Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Jens Schaa

Konstruktion eines Constraint Solvers zur Erstellung von Stundenplänen an Hochschulen

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Jens Schaa

Konstruktion eines Constraint Solvers zur Erstellung von Stundenplänen an Hochschulen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

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

Betreuender Prüfer: Prof. Dr. rer. nat. Michael Neitzke
Zweitgutachter: Prof. Dr. Julia Padberg

Eingereicht am: 24. März 2014

Jens Schaa

Thema der Arbeit
Konstruktion eines Constraint Solvers zur Erstellung von Stundenplänen an Hochschulen

Stichworte
Hard und Soft Constraints, Stundenplanung, Constraint Satisfaction Problem, Min-Conflict Algorithmus, Lua

Kurzzusammenfassung
Die Stundenplanung an einer Hochschule ist ein komplexes kombinatorisches Problem welches sich jedes Semester stellt und oft in mühevoller Handarbeit gelöst wird. Um den Prozess der Stundenplanung zu unterstützen wurde in der vorliegende Bachelorarbeit, auf Basis des Min-Conflict Algorithmus, ein spezialisierter Constraint Solver entwickelt, der in der Lage ist Stundenpläne für die HAW Hamburg zu erstellen. Dabei wurde eine Verfahren konzipiert bei dem man Zeitbasierte Soft Constraints flexibel und einfach hinzufügen kann.
Jens Schaa

Title of the paper
Construction of a Constraint Solver for the Creation of Timetables at Universities

Keywords
Hard and Soft Constraints, Timetabling, Constraint Satisfaction Problem, Min-Conflict Algorithm, Lua

Abstract
University course timetabling is a complex combinatorial problem which arises every semester and is often solved in laborious hand work. To assist the process of timetabling a specialized constraint solver has been developed, based on the Min-Conflict algorithm, which is able to create timetables for the HAW Hamburg. A method was developed to add time-based soft constraints in an flexible and easy manner.

Inhaltsverzeichnis

> "Constraint Programming represents one of the closest approaches computer has yet made to the Holy Grail of programming: the user states the problem, the computer solves it"

Freuder, „In Pursuit of the Holy Grail" [Fre97]

1. Einleitung

1.1. Motivation

Das Erstellen und Pflegen von Stundenplänen an Hochschulen ist ein wohl bekanntes und immer wiederkehrendes Problem, das durch seine schiere Komplexität teilweise Wochen an Arbeit und Abstimmung zwischen den Mitarbeitern einer Hochschule erfordert. Diese Pläne werden an vielen Hochschulen per Hand von einer oder mehreren Personen angefertigt. Sollte sich etwas ändern, so müssen die Betroffenen mühsam den Stundenplan anpassen.

Die Mitarbeiter der Hochschulen stehen hier vor dem Problem, begrenzte Ressourcen in zeitliche Abfolgen zu bringen sowie darauf zu achten, dass alle Konditionen der Dozenten und Veranstaltungen erfüllt werden.

Constraint Programming ist Deklarativ und kann, je nachdem wie das Modell definiert ist, auch von Personen genutzt werden, die keine Informatik- oder Programmiervorkenntnisse besitzen. Modelliert man ein Problem mittels Constraints, beschreibt man lediglich das *Was* und nicht das *Wie*. So muss man sich beim Modellieren nicht an einem bestimmten Lösungsweg orientieren, was hilft die Modellierung von der eigentlichen Problemlösung zu trennen. So kann man sich das *Wie* aussuchen, das zu dem vorliegenden *Was* passt.

Die Verwendung und Erforschung von Constraint Programming wird im Bereich der Planungsprobleme schon seit Jahrzehnten betrieben (CSP Mackworth 1977; DSCP Dechter und Dechter 1988) und für die verschiedensten Probleme sind eine Vielzahl von Algorithmen und Methoden entwickelt wurden, die alle wiederum verschiedene Anforderungen begünstigen.

Ein Constraint Solver welcher autonom Stundenpläne erstellt, oder auch nur hilft die Komplexität dieses sich immer wiederholenden Problems zu reduzieren, wäre eine Bereicherung und würde den Planungsprozess für alle beteiligten erleichtern und verbessern.

1.2. Zielsetzung

Ziel dieser Arbeit ist die Konstruktion eines spezialisierten Constraint Solvers der dazu in der Lage ist Stundenpläne für Hochschulen zu erstellen, wofür die Hochschule für Angewandte Wissenschaften Hamburg (HAW) als Vorbild dient.

Dabei reicht es nicht das vorhandene Stundenplan Problem einfach nur zu lösen, es müssen dabei verschiedene Qualitätskriterien beachtet werden. Diese sollen in Form von Soft Constraints zwar Einfluss auf die Planung haben, eine Lösung des Problems darf dadurch jedoch nicht verhindert werden. Dies ist sehr wichtig, da Planungsprobleme dazu neigen *Over-Constraint* zu sein [RM02] und so nur partielle Lösungen möglich sind.

Es soll außerdem Rücksicht auf potenzielle Benutzer genommen werden, die über wenig bis keine Programmierkenntnisse verfügen könnten.

Zeit ist außerdem ein Faktor und muss auch hier Berücksichtigt werden, der Solver soll die Planung unterstützen und sie nicht verzögern.

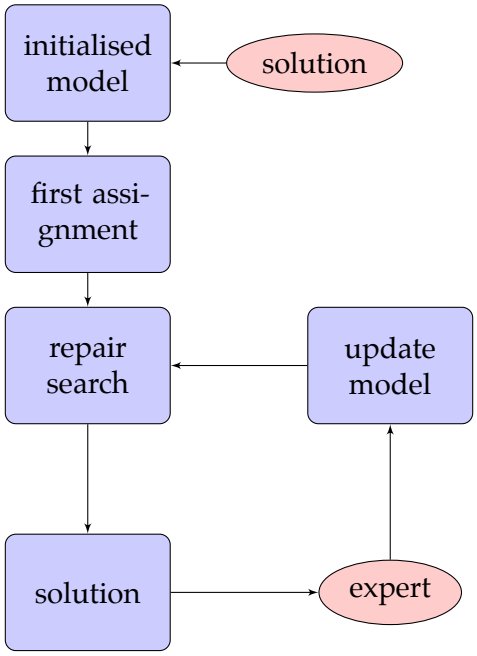


Abbildung 1.1.: Mögliche Arbeitsschritte des Solvers

Wie man in Abbildung 1.1 sehen kann, soll es für den Solver die Möglichkeit geben auf einer älteren Lösung aufzusetzen, und diese an die aktuellen Anforderungen anzupassen. Außerdem soll, wie bei dem momentan existierenden Verfahren an der HAW, ein iterativer Prozess möglich sein, bei dem der Solver Lösungen für das bestehende Pro-

blem generiert. Der Nutzer kann dann Anpassungen vornehmen, auf dessen Grundlage der Solver wiederum eine neue Lösung erstellt und eventuell entstandene Constraints Verletzungen repariert.

Auf diese Weise soll der Stundenplan inkrementell verbessert werden, bis eine für alle akzeptable Lösung gefunden ist.

1.3. Aufbau der Arbeit

Kapitel 2 soll zunächst dazu dienen die Grundlagen des sehr weiten Feldes des Constraint Programming darzulegen und einen Überblick über die vorhandenen Techniken und Algorithmen zu verschaffen.

In Kapitel 3 wird der Planungsprozess an der HAW beschrieben, in diesem Zusammenhang werden die verschiedenen Ressourcen identifiziert und festgestellt wie der derzeitige Planungsprozess mit ihnen umgeht.

Kapitel 4 widmet sich der Analyse des Problems, dort werden die aus Kapitel 3 hervorgehenden Constraints vorgestellt und die vorhandenen Ressourcen betrachtet. Im Anschluss dazu wird der hier verwendete Algorithmus vorgestellt.

Die Details der Implementation und die zugrundeliegenden Designentscheidungen werden in Kapitel 5 dargelegt.

Kapitel 6, mit Fazit und Ausblick, bildet den Schluss dieser Arbeit.

2. Grundlagen

2.1. Constraints

> "A constraint is simply a logical relation among several unkowns (or variables), each taking a value in a given domain." Barták [Bar99]

Ein Constraint [1] beschreibt die Relationen zwischen Variablen, wobei diese Variablen als Platzhalter für Werte aus einer Domäne fungieren, die durch den Constraint eingeschränkt wird.

Ein Constraint kann eine oder mehrere Variablen umfassen, deren Anzahl als Stelligkeit bezeichnet wird.

