wieviele Stunden dieses Fach am Stück unterrichtet werden muss. Beispielsweise sollte Sport immer 2 Stunden am Stück unterrichtet werden und hat demnach die Blockgröße 2. Über **spezieller Raum** ist auswählbar, ob das Fach einen speziellen **Raumtyp** erfordert. Ist diese Option gewählt, kann spezifiziert werden, welcher **Raumtyp** benötigt wird. Sport sollte beispielsweise nur in einer Sporthalle unterrichtet werden.

## 2.5 Bearbeitung von Lehrern

In Abbildung 2.6 ist der Dialog zur Bearbeitung eines **Lehrers** dargestellt. Unter **Allgemein** können der **Name** des Lehrers sowie die Anzahl der **Stunden**, die er unterrichten muss, eingestellt werden. Unter **Typ** kann man auswählen, ob es sich bei dem Lehrer um einen normalen Lehrer, einen Referendar oder um eine Präventionslehrkraft handelt[2]. Desweiteren kann festgelegt werden, ob der Lehrer der **Klassenlehrer** einer bestimmten Klasse ist. Unter dem Punkt **Bild** kann dem Lehrer ein Symbol zugewiesen werden. Falls Sie ein eigenes Bild erstellt haben, können Sie dies in das System einbinden, indem Sie das Bild in den Ordner **resources\teacher_pics** im Hauptverzeichnis des Programmes kopieren. Es ist dann in der Liste anwählbar.
Unter **Fächer** können Sie bestimmen, zu welchem Unterricht der Lehrer qualifiziert ist. Die Registrierkarte **Verfügbarkeit** bestimmt, zu welchen Zeiten der Lehrer verfügbar

[2]Der Lehrertyp wird bei der Generierung eines Stundenplanes nicht beachtet und nur in den Stammdaten zur Information mitgeführt


Abbildung 2.6: Bearbeitung von Lehrern

ist. Ist ein Lehrer zu einer bestimmten Zeit nicht anwesend, teilt ihn das System bei der Gernerierung eines Stundenplans für diese Zeit nicht ein.

## 2.6 Bearbeitung von Räumen


Abbildung 2.7: Bearbeitung von Raeume

Der Dialog zur Bearbeitung eines **Raumes** enthält wiederum zwei Registrierkarten. Unter **Allgemein** kann der Name des Raums eingestellt werden und ob es sich um einen

speziellen **Raumtyp** handelt. Ist dies der Fall, kann der Typ des speziellen Raums festgelegt werden.
Neue Raumtypen können erzeugt werden indem man auf den Button **Raumtypen** klickt. Dieser befindet sich im Fenster für die Stammdatenverwaltung innerhalb der Registrierkarte **Räume** (unten rechts).

# Kapitel 3

# Stundenplan

Die Hauptansicht zeigt einen erstellten oder geladenen Stundenplan an, falls ein solcher existiert. Darunter können in zwei Registrierkarten die **Suche** gesteuert werden und Informationen über die aktuellen **Constraints** eingesehen werden. Außerdem befindet sich im Menü ein Eintrag in Bezug auf den aktuellen Stundenplan.

## 3.1 Menüpunkt Stundenplan

Abbildung 3.1 zeigt die möglichen Aktionen unter dem Menüpunkt **Stundenplan** in der Navigationsleiste.

- **leerer Plan...**: Erzeugt einen komplett leeren Stundenplan für alle Klassen, welcher dann vom Benutzer bearbeitet werden kann.
- **neu generieren...**: Startet die Generierung eines neuen Stundenplans abhängig von den eingestellten Suchoptionen. Im generierten Plan hat jede Klasse genau die in den Stammdaten festgelegten Fächer mit den spezifizierten Stundenanzahlen.
- **verbessern...**: Versucht, den vorhandenen Stundenplan zu optimieren. Dabei werden die Stundenplaneinträge nur verschoben oder es werden Lehrer- bzw. Räume geändert. Neuen Einträge werden nicht hinzugefügt, damit die geforderte Stundenanzahl der Klassen weiter eingehalten werden (im anderen Fall sollte die Option **vervollständigen...** genutzt werden). **Vorsicht**: Angenommen Sie haben seinen Plan erstellt, an dem gewisse Einträge zu dieser Zeit fest unterrichtet werden sollen. Wenn Sie danach auf **verbessern...** klicken, können die von Ihnen gesetzten Einträge unter Umständen verschoben oder geändert werden. Wenn Sie das nicht zulassen wollen, verwenden Sie für diese Einträge die Option **festsetzen** im Kontextmenü.
- **vervollständigen...**: Vervollständigt den aktuellen Plan, so dass alle Klassen die in den Stammdaten festgelegten Stundenanzahlen erfüllen. Die bereits eingefügten

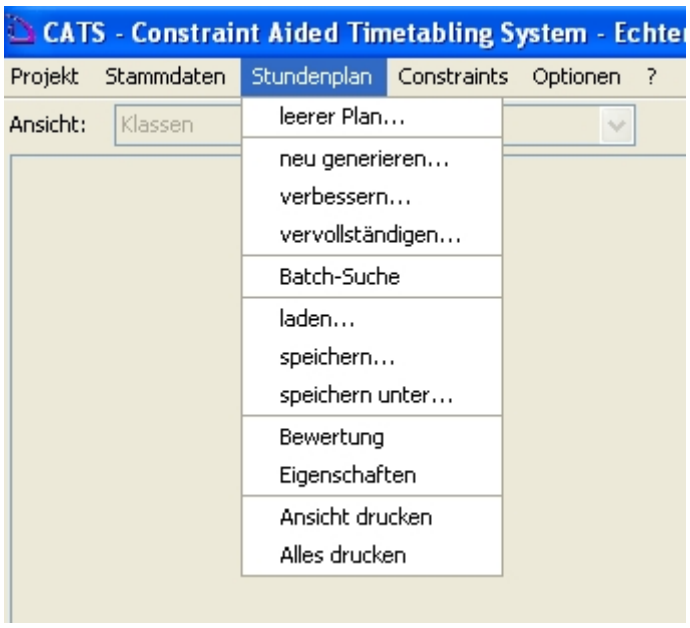


Abbildung 3.1: Menüpunkt Stundenplan

Stundenplaneinträge werden dabei nicht verändert und eine Optimierung findet nicht statt. Nach dem Vervollständigen kann man diesen Plan mit der Option **verbessern...** optimieren.

- **Batch-Suche**: Diese Option erlaubt es Ihnen, das System eine längere Zeit laufen zu lassen, um mehrere Stundepläne zu erstellen. Nachdem Sie die Option gewählt haben, können Sie angeben, wie viele Pläne erstellt werden sollen. Desweiteren kann ein Name festgelegt werden, unter welchem die erstellten Pläne dann gespeichert werden. Wenn Sie z.B. 5 Pläne erzeugen wollen und als Namen **test** eingeben, dann werden die Pläne mit den Namen *test1* - *test5* erzeugt. Die Pläne können dann mittels **laden...** aufgerufen werden.
- **laden...**: Lädt einen vorher gespeicherten Plan und zeigt diesen an.
- **speichern...**: Speichert einen Plan unter seinem Namen. Hat der Plan noch keinen Namen wird er unter *unbekannt* gespeichert. Wenn Sie einen neu generierten Plan das erste mal speichern, verwenden Sie daher **speichern unter...**, um ihm einen Namen zu geben.
- **speichern unter...**: Speichert den gerade angezeigten Stundenplan unter einem von Ihnen gewählten Namen.
- **Bewerten**: Bewertet einen Stundenplan anhand der eingestellten Constraints und zeigt die Details dieser Bewertung an.
- **Eigenschaften**: Zeigt für alle Klassen, wie viele Stunden diese haben müssen und wie viele Stunden sie im aktuellen Plan tatsächlich haben. Das Gleiche gilt für die

Lehrer und Räume. Besonders die Übersicht über die Lehrerstunden ist für die Planung sehr hilfreich.