Nimmt man zum Beispiel die Aussage: 'Ich werde zwischen 10 und 12 Uhr ankommen'. Wollten wir diese Aussage mit Constraint Programming (CP) modellieren, wäre die Zeit in der wir ankommen unsere Variable V_i mit der Zeit (0 - 24 Uhr) als Domäne D_i . Der Constraint C_i würde so lauten, das V_i nicht kleiner als 10 und nicht größer als 12 sein darf.

Wollten wir das Problem nun lösen müssten wir einen Zeitpunkt auswählen der diesen Constraint erfüllt.

Definition 1 *Ein k-stelliges Constraint C ist ein Tupel bestehend aus einer Menge von Variablen $v_1, \ldots, v_k$ und einer entscheidbaren Relation R. Wobei die v_i, $i = 1, \ldots, k$, Werte aus gegebenen Wertebereichen D_i annehmen können und R eine Teilmenge von $D_1 \times \cdots \times D_k$ ist. [Run06]*

2.2. Domänen

Beim Lösen von Constraint-Netzen gibt es zwei Ansätze, die *Constraint Satisfaction* und das *Constraint Solving*. Die beiden Techniken unterscheiden sich in ihrer Definition einer Domäne.

[1] Englisch für Beschränkung, Bedingung

2.2.1. Constraint Satisfaction

Hier werden Domänen als feste Wertebereiche definiert. Das Problem was man zu lösen versucht ist ein kombinatorisches, da man jeweils versucht einen Wert aus der jeweiligen Domäne einer Variable zuzuordnen um die Constraints zu erfüllen. 95% aller praktischen Probleme können mit Constraint Satisfaction gelöst werden [Bar99][Tsa93].

2.2.2. Constraint Solving

Bei Constraint Solving sind Domänen keine fest definierten Werteräume, sondern unendlich. Das Problem ist mathematischer Natur und viele Mathematiker haben sich schon damit beschäftigt herauszufinden ob bestimmte Constraints praktisch lösbar sind.

In dieser Arbeit wird es hauptsächlich um Constraint Satisfaction gehen , weswegen ich nicht weiter auf Constraint Solving eingehen werde.

2.3. Constraint Satisfaction Problem

CSP ist eine weit verbreitete Technik zum Lösen von Problemen oder Planen von Abläufen im Bereich der künstlichen Intelligenz [BS11]. Seit seinem Ursprung im Jahre 1977 (Mackworth, Consistency in Networks of Relations [Mac77]) wurde es für eine Vielzahl von Problemen erfolgreich eingesetzt.

Ein Constraint Satisfaction Problem (CSP) ist als Triple definiert $\{X, D, C\}$:

- Eine Menge von Variablen $X = \{x_1, ..., x_n\}$
- Jede Variable x_i besitzt eine Domäne D_i mit möglichen Werten, die der Variablen zugewiesen werden können.
- Eine Menge von Constraints $C = \{c_1, ..., c_n\}$ das eine oder mehrere Variablen umfasst und die gemeinsame Belegung von Werten einschränkt.

Sinn eines CSP ist es eine Belegung aller Variablen zu finden, so dass alle Constraints erfüllt sind.

Will man ein solches Problem lösen, muss man die Variablen so belegen, das alle Constraints erfüllt sind. Natürlich ist bei kleinen Domänen und wenigen Variablen das Durchprobieren aller Kombinationen trivial, dies kann jedoch sehr schnell explodieren und trotz der einfachen Operation sehr viel Zeit in Anspruch nehmen.

Am einfachsten kann man sich ein CSP als Graph vorstellen, wobei für jede Variable ein Knoten vorhanden ist und die Verbindungen (sog. Kanten) dazwischen die Constraints darstellen.

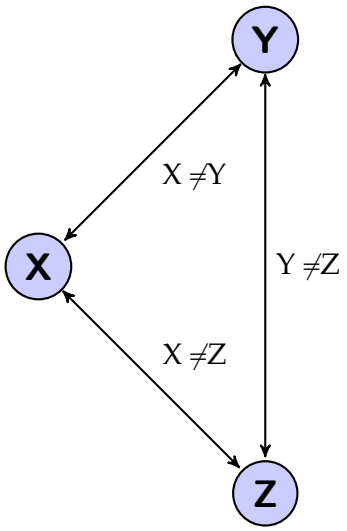


Abbildung 2.1.: Beispiel für einen Constraints Graph

2.3.1. Dynamische CSP

Beim klassischen CSP geht man davon aus das alle Aspekte eines Problems bekannt sind und sich nicht ändern. Als Variation des CSP versucht das Dynamic Constraint Satisfaction Problem (DCSP) gerade diese Problemstellungen aufzufangen in denen sich Aspekte im Laufe der Zeit verändern werden, das Problem jedoch größtenteils gleich bleibt. Im Grunde kann man kann ein DCSP als eine Sequenz von mehreren CSP sehen, da versucht wird soviel wie möglich aus der Lösungen des alten CSP mitzunehmen und dies als Hilfestellung zur Erzeugung des neuen CSP zu verwenden.

Dabei sind besonders zwei Aspekte besonders wichtig: Effektivität und Stabilität.

Die vorhandenen Strategien kann man in 3 Kategorien einordnen [VS94]:

Oracel Man nutzt die vorangehende Lösungen als Heuristik (Hentenryck & Provost 1991).

Local Repair Man nimmt die vergangene Lösung als Grundlage und versuchen sie zu reparieren, indem man inkrementell einzelne Variablenbelegungen ändert und

uns so einer vollständigen Lösung annähern (Minton 1992; Selman, Levesque & Mitchell 1992; Ghedira 1993).

Constraint Recording Trifft man bei der Suche auf inkonsistente Werte, werden neue Constraints hinzugefügt die das Benutzen dieser Werte verbietet. Diese zusätzlichen Constraints werden aufgezeichnet. Wenn man in Zukunft ein ähnliches Problem lösen möchte, kann man auf diese Constraints zurückgreifen und Konfliktwerte bereits von vornherein ausließen. Diese Erfahrungswerte beschleunigen das Finden einer Lösung (de Kleer 1998; Hentenryck & Provost 1991; Schiex & Verfaille 1993).

2.3.2. Flexible CSP

Auch wenn wir mit DCSP in der Lage sind Änderungen aufzufangen und neue Lösungen mit weniger Aufwand zu generieren sind eine sich verändernde Umgebung nur eine Schwierigkeit bei Problemen aus der Realen Welt. Normalerweise können Constraint in einem DCSP auch nur zwei Werte annehmen: Erfüllt oder unerfüllt. Was ist allerdings wenn ein Constraint wichtiger ist als ein anderer, oder manche Constraints vielleicht sogar nur 'wünschenswert' anstatt 'notwendig' sind?

Genau dafür wurden Flexible Constraint Satisfaction Problem (FCSP) entwickelt [DP98]. In seiner simpelsten Ausführung besteht ein FCSP aus Soft- und Hard-Constraints. Hard Constraints sind klassische Constraints die erfüllt werden müssen, wohingegen Soft-Constraints wünschenswert, jedoch zum Erfüllen des CSP nicht unbedingt erforderlich sind.

Im Folgenden werde ich einige Variationen vorstellen die gleichwohl alle eine Gemeinsamkeit haben, es gibt eine Funktion mit der die Güte der jeweiligen Lösung bestimmt wird, und Ziel der meisten Algorithmen in diesem Bereich ist es mit Hilfe dieser Funktion eine optimale Lösung zu finden.

MAX-CSP [Wal96] Gehört zu den kommutativen Methoden, die die Qualität einer Lösung durch Summenbildung bestimmen. Beim MAX-CSP im besonderen wird die Anzahl der erfüllten Constraints zusammen gezählt, wobei das Maximum die optimale Lösung darstellt. In Verbindung mit Soft-Constraints

Weighted CSP Als individuellere Variante können hier Constraints mit Gewichten bzw. einer Priorität versehen werden. Dabei bildet man wenn möglichst Klassen von Constraints. So können beim Lösen dieses CSP schwächere Constraints zugunsten

wichtigerer Constraints unerfüllt bleiben. Ziel ist das erfüllen aller wichtigen, und so vielen schwachen wie möglich.

Diese Lösung kann mit dem MAX-CSP zusammen benutzt werden. Zum Beispiel indem man die Qualität anhand der kumulierten Gewichte oder der Anzahl an erfüllten Constraints aus den verschiedenen Klassen bestimmt.

Fuzzy CSP [DFP96] benutzt das aus der Regelungstechnik bekannte Verfahren um Constraints als Fuzzy Mengen zu definieren. Dabei wird eine Zugehörigkeitsfunktion benutzt um zu bewerten wie weit ein bestimmter Wert einen Constraint erfüllt.

Mit jeder dieser Methoden können verschiedene Aspekte einer Problemstellung abgedeckt werden. Während MAX-CSP das CSP nur als ganzes Bewerten kann, können bei Weighted CSP zwischen verschiedenen Klassen unterschieden werden.

2.4. Suche

Um ein CSP zu lösen muss man einen Weg finden die Variablen mit Werten zu belegen, die alle Constraints erfüllen. Eine Art dies zu tun ist die Systematische Suche. Die Constraints an sich spielen bei diesem Ansatz nur eine untergeordnete Rolle, da sie lediglich dazu verwand werden um zu überprüfen, ob die momentane Wertebelegung eine Lösung ist, sprich sie alle Constraints erfüllt.

2.4.1. Generate and Test

Generate and Test (GT) ist ein Beispiel für eine simple uninformierte Suche. Alle Variablen werden (meist randomisiert) mit Werten belegt und danach überprüft. Verletzt eine der Belegungen einen Constraint wird sie verworfen.

Da hier nach jedem Fehlschlag eine neue mögliche Lösung generiert wird, ist die Leistungsfähigkeit sehr beschränkt.

2.4.2. Backtracking

Als grundlegende Verbesserung des GT belegt das Backtracking (BT) die Variablen nach und nach mit Werten aus den Domänen. Nach jeder Zuweisung wird überprüft ob die momentane Belegung Constraints verletzt. Trifft dies zu, wird der nächste Wert versucht. Ist keiner der Werte valide, so geht BT einen Schritt zurück zur vorherigen Variable und versucht diese mit einem neuen (dem nächsten) Wert zu belegen.

Nach jedem dieser Schritte erhält man zwar eine valide Teillösung, trotzdem sieht man sich mit exponentieller Laufzeit konfrontiert [Kum92].

Zu bekannten Nachteilen von Backtracking gehören:

Trashing Das wiederholte Scheitern durch den selben Grund.

Redundante Arbeit Variablen die oft Konflikte verursachen werden immer wieder ausprobiert.

Späte Entdecken von Konflikten Variablen müssen erst belegt werden damit ein Konflikt festgestellt werden kann.

2.5. Konsistenz Techniken

> "Lookahead to the future in order not to worry about the past"
> Haralick und Elliott [HE79]

Einen Weg Suchverfahren effizienter zu machen ist es den Suchraum einschränken, bzw. zu versuchen den Suchraum auf valide Werte zu reduzieren, während man den Suchraum traversiert. Man kann zum Beispiel Konsistenztechniken anwenden bevor und auch während einer Suche. Jeder der verbleibenden Werte kann Teil einer Lösung für das CSP sein.

2.5.1. Knotenkonsistenz

Knotenkonsistenz bezieht sich nur auf einen Knoten und damit nur auf eine Variable mit einem einstelligen Constraint.

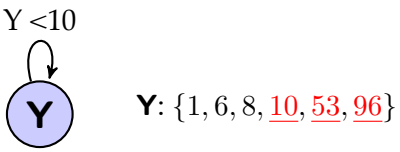


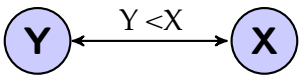
Abbildung 2.2.: Beispiel für einen einstelligen Constraint mit bestehender Knotenkonsistenz.

Um Knotenkonsistenz herzustellen schauen wir uns alle Werte der Domäne an. Befinden sich darin Werte die den Constraint nicht erfüllen können, werden diese entfernt. Wie man bei Abbildung 2.2 sehen kann bedeutet das, das alle Werte die größer sind als 10 entfernt werden müssen.

Definition 2 *Ein Knoten V_i ist konsistent, wenn für jedes $x \in D_i$ gilt das $V_i = x$ den Constraint von V_i erfüllt.*

2.5.2. Kantenkonsitenz

Die Kantenkonsistenz schaut sich immer zwei Variablen an und den Constraint der sie verbindet. Dabei wird geschaut ob, wenn wir diese Variable mit einem Wert belegen, es einen Konflikt mit der anderen Variable geben kann. Dies kann man ausschließen, wenn es für jeden Wert in der einen Domäne einen Wert in der anderen Domäne gibt, die beide den Constraint erfüllen.



Y: $\{3, 5, \underline{16}\}$ **X**: $\{2, 8, 11\}$

Abbildung 2.3.: Beispiel für Kantenkonsistenz für die Kante (V_y, V_x).

Bei Abbildung 2.3 gibt der Constraint vor das Y kleiner sein soll als X. Wollen wir Kantenkonsistenz herstellen, müssen wir alle Werte aus D_y entfernen, die einen Konflikt mit X erzeugen würden. Wie man sehen kann ist in Abbildung 2.3 (V_y, V_x) nach der Entfernung von 16 Kantenkonsistent, (V_x, V_y) ist dies aber nicht (es gibt keinen Wert in D_y der kleiner ist als 2). Dieser Effekt kommt zustande da das Konzept der Kantenkonsistenz gerichtet ist.

Definition 3 *Eine Kante (V_i, V_j) ist dann konsistent wenn für jedes $x \in D_i$ ein $y \in D_j$ sodass $V_i = x$ und $V_j = y$ den Constraint zwischen (V_i, V_j) erfüllt.*

2.5.3. Directional Arc Consistency

Um Directional Arc Consistency (DAC) anwenden zu können, müssen die Variablen in eine Reihenfolge gebracht werden. Danach stellt DAC Kantenkonsistenz mit zukünftigen Variablen her, indem es Konsistenz entlang dieser Reihenfolge prüft.

2.5.4. Pfadkonsistenz

Pfadkonsistenz geht noch einen Schritt weiter als Kantenkonsistenz. Auch hier nimmt man sich jeweils zwei Variablen, konstruiert sich einen Pfad der diese beiden verbindet und überprüft ob man für jeden Wert eine Entsprechung in der nächsten Variable findet und mit diesem eine Entsprechung in der Ziel-Variable.

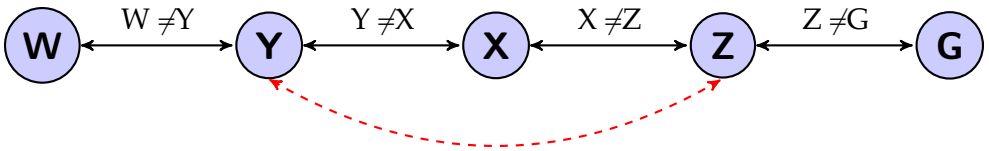


Abbildung 2.4.: Beispiel für Pfadkonsistenz.

Auch wenn man für das Herstellen kompletter Pfadkonsistenz lediglich Pfade der Größe 2 benötigt (siehe [Mon04]), sind Algorithmen die dies Erreichen sehr speicherintensiv und deswegen nur bedingt praktisch nutzbar.

Definition 4 *Ein Pfad zwischen* (V_i, V_k, V_j) *ist dann konsistent wenn für jedes* $x \in D_i$ *ein* $z \in D_k$*, für das ein* $y \in D_j$ *existiert so das* $V_i = x$*,* $V_k = z$ *und* $V_j = y$ *die Constraints zwischen* (V_i, V_k) *und* (V_k, V_j) *erfüllen.*

2.6. Constraint Propagation

Ändert man einen Wert in einer Variable des Constraint-Netzes muss überprüft werden, ob diese Änderung, mit denen über Constraints mit diesem Wert verbundenen Variablen, eine valide Teillösung bildet. Techniken wie Suche und Konsistenzprüfung haben wir bereits kennengelernt, jedoch werden sie selten alleine eingesetzt sondern zusammen in ihren Grund- oder Hybridformen um die jeweiligen Schwächen auszugleichen.

2.6.1. Look Back

Unter Look Back lassen sich Beispielsweise Suchstrategien, die auf dem BT aufbauen, einordnen. Erweitert man das BT um einen *look back* erhält man eine informierte Suche die in die Vergangenheit schauen und so effizienter Suchen kann in dem sie mögliche Belegungen mit früheren Erfahrungen vergleicht. So können einige Nachteile des BT beseitigt werden.

Backjumping

Eine Version des BT der das *trashing* zu verhindern versucht ist das Backjumping (BJ). Dieser Algorithmus unterscheidet sich von dem BT in der Art des zurück Gehens. Hat das BT alle Werte einer Domäne durchprobiert geht es genau einen Schritt im Suchbaum zurück. BJ hingegen kann mehrere Werte zurückspringen.