- **Ansicht drucken**: Mit Hilfe dieser Option wird ein Stundenplan in der aktuell angezeigten Ansicht gedruckt. Dabei wird ein Browserfenster mit der Druckansicht geöffnet, so dass Sie noch Einstellungen an der Skalierung und dem Seitenformat vornehmen können. Klicken Sie in Ihrem Browser dann auf **drucken**, wird diese Ansicht gedruckt.
- **Alles drucken**: Hier wird der aktuelle Stundenplan in der Gesamtansicht ausgedruckt, wobei auch hier zunächst ein Browserfenster geöffnet wird.

## 3.2 Stundenplan bearbeiten

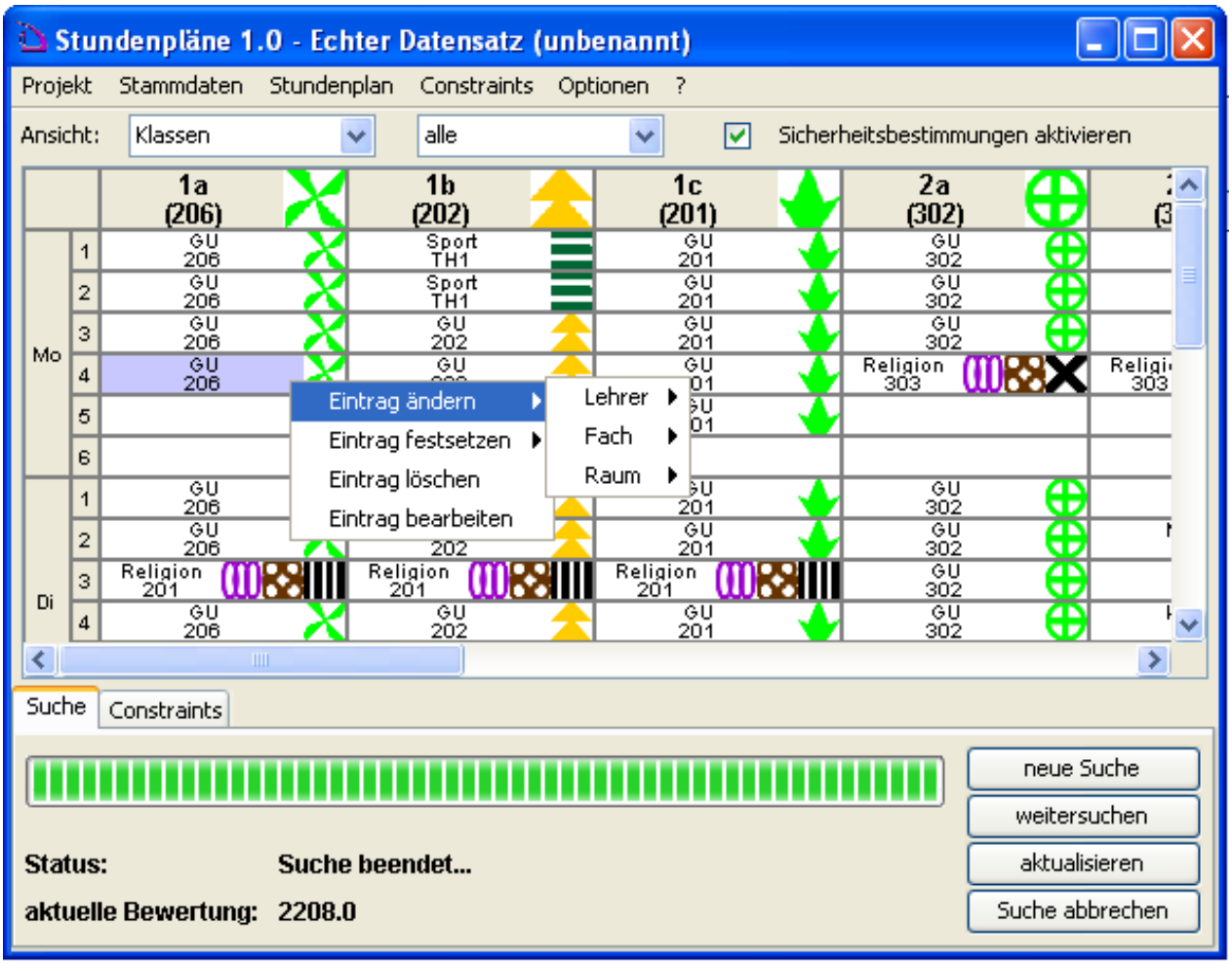


Abbildung 3.2: Bearbeitung des Stundenplans

In Abbildung 3.2 ist ein generierter Stundenplan und das Kontextmenü eines Stundenplaneintrages zu sehen. Fährt man mit der Maus auf einen Stundenplaneintrag und drückt die rechte Maustaste, so öffnet sich das gezeigte Kontextmenü. Dieses bietet die folgenden Optionen:

- **Eintrag ändern**: Erlaubt das schnelle und einfache Ändern des Lehrers, des Faches oder des Raumes für diesen Stundenplaneintrag. Falls die Option **Sicherheitsbestimmungen aktivieren** eingeschaltet ist, können hier nur zur dieser Zeit freie Lehrer, Räume etc. ausgewählt werden.
- **Eintrag festsetzen**: Über diese Option können Sie einen Eintrag auf diese Zeiteinheit fixieren. Ein festgesetzter Eintrag wird rot umrandet dargestellt und vom Programm bei der Optimierung des Plans nicht mehr verändert.
- **Eintrag löschen**: Löscht den momentan ausgewählten Eintrag.