Es gibt verschiedene Methoden wann ein Sprung erfolgen kann. In der Grundversion (Gasching 1979) [Pro93] passiert ein solcher Sprung im *leaf dead end* [2], das bedeutet das man alle Werte einer Variablen ausprobiert, und kein einziger eine valide Teillösung erzeugen konnte. Für diesen Fall führt jede Variable eine Referenz auf die Variablen mit denen zuletzt ein Konflikt bestand. Eine dieser Variable muss dafür verantwortlich sein das wir keine passende Belegung finden. Aus diesen Variablen wird die ausgewählt, die unserer in der Reihenfolge der Belegungen am nächsten ist (siehe Gaschings: min. Prefix) und dort wird der nächste Wert ausprobiert und das BJ arbeitet wie das BT normal weiter.

Backchecking

Als weitere Variante des BT gibt es das Backchecking (BC). Dieser Algorithmus versucht einiges an redundanter Arbeit einzusparen. Dabei wird versucht das Testen von Teillösungen gegen die Constraints nicht zu wiederholen. Dafür wird gespeichert welche Belegungen miteinander inkompatibel waren. Soll also eine Variable mit einem Wert belegt werden, kann der Algorithmus nachschauen ob in der Vergangenheit dieser Wert schon mal mit einem Wert der vorhandenen Teillösung in Konflikt stand.

2.6.2. Look Ahead

Anders als das Look Back, das die Vergangenheit betrachtet, versucht das Look Ahead in die Zukunft zu blicken. Dabei wird immer, wenn eine Variable belegt wird, nicht nur geschaut ob sie die verbundenen Constraints verletzt, sondern auch die Auswirkungen auf die Nachbarvariablen die mit dieser verbunden sind. Dabei kommen Konsistenztechniken zum Einsatz die versuchen mit jedem Schritt der Suche den verbleibenden Suchraum weiter einzugrenzen und mögliche Konflikte auszuschließen.

[2] Deutsch: Blatt Sackgasse, Blatt in Sinne eines Blattes am Suchbaum

Forward Checking

Als Vertreter des Look Ahead versucht Forward Checking (FC) Konsistenz mit möglichen zukünftigen Variablenbelegungen herzustellen. Bei Belegung einer Variable prüft es die Nachbarvariablen auf Kantenkonsistenz und entfernt inkonsistente Werte für die Zeit wie unsere momentane Variable mit diesem bestimmten Wert belegt ist.

Partial Look Ahead

Als Erweiterung des FC überprüft das Partial Look-Ahead (PLA) nicht nur Variablen, die direkt miteinander Verbunden sind, sondern schließt noch weitere unbelegte Variablen mit ein. PLA verwendet DAC, wobei jedes Paar nur einmal überprüft wird.

2.7. Lokale Suche (Heuristik)

Neben der systematischen Suche nach Variablenbelegungen gibt es Methoden die Heuristik verwenden um Teillösungen zu erzeugen oder diese zu erweitern bzw. zu reparieren. Dabei kommen bei der Variablenauswahl und -belegung Heuristik, Erfahrungswerte oder zufällige Entscheidungen zum Einsatz.

Durch die nicht-deterministische und unvollständige Natur können diese Algorithmen jedoch nicht garantieren das eine Lösung gefunden wird. Dafür sind diese Verfahren in der Lage auch in großen Domänen schnell eine Lösung zu finden [Run06], besonders wenn eine klassische Suche zu lange dauert oder wenn eine beliebige Lösung für das vorhandene Problem ausreicht.

2.7.1. Hill Climbing

Beim Hill Climbing wird zunächst eine (ggf. zufällige) Variablenbelegung erzeugt. Danach werden Schritt für Schritt Variablen so neu belegt, das mehr Constraints erfüllt werden um so zu einer besseren Lösung zu kommen.

Der Algorithmus eignet sich besonders ein *lokales Optimum*, eine Lösung die durch betrachten von Nachbarschaftsbeziehungen nicht weiter verbessert werden kann, zu finden. Durch seine teilweise zufällige Arbeitsweise ist es schwierig die beste Lösung (ein Globales Optimum) zu finden.

2.7.2. Min-Conflicts

Min-Conflict (MC) wählt randomisiert eine Konfliktvariable aus und ersetzt den vorhandenen Wert durch einen, der möglichst mehr Constraints, die mit dieser Variable in Beziehung stehen, erfüllt [Min+92]. Kann der Algorithmus keinen solchen Wert finden wird per Zufall ein Wert gewählt der nicht noch mehr bzw. gleich viele Constraints verletzt, der vorhandene Wert wird nur beibehalten wenn kein solcher Wert existiert.

Der Algorithmus von Minton u. a. basiert auf dem Neuralen Netzwerk von Adolf und Johnson welches genutzt wurde um die Aktivitäten des Hubble Space Telescope zu planen. Zudem haben sie gezeigt, das MC besonders gut mit großen Versionen des *n-queens Problem* zurechtkommt und im Allgemeinen mit Problemen deren mögliche Lösungen eine geringe Distanz aufweisen.

Neben Hill Climbing kann die MC Heuristik kann auch in Verbindung mit Backtracking angewandt werden. Dabei werden die Variablen nach MC geordnet und der Algorithmus versucht den Wert jeder Variable nur einmal neu zu belegen. Gibt es keine passende Belegung wird Backtracking auf die vergangenen Teillösungen angewandt.

2.7.3. Tabu Search

Ein Problem mit randomisierten Algorithmen ist, das die Möglichkeit besteht das Konflikt-Zustände zum wiederholten Mal ausprobiert werden. Tabu Search (TS) versucht diese Zyklen zu vermeiden. Der Algorithmus führt eine Tabu-Liste mit Teillösungen oder Belegungen, die während einer bestimmten Anzahl von Iteration nicht verwendet werden dürfen. Zudem können auch Regeln definiert werden, um die Suche in bestimmte Richtungen zu lenken. Dies führt auch dazu, das der Algorithmus nicht bei einem lokalen Minimum hängen bleibt, sondern die Möglichkeit hat auch andere Zustände mit besseren Lösungen zu erreichen.

Unter bestimmten "Aspirationskriterien" dürfen Tabus jedoch aufgehoben werden, zum Beispiel unter der Bedingung das der zu erreichende Zustand besser ist als der momentane.

2.8. Variablen- und Werteordnung

> "To succeed, try first where you are most likely to fail" Haralick und Elliott [HE79]

Neben der Auswahl des richtigen Algorithmus kann die Reihenfolge in denen die

Variablen und Werte betrachtet werden entscheidend sein. Welche Heuristik man anwenden sollte hängt stark vom jeweiligen Problem ab. Setzt man zum Beispiel eine Heuristik ein die das Backtracking reduziert kann man beim BT viel *trashing* [3] einsparen, welches eines der drei Hauptschwachstellen des BT ist. Dabei kann die Auswahl der Reihenfolge statisch (static variable ordering) oder dynamisch (dynamic variable ordering) erfolgen. Während statische Auswahl weniger Rechenaufwand vor dem eigentlichen Suchvorgang verspricht, eignet sich die dynamische Ordnung der Werte beim Einsatz von look-ahead Verfahren. Dabei sind dynamische Verfahren besonders nützlich wenn sich der Wertebereiche der Variablen stark verändern, wohingegen sich statische Verfahren lohnen wenn die verschiedenen Variablen unterschiedlich stark beschränkt sind.

2.8.1. Minimum Domain

Als eine der einfachsten Methoden versucht Minimum Domain Konflikte in der Zukunft zu vermeiden indem die Variablen nach Domänengröße geordnet werden. Es wird angenommen das Variablen mit kleinen Domänen schneller erschöpft sind und so früher zu Konflikten führen. Diese Ordnung kann auch dynamisch in Verbindung mit Konsistenztechniken eingesetzt werden, da sie während der Suche die Domänen reduzieren. In der Literatur finden sich viele Hinweise das es sich bei Minimum Domain um eine der effektivsten und auch günstigsten Heuristiken handelt.

Zusätzlich kann Minimum Domain auch mit anderen Heuristiken kombiniert werden um ein unentschieden beim Ordnen zu entscheiden.

2.8.2. minimal-width-ordering

Eine Ordnung zum Reduzieren von Backtracking ist die minimal-width-ordering [4]. Diese Heuristik schaut sich die Constraints an die verschiedenen Variablen verbinden und bringt sie in eine Reihenfolge.

Dabei sollen Variablen mit vielen Constraints zuerst belegt werden, da man annehmen kann das man für Variablen mit wenigen Abhängigkeiten einfacher einen validen Wert finden kann.

Um dies zu erreichen wird muss eine Ordnung mit einer möglichst geringen *Weite* gefunden werden. Die Weite einer Variable V_i definiert sich durch die Anzahl der durch einen Constraint verbundenen Variablen die sich in der Ordnung vor V_i befinden. Die Weite einer Ordnung ergibt sich durch die maximale Weite der Variablen.

[3] Das wiederholte Ausprobieren von Werten die einen Konflikt verursachen
[4] Ordnung mit minimaler Breite

Da sich die Constraints bei einer Suche nicht ändern kann man diese Methode statisch durchführen.

Als simplere alternative zum minimal-width gibt es noch das maximum-degree-ordering [5] welches weniger optimale Lösungen finden kann, dies jedoch mit wesentlich weniger Rechenaufwand tut. Die Variablen werden absteigend nach ihrem Grad sortiert, bzw. nach der Anzahl von Constraints die sie einschränken.

2.9. Hybride Verfahren

Unter Hybriden Verfahren versteht man im Allgemeinen das Kombinieren von *look-back* und *look-ahead* Methoden in solch einer Weise das beide Verfahren von der Arbeitsweise des anderen profitieren.

So kann man sagen das die Lösung eines CSP in der Regel aus vorwärts und rückwärts Schritten besteht, wobei die jeweiligen Varianten von look-back immer mehr Informationen für die Suche hinzufügen und so in der Lage sind beim Schritt rückwärts größere Teile des Baumes zu überspringen, wohingegen sie bei einem Schritt vorwärts das einfache Belegen von Variablen verwenden. Während die Basis look-ahead Methoden verschiedene Varianten der Konsistenztechniken für den Schritt vorwärts anbieten, verwenden sie beim einem Schritt rückwärts weiterhin chronologisches Backtracking.

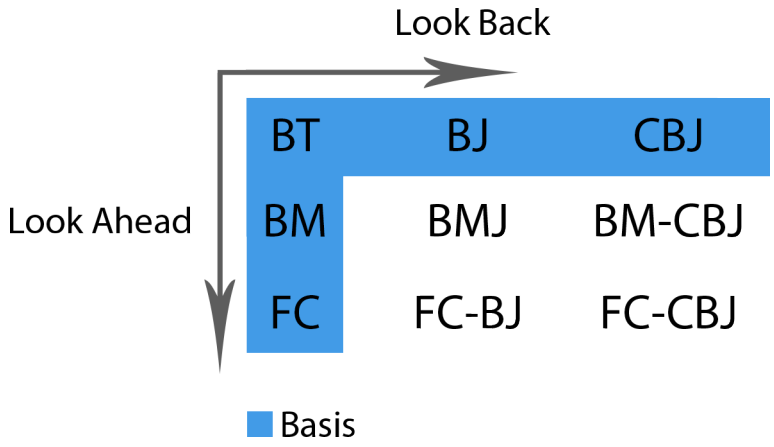


Abbildung 2.5.: Die 5 Basis Algorithmen und die 4 Hybriden [Pro93]

Theoretisch bietet es sich an den Vorwärtsschritt einer look-ahead Methode mit dem Rückwärtsschritt eine look-back Methode zu verbinden um das Beste aus beiden Welten zu erhalten.

[5] Ordnung des Maximalen Grades

Beide Strategien müssen sich nicht unbedingt ergänzen, sondern können sich auch negativ beeinflussen. Umso mehr Aufwand man in Konsistenztechniken investiert, umso ineffizienter wird Backtracking im Vergleich. Besonders bei teuren Backtracking Strategien kann der Overhead der jeweiligen Methoden um so mehr ins Gewicht fallen [CB11].

Der nutzen eines hybriden Verfahrens hängt stark von der vorhandenen Problemstellung ab. Viele Veröffentlichungen untersuchen die vorgeschlagenen Methoden anhand empirischer Untersuchungen mit exemplarischen Problemstellungen an denen man sich orientieren kann. Trotzdem muss man im Einzelfall selbst Untersuchungen vornehmen welche Kombination für das vorhandene Problem geeignet ist.

2.10. Constraint Optimierung

Im Gegensatz zum normalen CSP, wo das finden einer Lösung im Fokus steht, geht es beim Constraint-Optimisation-Problem (COP) darum, wenn möglich die beste Lösung zu finden. Es gibt mehrere Wege um eine Lösung bewerten zu können, die einfachste wäre nach der Anzahl der Soft-Constraints zu gehen die erfüllt sind. In der Regel braucht man jedoch eine Zielfunktion die den Zustand auf einen numerischen Wert abbildet, den man Mini- oder Maximieren kann [Min+92].

Viele Algorithmen, die auf bestehenden Lösungen aufsetzen und nach neuen Lösungen suchen, zum Beispiel das bereits erwähnte Hill Climbing, kann man durch eine solche Zielfunktion erweitern um ein COP zu lösen.

2.10.1. Branch and Bound

Eine Methode zur dediziert Optimieren einer Lösung ist das Branch and Bound (BnB). Da eine erschöpfende Suche nach der besten Lösung lange dauern würde teilt das BnB den Suchraum in kleinere Teilmengen (Branches) auf in denen anhand einer Heuristik (Bound) entschieden wird ob eine spezifische Teilmenge weiter untersucht werden soll.

2.11. Over-Constraint

Für manche sehr komplexe Probleme gibt es keine vollständige Lösung. Im Falle eins CSP bedeutet das, dass man nicht alle Constraints erfüllen kann. Ein solches Problem nennt sich *Over-Constraint* und kann von klassischen Algorithmen nicht gelöst werden, obwohl Lokale Suche es schaffen kann eine entsprechende Teillösung zu optimieren.

2.11.1. Partial Constraint Satisfaction

Ein Weg eine Lösung für ein Over-Constraint-Problem zu finden ist es die bestehenden Constraints zu verändern um das Problem zu vereinfachen. Dazu kann man eine Domäne erweitern oder Variablen und Constraints entfernen. Dieser Vorgang wird "ein Constraint schwächen" genannt. Beim Partial Constraint Satisfaction (PCS) wird ein bestehendes CSP zusätzlich um eine Bewertungsfunktion erweitert, der den Zustand des CSPs auf einen Wert abbildet. Wenn man daran geht Constraints zu schwächen um eine Lösung zu erhalten kann man mit diesem Wert die verschiedenen Lösungen miteinander vergleichen um eine Lösung zu finden die in ihrer Qualität dem eigentlichen Problem am nächsten ist.

2.11.2. Constraint Hierarchies

Bei praktischen Problemen muss man zwischen Constraints differenzieren, die veränderbar sind und welchen deren Änderung in der Realität nicht möglich ist.

Dem nimmt sich die Constraint Hierarchie an indem man die Constraints in einer Hierarchie einordnet. Die oberen Schichten stellen harte Constraints dar, die erfüllt werden müssen wohingegen die Constraints auf den unteren Schichten immer weicher werden. Weiche Constraints müssen nicht erfüllt werden um eine valide Lösung zu erzeugen und dürfen die Erfüllbarkeit von harten Constraints nicht beeinflussen.

Dabei kann man ein solches Problem auf mehrere Arten lösen [BFW92]. Bei der *refining method* entfernt man zunächst alle Constraints und fügt sie anschließend Schritt für Schritt ein und belegt die Variablen mit einem Wert. Entweder zuerst die starken und später die schwächeren Constraints oder indem man vorher eine Reihenfolge plant und sich anschließend durch die verschiedenen Ebenen durcharbeitet.

Bei der *perturbation method* liegt eine vorhandene Lösung vor bei denen sich die Constraints geändert haben und das Netz repariert werden muss. Die Aufgabe liegt darin, die Variablen auszuwählen die geändert werden müssen um eine valide Lösung zu erhalten. Dafür werden Konfliktvariablen Schrittweise betrachtet und als *read only* deklariert um die Auswahl so lange einzuschränken bis eine Entscheidung getroffen ist.

3. Anforderungen

Wie an vielen Hochschulen wird die Veranstaltungsplanung auf die verschiedenen Departements verteilt, die einzeln für sich einen Stundenplan mit den für sie vorhandenen Ressourcen erstellen. Dabei vereint ein Departement, als Untergruppe einer Fakultät, Studiengänge verschiedener Fachrichtungen. Gehen wir von dem einfachen Fall aus, das zwischen den verschiedenen Departements keine Ressourcenkonflikte vorhanden sind, können die Pläne unabhängig voneinander erstellt und später zusammengefügt werden.

Als Beispiel gehören zum *Departement Informatik* an der HAW Hamburg die folgenden Studiengänge:

- Angewandte Informatik
- European Computer Science
- Technische Informatik
- Wirtschaftsinformatik
- Master Informatik

Jeder dieser Studiengänge besteht aus Modulen die Veranstaltungen, Praktika und Seminare beinhalten. Zu den Ressourcen eines Departements gehören Professoren, wissenschaftliche Mitarbeiter und Räume. Die Räume lassen sich je nach Ausstattung in verschiedene Typen unterteilen, wobei jede Veranstaltungen andere Anforderungen hat die berücksichtigt werden müssen.