- **Eintrag bearbeiten**: Hier öffnet sich ein Fenster zur Bearbeitung des Stundenplaneintrages. Neben dem Lehrer, dem Raum und dem Fach kann hier auch festgelegt werden, ob es sich bei dem ausgewählten Stundenplaneintrag um eine Diff-Stunde oder um eine normale Stunde handeln soll.

Öffnet man das Kontextmenü auf einem Feld des Plans, welches noch nicht belegt ist, enthält dies nur die Option **neuer Eintrag**. Wählt man diese aus, erscheint ein Fenster, in dem ein neuer Eintrag erstellt werden kann.

Ein Stundenplaneintrag kann verschoben werden, indem man diesen mit der linken Maustaste anklickt, diese gedrückt hält und den Eintrag auf die gewünschte Position schiebt. Falls sich an der Zielposition schon ein Eintrag befindet, werden die beide Einträge getauscht. Die Einträge einer Bandstunde werden immer zusammen verschoben. Ist die Option **Sicherheitsbestimmungen aktivieren** eingeschaltet, so sind keine Verschiebungen möglich, die einen Plan ungültig machen. Dies bedeutet, dass kein Lehrer zweimal zur gleichen Zeit unterrichten darf und die Verfügbarkeiten eingehalten werden müssen. Analoges gilt für Räume. Sie können den Sicherheitsmodus deaktivieren, um den Plan flexibler bearbeiten zu können. Anhand der Lehreransicht sowie die Raumansicht des Stundenplanes kann man doppelt belegte Räume oder Lehrer immer noch leicht erkennen.
Die verschiedenen Ansichten des Plans können direkt unter dem Navigationsmenü bei **Ansicht** ausgewählt werden. Es existieren Ansichten für Klassen, Lehrer und Räume. Diese können jeweils für alle oder für einzelne Klassen, Lehrer und Räume angezeigt werden.

## 3.3 Stundenplan generieren

Abbildung 3.3 zeigt den unteren Abschnitt des Hauptfensters während einer laufenden Suche.