Anders als bei Universitäten, liegt bei der Fachhochschule (FH) die Hauptlast der Veranstaltungsplanung bei der Einrichtung selbst. Die Studiengänge folgen einer festen Linie und die Veranstaltungen pro Semester sind für die Studenten in der Regel vorgegeben. Streng gesehen müssen nur diese bei der Planung kollisionsfrei sein. An Universitäten an denen die Kurse frei gewählt werden können nimmt die Planung eine ganz andere Komplexität an, da im Idealfall viele Kurse auch über Semester hinweg

kollisionsfrei sein müssen. So sind an Universitäten hauptsächlich die Studenten verpflichtet sich darum zu kümmern ihre Kurse im Semester so zu wählen, das sie an ihnen teilnehmen können. Natürlich können Studenten einer FH auch über verschiedene Semester hinweg Kurse besuchen, dies muss jedoch bei der Planung nicht unbedingt berücksichtigt werden.

3.1. Planungsprozess

Damit das Planen beginnen kann werden zu Anfang die Professoren den Lehrveranstaltungen zugeordnet. Dieser Prozess ist administrativ und geschieht unabhängig von der eigentlichen Planung.

Neben der Tatsache das ein Professor nicht mehrere Veranstaltungen zur gleichen Zeit haben kann, bringen diese noch weitere Einschränkungen mit. Während der Planung können sie Wünsche über die Legung ihrer Veranstaltung äußern. Zudem müssen Zeiten wie Sprechstunden, Dienstverpflichtungen und Gremienarbeit berücksichtigt werden, während dessen keine Vorlesung stattfinden kann.

Als erstes wird ein vorläufiger Plan erstellt, in dem sich das Einfügen von Veranstaltungen und das Einarbeiten des Feedbacks abwechseln. Man beginnt mit der Veranstaltung, mit der am meisten Bedingungen verknüpft sind. Im Falle der HAW sind dies die Praktika. Labor Räume in denen Praktika durchgeführt werden können sind eine kritische Ressource, zudem muss man die Zeiten einmal mit dem Professor sowie mit einem wissenschaftlichen Mitarbeiter abstimmen.

Ist das Planen der Praktika abgeschlossen, werden nach und nach die Vorlesungen hinzugefügt. Nach jedem Schritt werden die Änderungen den Beteiligten kommuniziert, die selbst noch einmal Änderungswünsche einbringen können. Führen diese zu Konflikten müssen diese manuell in den Plan eingearbeitet und nochmals abgestimmt werden.

3.2. Zeit

Geplant wird wochenweise, denn mit wenigen Ausnahmen findet jede Veranstaltung einmal in der Woche, zur gleichen Zeit statt. Ist die Woche eines Semesters geplant, kann diese auf das gesamte Semester erweitert werden.

Das Sommersemester erstreckt sich Beispielsweise von Woche 10 bis 28. Wobei die Wochen 10 und 11 für die Orientierungseinheit reserviert sind und für die Planung nicht berücksichtigt werden. In Woche 27 und 28, am Ende des Semesters, finden die

Klausuren statt. Das heißt die eigentlichen Veranstaltungen finden über 15 Wochen, ab der 12. bis zur 26. Kalenderwoche, statt.

Ein Tag an der HAW wird in 6 gleichgroße Zeiteinheiten, Akademische Viertel genannt [1], eingeteilt. Zwischen dem 2. und 3. Viertel befindet sich eine einstündige Mittagspause, die ihre Entsprechung am Nachmittag in einer 30 minütigen Pause zwischen dem 4. und 5. Viertel findet.

1. Viertel 8:15 - 9:45

2. Viertel 10:00 - 11:30

3. Viertel 12:30 - 14:00

4. Viertel 14:15 - 15:45

5. Viertel 16:15 - 17:45

6. Viertel 18:00 - 19:30

3.3. Aufbau der Angewandten Informatik

Nehmen wir als Beispiel den Bachelorstudiengang der *Angewandten Informatik*. Dieser ist auf 6 Semester ausgelegt und besteht aus 29 Modulen [2].

In Zahlen ausgedrückt baut sich der Studiengang wie folgt auf:

Vorlesungen 24

Übungen 4

Praktika 19

Gesellschaftswissenschaften 3

Seminar 1

Projekt 1

[1] Bei der HAW handelt es sich dabei nur um die Bezeichnung der Zeiteinheiten. Die Veranstaltungen beginnen wie im Veranstaltungsplan angegeben, ohne die Anwendung des bei manchen Universitäten üblichen Akademischen Viertels.

[2] Quelle: Prüfungsordnung Studiengang Angewandte Informatik 2008

3.4. Modul

In der Prüfungsordnung jedes Studiengangs ist festgelegt in welchem Semester ein Modul stattfindet, welche Prüfungsvorleistung erfüllt werden müssen und wie viele Semesterwochenstunden (SWS), für die jeweilige Veranstaltung eingeplant werden muss. Dabei entspricht eine SWS einer Schulstunde von 45 Minuten pro Woche.

3.5. Vorlesung

Zu jedem Modul gehört eine Vorlesung die einmal pro Woche stattfindet. In der Prüfungsordnung findet man die veranschlagten SWS und üblicherweise sind für eine Vorlesungen 3 SWS eingetragen. An der HAW dauern die Vorlesungen üblicherweise 2 Viertel (180 Minuten) und damit eine Schulstunde länger als in der Prüfungsordnung eingetragen.

Durch diesen Umstand kann der vorhandene Stoff bereits in 12, statt 16 Wochen bearbeitet werden was den Lehrenden einen Puffer bietet. Fällt zum Beispiel eine Vorlesung aus, kann diese auf den nächsten Termin verschoben werden ohne die Planung der anderen Veranstaltungen zu gefährden.

3.6. Prüfungsvorleistungen

Um an der Klausur am Ende des Semesters teilnehmen zu dürfen, sind auch an der HAW Prüfungsvorleistungen zu erbringen. In unserem Beispiel, der Angewandten Informatik, besteht dies in der Regel aus dem erfolgreichen Teilnehmen an einem Praktikum oder einer Übung.

Die Prüfungsvorleistungen heben sich durch eine besondere Eigenschaft von den anderen Veranstaltungen ab, die Gruppengröße ist auf 16 Teilnehmern pro Termin beschränkt. Dadurch müssen die Studenten auf Gruppen verteilt werden. Zudem handelt es sich dabei nicht unbedingt um eine kontinuierliche Veranstaltung, denn für beide Varianten der Prüfungsvorleistung sind in der Prüfungsordnung 1 SWS veranschlagt. Da auch hier die traditionellen zwei Viertel pro Termin greifen, kann die Anzahl an Praktikumsterminen pro Semester auf 4 reduziert werden.

Die andere Art der Prüfungsvorleistung ist die Übung. In der momentan stattfindenden Planung wird diese einfach an die Vorlesung angehängt, die sich dadurch um ein Viertel verlängert. Dies ist bei drei Übungsgruppen auch Problemlos möglich. Bei einer

anderen Gruppenanzahl oder anderen zeitlichen Bedingungen muss jedoch auch hier ein eigener Termin gefunden werden.

3.7. Wahlpflichtmodule und Projekt

Anders als die normalen Vorlesungen, sind die Themen der WP und des Projektes nicht fest vorgegeben. Das entscheiden die Professoren oder Dozenten die sich entschließen eine solche Veranstaltung anzubieten. Dadurch kann auch die Anzahl an WP pro Semester stark variieren. Wichtig für die Planung ist tatsächlich nur in welchem Semester das Wahlpflichtmodul verankert ist, da es sich nicht mit den anderen Veranstaltungen dieses Semesters überschneiden darf. In unserem Beispiel, der Angewandten Informatik, müssen drei WP absolviert werden, jeweils eines im Vierten, Fünften und Sechsten Semester. Da die Studenten nur jeweils ein Wahlpflichtmodul (WP) pro Semester besuchen müssen, darf sich kein WP mit den Vorlesungen seines Semesters überschneiden. Untereinander sind Überschneidungen jedoch möglich.

3.8. Gesellschaftswissenschaften

GW Kurse werden von internen und externen Dozenten abgehalten. Im momentanen Planungsablauf werden diese nicht berücksichtigt und erst später hinzugefügt, da sie meist an den Wochenenden oder am späten Nachmittag stattfinden.