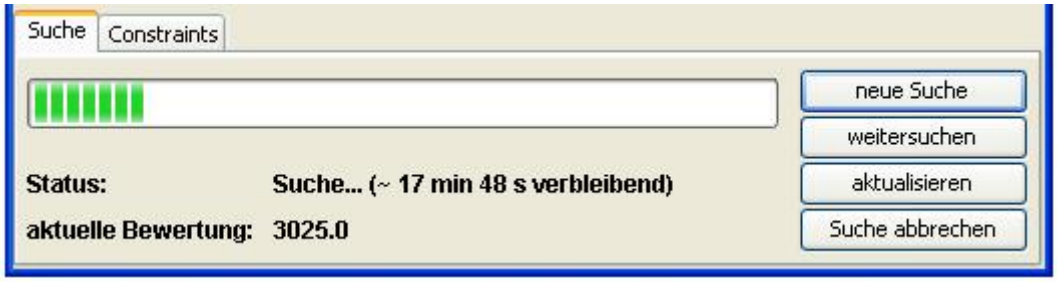


Abbildung 3.3: Unterer Anschnitt während der Suche

Wenn Sie eine neue Suche starten, wird deren Fortschritt durch einen Balken angezeigt. Desweiteren können Sie eine Abschätzung der Zeit sehen, die noch für die Generierung des Plans benötigt wird. Darunter erscheint die Bewertung des bis jetzt besten gefundenen Stundenplans.
Sie haben jederzeit die Möglichkeit, auf **aktualisieren** zu klicken, um sich den bis jetzt besten gefundenen Stundenplan anzeigen zu lassen. Die Suche läuft dabei weiter. Sie können die Suche jederzeit mit **Suche abbrechen** stoppen und so den aktuellen Plan manuell bearbeiten. Danach können Sie jederzeit auf **weitersuchen** klicken, worauf das System versucht, den momentanen Plan zu verbessern.
Weitere Möglichkeiten, einen Stundenplan zu generieren werden in Abschnitt 3.1 beschrieben.

# Kapitel 4

# Constraints

Als Constraints werden die Kriterien bezeichnet, nach denen der Stundenplan generiert bzw. optimiert wird. Ein Klick auf **Editor** unter dem Menüpunkt **Constraints** in der Navigationsleiste öffnet ein Fenster mit der Übersicht über die im Moment aktivierten Constraints (Abbildung 4.1).


Abbildung 4.1: Übersicht über die Constraints

Sie können jedes angezeigte Constraint **löschen** oder sich dessen **Details** anzeigen lassen. Innerhalb der Details können Sie das **Penalty** des Constraints verändern. Je höher dieses Penalty ist, desto schlimmer ist die Verletzung dieser Bedingung. Desweiteren sehen sie die Definition des Constraints in einer Sprache, die an die Prädikatenlogik angelehnt ist (siehe Kapitel 4.1).
Über die Option **neues Constraint** können Sie neue Constraints erstellen. Hierzu müs-

sen Sie zunächst die Beschreibung und den Typ des Constraints angeben. Sie haben die Möglichkeit zwischen zwei Assistenten bei der Erstellung des Constraints zu wählen. Benutzen Sie den Assistenten **Expression-Editor** (für Experten), müssen Sie den Ausdruck für das Constraint in der Constraints-Sprache[1] selber schreiben. Dies ermöglicht die Erstellung von beliebigen Bedingungen als Constraints. Die Erlernung der Sprache ist jedoch nicht einfach. Diese wird in Abschnitt 4.1 genauer beschrieben.
Wählen Sie den Assistenten **Standard-Constraints** (für Standard-Nutzer), können Sie ein vorgefertigtes Constraint aus einer Liste wählen. Wir haben versucht, Ihnen darüber möglichst viele Constraints bereitzustellen, die uns als relevant erschienen. Auf diese Weise müssen Sie keine eigenen Constraints formulieren.
**Relationen** und **Funktionen** sind Elemente, die in der *Constraints-Sprache* verwendet werden. Diese können Sie sich durch einen Klick auf den Knopf **Relationen** bzw. **Funktionen** anzeigen lassen. Es ist möglich, neue Relationen zu erstellen oder bestehende zu bearbeiten. Dies wird genauer in Abschnitt 4.1 beschrieben.

## 4.1 Die Constraints-Sprache

Die *Constraints-Sprache* entspricht in etwa der Sprache zur Beschreibung der Prädikatenlogik. Wie in dieser ist es möglich, Ausdrücke mit *und-*, *oder-*, *wenn-dann-* Verknüpfungen, Negationen sowie Quantoren zu erstellen.
Die atomaren Bausteine sind Relationen, Vergleichsausdrücke oder die Wahrheitswerte **true** und **false**. Ein atomarer Baustein ist entweder wahr (**true**) oder falsch (**false**).

### 4.1.1 Verknüpfungen und Negationen

Mit Hilfe der atomaren Bausteine, den Verknüpfungen sowie den Negationen können Ausdrücke formuliert werden, die zu **true** oder **false** ausgewertet werden können. Dies bedeutet, dass sie entweder zutreffen oder nicht zutreffen. Das folgende Beispiel zeigt die beiden einfachsten Constraints, die man formulieren kann.

**Beispiel**

- **true** (Dieses Constraint ist für alle Stundenpläne wahr)
- **false** (Dieses Constraint ist für alle Stundenpläne falsch)

Mit den Verknüpfungen und Negationen können jetzt beliebig komplexe Ausdrücke formuliert werden. Dabei gelten die in Tabelle 4.1 definierten Regeln für die Verknüpfungen.

[1] Mit Constraints-Sprache bezeichnen wir die Sprache zur Definition von Constraints

In den linken beiden Spalten stehen die Argumente für die Verknüpfungen und in der rechten Spalte das Ergebnis der jeweiligen Kombination. **&**, | sowie **->** sind die Zeichen für *und*-, *oder*- sowie *impliziert*-Verknüpfungen.

| Argument 1 | Argument 2 | & | \| | -> |
|---|---|---|---|---|
| true | true | true | true | true |
| true | false | false | true | false |
| false | true | false | true | true |
| false | false | false | false | true |

Tabelle 4.1: Auswertung von Verknüpfungen

Die Verknüpfungen enstprechen hierbei den intuitiven Bedeutungen dieser Operatoren. Bei einer **und**-Verknüpfung muss beispielsweise das erste **und** das zweite Argument wahr sein, damit das Ergebnis insgesamt wahr ist. Eine impliziert-Verknüpfung stellt eine wenn-dann-Beziehung dar. **Wenn** das erste Argument wahr ist, **dann** muss auch das zweite Argument wahr sein, damit das Ergebnis wahr ist.
Bei der Auswertung eines Ausdrucks hat der **&**-Operator Vorrang vor |, welcher wiederum Vorrang vor **->** hat.
Negationen werden mithilfe eines **!** beschrieben. Sie invertieren den Wahrheitswert ihres Argumentes und haben Vorrang vor allen anderen Verknüpfungen (abgesehen von Klammerausdrücken).
Das folgende Beispiel zeigt einige Ausdrücke sowie deren Ergebnis:

**Beispiel**

- **false & true | true** (ergibt wahr)
- **false & (true | true)** (ergibt falsch)
- **!(false & (true | true))** (ergibt wahr)
- **true -> true & false** (ergibt falsch)

Für die Formulierung der obigen Constraints wurden nur die atomaren Bausteine **true** und **false** verwendet. Klarerweise kann man mit diesen bei der Stundenplangenerierung keine Aussagen über zutreffende oder nicht zutreffende Eigenschaften des Stundenplanes machen. Sie werden immer gleich ausgewertet, egal welcher Stundenplan gerade bewertet wird. Statt **true** und **false** braucht man also Aussagen über den zu bewertenden Stundenplan. Diese können Sie über Relationen und Vergleichsausdrücke treffen.

### 4.1.2 Relationen

Mit Hilfe von Relationen kann geprüft werden, ob der Stundenplan bestimmte Eigenschaften hat. Ein Besipiel für eine Relation ist die Relation
$LESSON(day \times hour \times teacher \times group \times room \times subject)$.
Diese wird intern im System aus dem Stundenplan generiert und kann wie folgt abgefragt werden. Man kann die einzelnen Stellen der Relation mit Konstanten oder Variablen belegen. Variablen werden in Abschnitt 4.1.3 näher beschrieben. Das folgende Beispiel zeigt drei Constraints, die die LESSON-Relation beinhalten.