3.9. Seminar

Das Seminar unterscheidet sich in seinen Eigenschaften nicht anderen Vorlesungen. Traditionsgemäß findet es am Nachmittag statt.

3.10. Räume

An der HAW gibt es eine Vielzahl von verschiedenen Räumen, wobei jede Veranstaltung andere Ansprüche stellt. Es ist notwendig die vorhandenen Räume und auch die entsprechenden Bedürfnisse zu typisieren, damit eine richtige Zuordnung erfolgen kann. Spezielle Räume müssen dabei speziellen Vorlesungen vorbehalten bleiben, da sonst der bereits limitierte Planungsspielraum dieser Veranstaltungen weiter dezimiert wird.

Typ	Anzahl
Konferenzraum	1
Lernraum	4
Multimediaraum	2
Raum mit Doppel-Tafel	18
PC-Pool	9
Raum mit Beamer	26
Großer Vorlesungsraum (>50 Personen)	6
Vorlesungsraum mit Fenster	12

Tabelle 3.1.: Raumtypen und deren Anzahl [Wei13]

4. Analyse

4.1. Constraints

Bevor die Entwicklung eines Constraint Solver begonnen werden kann, muss festgestellt werden welche Constraints er lösen können soll. Diese bieten die Grundlage für etwaige Lösungsansätze.

Betrachtet man die Anforderungen an die Planung und die Art und Weise wie das Problem gelöst wird, kann man bereits die wichtigsten Constraints extrahieren.

4.1.1. Hard Constraints

C1 Vorlesungen müssen in eine Zeitstruktur gegliedert werden, in der eine Woche 5 Tage umfasst, Montag bis Freitag. Dabei können Vorlesungen vom 1. bis zum 6. Viertel angeboten werden.

C2 Jeder Dozent kann zu einem Zeitpunkt nur eine Vorlesung besuchen.

C3 Jeder Raum darf zu einem Zeitpunkt nur mit einer Vorlesung belegt sein.

C4 Vorlesungen eines Semesters, dürfen sich nicht mit Vorlesungen desselben Semesters im selben Studiengang überschneiden.

C5 Vorlesungen dürfen nur in Räumen stattfinden, die einem vorher definierten Raumtyp entsprechen.

C6 Veranstaltungen dürfen nicht auf mehrere Tage aufgeteilt werden.

Wie man hier sehen kann, sind die meisten Hard Constraints für die Vermeidung von Mehrfachbelegungen, was natürlich daraus resultiert dass es für Personen physikalisch nicht möglich ist an zwei Orten gleichzeitig zu sein. Seien es Studenten die ihrem Semesterplan folgen oder Dozenten die ihre Vorlesungen halten.

4.1.2. Soft Constraints

C7 Vorlesungen sollten nicht am späten Nachmittag (bzw. im 5. und 6. Viertel) stattfinden.

C8 Zwischen zwei Vorlesungen im gleichen Semester sollte keine Lücke entstehen.

C9 Für jeden Dozenten gibt es bestimmte vordefinierte Zeiten die Vorlesungsfrei bleiben sollten.

C10 Für bestimmte Vorlesungen sollte ein Zeitrahmen gesetzt werden können, in die sie bevorzugt stattfinden.

C11 Bevorzugt ein Professor einen bestimmten Raumtyp, sollte das berücksichtigt werden.

Die Art der Soft Constraints richten sich hauptsächlich danach, wie die Vorlesungen gelegt werden. So ist es zum Beispiel organisatorisch ohne weiteres möglich Veranstaltungen auf den späten Nachmittag zu legen, für die Studenten und Dozenten ist das jedoch sehr unschön, da ihre Konzentrationsfähigkeit über den Tag abnimmt.

Ziel der Hard Constraints ist es demnach einen Stundenplan zu beschreiben der physikalisch möglich ist, wohin die Soft Constraints die Qualität desselben für alle beteiligten verbessern soll. Nachdem das Planungsverfahren auf der Zustimmung der Dozenten basiert, dürfen die Soft Constraints unter keinen Umständen außer Acht gelassen werden.

4.2. Variablen und Domänen

Wie bereits erwähnt handelt es sich bei dem hier vorliegendem Problem um ein kombinatorisches. Vorlesungen müssen nach bestimmten Regeln, den Constraints, auf die Ressourcen verteilt werden. Dabei existiert jede Ressource (Semester, Professoren und Räume) zu jedem Zeitpunkt (bzw. Viertel) genau einmal. Dadurch ergibt sich auch die Einsicht das alle Ressourcen etwas gemeinsam haben, die Zeit die für jede von ihnen gleich ist.

Geht man von einer klassischen Modellierung mit einer 1-zu-1 Beziehung zwischen Variablen und Domänen aus, so reicht es nicht jede Ressource als eine Variable zu definieren, da sie nur einen Wert annehmen könnte. Will man das auf alle Zeitpunkte erweitern, müsste man als Beispiel für einen Dozenten, für jeden Zeitpunkt eine Variable haben (30

Stück, für 5 Tage mit 6 Vierteln), jeweils mit den Vorlesungen des Dozenten als Domäne. Jeder dieser Variablen müssten untereinander mit Constraints belegt werden so das keine denselben Wert annehmen darf. Dazu kommen noch die Constraints, welche die Hard Constraints zwischen den vielen verschiedenen Variablen der Dozenten, Räumen und Semestern abbilden würden. In diesem mächtigen Netzwerk aus Variablen ist es natürlich möglich die verschiedensten Constraints zu beschreiben.

Ein Vorteil des hier vorliegenden Problems ist, das es nicht möglich sein muss beliebige Constraints zu definieren. Beschränkt man das Problem auf die bereits erwähnten Constraints kann man das Netz stark reduzieren.

Geht man von der Vorlesung als zentrales Element aus und modelliert die Zeit als gemeinsame Domäne so kann man jede Vorlesung mit einem Set von drei Tupeln beschreiben. Für das Semester (s), den Dozenten (d) und den Raum (r), zusammen mit einer Variable deren Domänen alle möglichen Zeitpunkte enthält zu denen die Veranstaltungen stattfinden kann.

Definition 5 *Gegeben sind:*
$V = v_1, ..., v_n$ *als die Menge aller Vorlesungen.*
$D = d_1, ...d_n$ *als die Menge aller Dozenten.*
$S = s_1, ..., s_n$ *als die Menge aller Semester.*
$R = r_1, ..., r_n$ *als die Menge aller Räume.*
Veranstaltungen $v = ((d \in D, w_1), (s \in S, w_2), (r \in R, w_3)$ *wobei*

$$\begin{aligned} D(w_1) = D(w_2) = D(w_3) = \{ \quad & (1,1),(1,2),(1,3),(1,4),(1,5),(1,6), \\ & (2,1),(2,2),(2,3),(2,4),(2,5),(2,6), \\ & (3,1),(3,2),(3,3),(3,4),(3,5),(3,6), \\ & (4,1),(4,2),(4,3),(4,4),(4,5),(4,6), \\ & (5,1),(5,2),(5,3),(5,4),(5,5),(5,6) \quad \} \end{aligned}$$

Wobei d für den einen Professor steht der die Vorlesung gibt, s für das Semester in dem sie stattfindet und r für einen der Räume.

Eine Veranstaltung kann demnach unter folgenden Bedingungen auf einen Platz gelegt werden.

Definition 6 *Gegeben sei Vorlesung* $v = ((d \in D, w_1), (s \in S, w_2), (r \in R, w_3)$. *Diese darf auf Zeitpunkt* t *gelegt werden, genau dann wenn* $t \subseteq D(w_1)$, $t \subseteq D(w_2)$ *und ein Raum* r *existiert mit* $t \subseteq D(w_3)$.

$$\forall v_1, v_2 \in V : v_1((Dozent_d, w_1), s_1, r_1) \wedge v_2((Dozent_d, w_2), s_2, r_2) \wedge w_1 \neq w_2$$
$$\forall v_1, v_2 \in V : v_1(d_1, (Semester_s, w_1), r_1) \wedge v_2(d_2, (Semester_s, w_2), r_2) \wedge w_1 \neq w_2$$
$$\forall v_1, v_2 \in V : v_1(d_1, s_1, (Raum_r, w_1)) \wedge v_2(d_2, s_2, (Raum_r, w_2)) \wedge w_1 \neq w_2$$

Das bedeutet das eine Vorlesung nur auf einen Zeitpunkt gelegt werden darf, wenn dieser Zeitpunkt in den Domänen der benutzten Ressourcen verfügbar ist. Außerdem muss sichergestellt sein das alle Variablen der gleichen Ressource in allen Veranstaltungen nicht den gleich Wert annehmen dürfen. Eine Belegung die dieser Definition folgt erfüllt alle Hard Constraints um Doppelbelegungen zu vermeiden. Die Struktur der Zeit kann man dynamisch über den Aufbau der Domänen steuern ohne das weitere Constraints notwendig sind.