**Beispiel**

- LESSON(0,0,Müller,1a,206,Deutsch)
- LESSON(0,„Müller„,,Mathe)
- LESSON(0,0,Müller,1a,206,Deutsch) -> LESSON(0,1,Müller,1a,206,Deutsch)

Das erste Constraint wird zu wahr ausgewertet, wenn der Lehrer Müller Montags[2] in der ersten Stunde[3] die Klasse 1a in Raum 206 im Fach Deutsch unterrichtet. Ein Stundenplan, in dem dies der Fall ist, ist für das System also besser als ein Stundenplan, in dem dies nicht der Fall ist.
Nicht alle Felder in einer Relation müssen belegt werden. Wird ein Feld frei gelassen, ist diese Information nicht relevant für die Auswertung des Constraints. Das zweite Constraint wird also zu wahr ausgewertet, wenn der Lehrer Müller Montags Mathe unterrichtet. In welcher Stunde, in welcher Klasse oder in welchem Raum dies geschieht ist egal.
Das dritte Constraint soll ein Beispiel für die Verknüpfung von zwei Relationen durch einen **impliziert**-Ausdruck geben. Es besagt, dass wenn der Lehrer Müller Montags in der ersten Stunde die Klasse 1a in Raum 206 im Fach Deutsch unterrichtet, dann soll dies auch in der 2. Stunde gelten.
Neben der LESSON-Relation existieren noch weitere Relationen. Diese können Sie sich in der Relationen-Übersicht anzeigen lassen, welche über die Constraints-Übersicht erreichen können.
Hier ist es Ihnen auch möglich, neue Relationen anzulegen. Hierzu klicken Sie auf **neue Relation** und geben zunächst alle Stellen und den Namen der Relation ein. Anschließend können Sie die Relation markieren und mit **Relation bearbeiten** editieren. Hier können Sie dann die Einträge einfügen, für welche die Relation wahr sein soll.

[2] 0 = Montag, 1 = Dienstag, usw...
[3] 0 = 1. Stunde, 1 = 2. Stunde, usw...

### 4.1.3 Quantoren

Sie wissen jetzt wie Constraints mit Relationen, und Verknüpfungen formuliert werden können. Bis jetzt wurden in diesen Constraints nur Konstanten verwendet. Das bedeutet, dass Namen von Lehrern oder Räumen direkt in die Relationen eingesetzt wurden. Nur mit diesen Methoden ist es kaum möglich, allgemeine Bedingungen an den Stundenplan zu stellen (z.B. alle Klassenlehrer sollen Montags zur ersten Stunde ihre Klasse unterrichten).
Hiefür werden Quantoren verwendet. Es existieren zwei verschiedene Arten von Quantoren, die **forall**- und **exists**-Quantoren. An einen Quantor kann man Variablen binden. Diese werden dann in Klammern hinter den Quantor geschrieben mit der Angabe über ihren Typ an, also ob die Variable für einen Lehrer, einen Raum oder etwas anderes steht. Die folgenden zwei Constraints sollen die Anwendung von Quantoren verdeutlichen.

**Beispiel**

- forall(d IN day, g IN group): !LESSON(d,5„g„)
- exists(d IN day): !LESSON(d„Müller„,)

Das erste Constraint besagt: Für jeden Tag und für jede Klasse muss gelten, dass kein Unterricht in der 6. Stunde[4] stattfindet. Bei der Bewertung eines Plans wird das Penalty dann für jede Unterrichtsstunde, die zur 6. Stunde stattfindet, einzeln gezählt. Dafür prüft das System alle Tag-Gruppe-Kombinationen einzeln auf deren Wahrheitsgehalt und zählt die jeweils falschen.
Das zweite Constraint besagt, dass es mindestens einen Tag geben muss, an dem der Lehrer Müller frei hat.

### 4.1.4 Vergleichsausdrücke

Für die Formulierung einiger Constraints werden Vergleichsausdrücke benötigt. Sie können Funktionen, Zahlen und Variablen miteinander vergleichen. Hierfür stehen die Operatoren $=$, $!=$, $>$, $<$, $>=$ und $<=$ zur Verfügung.
Ein Beispiel für eine Funktion ist $TEACHERHOURS[teacher]$ welche für einen Lehrer die Anzahl seiner Unterrichtsstunden zurückgibt.
Das folgende Beispiel soll die Anwendung von Vergleichsausdrücken verdeutlichen.

[4] Da bei 0 angefangen wird zu zählen, bezeichnet 5 die 6. Stunde

**Beispiel**

- forall(d IN day, g IN group): CLASS(g) -> CLASSHOURSPERDAY[g,d] >= 3

Das Constraint besagt, dass jede Klasse mindestens 3 Stunden am Tag Unterricht haben muss. Über die vorgeschaltete Bedingung **CLASS(g)** wird erreicht, dass hierbei Gruppen (aus mehreren Klassen) nicht betrachtet werden.

## 4.2 Der Expression-Editor

Die Abbildung 4.2 illustriert den Constraint Editor, mit dem man eigene Constraints erstellen kann.

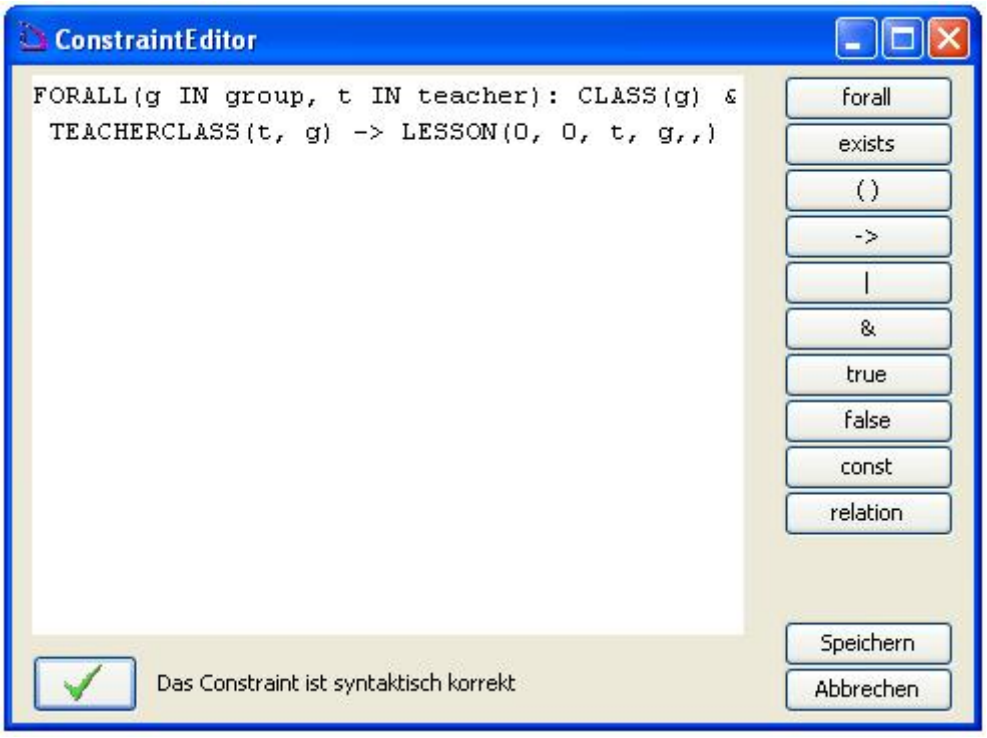


Abbildung 4.2: ConstraintEditor

Auf der rechten Seite sieht man eine Reihe von Schaltflächen, die bei der Erstellung des Ausdrucks für das Constraint behilflich sein sollen. Diese erlauben es, Sprachkonstrukte wie Quantoren oder Verknüpfungen zu generieren, so dass man nicht alles eigenhändig schreiben muss. Desweiteren hilft dies bei der Einhaltung der Syntax der Constraints-Sprache.

Bei **const** öffnet sich ein Fenster, in dem man zu jedem Variablentyp alle Konstanten auswählen kann, um diese beispielsweise in eine Relation einzufügen.
Unter **relation** erscheint eine Liste mit allen vorhandenen Relationen. Auch diese lassen sich einfach in den Ausdruck einfügen.
Durch einen Klick auf den **Prüfen**-Button unten links läßt sich prüfen, ob der formulierte Ausdruck syntaktisch korrekt ist. Erst wenn dies der Fall ist, kann das Constraint gespeichert werden.