Will man nun die Soft Constraints hinzunehmen, so lassen sich sie sich durch den Ausschluss bestimmter Werte auf den Variablen selbst definieren.

Als Beispiel die Wochen Domäne w eines Dozenten $d \in D$. In seiner Domäne sind zu beginn sämtliche Zeiten vorhanden.

$$\begin{aligned} w = \{ \quad & (1,1),(1,2),(1,3),(1,4),(1,5),(1,6), \\ & (2,1),(2,2),(2,3),(2,4),(2,5),(2,6), \\ & (3,1),(3,2),(3,3),(3,4),(3,5),(3,6), \\ & (4,1),(4,2),(4,3),(4,4),(4,5),(4,6), \\ & (5,1),(5,2),(5,3),(5,4),(5,5),(5,6) \quad \} \end{aligned}$$

Nehmen wir nun an das der Dozent am Dienstag keine Vorlesungen haben möchte, also das $w \neq (2,1),(2,2),(2,3),(2,4)(2,5),(2,6)$. Definiert man das als einfachen Constraint so würde man zB., wenn man ein klassisches Konsistenz verfahren verwendet, diese Werte aus seiner Domäne entfernen.

$$\begin{aligned} w = \{ \quad & (1,1),(1,2),(1,3),(1,4),(1,5),(1,6), \\ & (3,1),(3,2),(3,3),(3,4),(3,5),(3,6), \\ & (4,1),(4,2),(4,3),(4,4),(4,5),(4,6), \\ & (5,1),(5,2),(5,3),(5,4),(5,5),(5,6) \quad \} \end{aligned}$$

Das hätte zur Folge das dort gar keine Vorlesungen mehr stattfinden kann. An dieser Stelle ist keine Unterscheidung zwischen Hard und Soft Constraints möglich. Orientiert

man sich am Partial Constraint Satisfaction Problem (PCSP) [FW92] so besteht ein PCSP nicht nur aus dem Tupel (V, C), sondern ist um eine Funktion w erweitert, welche jeden Constraint auf ein Gewicht abbildet. Während Hard Constraints an ein unendliches Gewicht gebunden sind, kann es bei den Soft Constraints beliebig feine Unterscheidungen geben.

Beim Überprüfen der vorhandenen Lösung wird das Gewicht aller nicht erfüllten Constraints summiert, welches die Qualität der vorhandenen Lösung widerspiegelt. Die Lösung deren gesamt Gewicht am niedrigsten ist erfüllt die meisten Constraints.

Ein Solver welcher in der Lage sein möchte Soft Constraints zu verarbeiten benötigt also eine Funktion welche Constraints auf ein Gewicht abbildet.

4.3. Veranstaltungstypen

Betrachtet man die Veranstaltungen an der HAW , so kann man diese generell in zwei Typen einteilen, *kontinuierlich* und *periodisch*.

Kontinuierliche Veranstaltungen, wie die normalen Vorlesungen, nehmen ein und denselben Zeitpunkt und Raum jede Woche, über das gesamte Semester ein. Sie kollidieren typischerweise mit anderen kontinuierlichen Veranstaltungen.

Periodische Veranstaltungen hingegen sehen zunächst schwieriger aus. Dabei handelt es sich zB. um Praktika die in mehreren Gruppen alle paar Wochen stattfinden. Theoretisch müsste man jede einzelne Gruppe nehmen und diese nach und nach in den Plan einfügen. Schaut man sich jedoch das typische Praktikum an der HAW an, mit 3 Gruppen die jeweils 3 Wochen zwischen den einzelnen Praktika haben, kann man eine weitere Entdeckung machen.

Woche 1	Woche 2	Woche 3	Woche 4	Woche 5	Woche 6
Gruppe 1	Gruppe 2	Gruppe 3	Gruppe 1	Gruppe 2	Gruppe 3

Tabelle 4.1.: Beispiel mit 3 Gruppen

In Tabelle 4.1 kann man sehen das sich ein gleichmäßiger Fluss ergibt, in der das Praktikum jede Woche am gleichen Zeitpunkt stattfindet. Betrachtet man die Veranstaltung als eine Einheit kann man sie aus Planungssicht als kontinuierliche Veranstaltung planen, da sie über den gesamten Zeitraum einen Raum und einen Dozenten belegen.

Definition 7 ***Intervall****: Die Anzahl der Wochen zwischen den Veranstaltungen einer Gruppe.*

Was uns zu folgender Definition bringt.

Definition 8 *Eine Veranstaltung ist kontinuierlich, wenn ihre Gruppenanzahl gleich ihrem Intervall ist.*

Natürlich gibt es auch Veranstaltung mit mehr oder weniger Gruppen, siehe Tabelle 4.2 und Tabelle 4.3, die nicht außer Acht gelassen werden dürfen. Ein Solver der diese Vereinfachung nutzen will, sollte in der Lage sein periodische Veranstaltungen zu kontinuierlichen zu bündeln. Wobei die Summe der Gruppen, geteilt durch den Intervall, der Anzahl der kontinuierlichen Veranstaltungen entspricht. Für alle Gruppen die nicht in dieses Schema fallen müssen eigene Plätze im Stundenplan gefunden werden.

Woche 1	Woche 2	Woche 3	Woche 4	Woche 5	Woche 6
Gruppe 1	Gruppe 2	-	Gruppe 1	Gruppe 2	-

Tabelle 4.2.: Beispiel: 2 Gruppen

Woche 1	Woche 2	Woche 3	Woche 4	Woche 5	Woche 6
Gruppe 1	Gruppe 2	Gruppe 3	Gruppe 1	Gruppe 2	Gruppe 3
		Gruppe 4			Gruppe 4

Tabelle 4.3.: Beispiel: 4 Gruppen

Betrachtet man die Praktika aus Sicht der Studenten so besucht in der Regel jeder Student nur jeweils eine Gruppe der genannten Praktika. Jedes Praktika einzeln für sich auf einen Platz im Semester zu legen Besucht ein Student Gruppe 1 des Praktikums A, so kann er in der darauf Folgenden Woche Gruppe 2 des Praktikums B besuchen, veranschaulicht in Tabelle 4.4.

	Woche 1	Woche 2	Woche 3	Woche 4	Woche 5
Praktikum A	**Gruppe 1**	Gruppe 2	Gruppe 3	**Gruppe 1**	Gruppe 2
Praktikum B	Gruppe 1	**Gruppe 2**	Gruppe 3	Gruppe 1	**Gruppe 2**
Praktikum C	Gruppe 1	Gruppe 2	**Gruppe 3**	Gruppe 1	Gruppe 2

Tabelle 4.4.: Ein Student kann in jeder Woche jeweils eine Gruppe eines Praktikums besuchen.

Definition 9 *Verschiedene Praktika dürfen sich im selben Semester überschneiden, wenn sie demselben Intervall folgen und die Anzahl der Praktika den Intervall nicht überschreitet.*

Auf ähnliche Weise kann man die Lücke in Tabelle 4.2 schließen. Dort befinden sich zwei Gruppen in einem Raum, der jeweils in der Woche nach der zweiten Gruppe leer steht. Auch hier kann man durch Betrachtung des Intervalls überprüfen ob eine weitere Gruppe hinein passt.

	Woche 1	Woche 2	Woche 3	Woche 4	Woche 5	Woche 6
Praktikum A	Gruppe 1	Gruppe 2	-	Gruppe 1	Gruppe 2	-
Praktikum B	-	-	Gruppe 1	-	-	Gruppe 3

Tabelle 4.5.: Beispiel: 2 Gruppen

Definition 10 *Ein Raum kann zur selben Zeit von verschiedenen Praktika belegt werden, wenn ihre Gruppenanzahl den Intervall nicht überschreiten.*

4.4. Minimizing Conflicts Hill-Climbing

Der 1993 in „Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems“ beschriebene Algorithmus entstand bei der Untersuchung des Guarded Discrete Stochastic Network (GDS), einem Neuralen Netzwerk welches konstruiert wurde um die vielen Aufgaben des Hubble Space Telescope zu planen. Eine Aufgabe die mit traditionellen Mitteln nicht bewältigt werden konnte, wurde vom GDS gemeistert. Minton u. a. fanden bei ihren Untersuchungen heraus das GDS nicht systematisch an das Problem heran geht.

Sie beschreiben zwei Probleme der systematischen Tiefensuche die so vermieden werden. So geht viel Kapazität durch eine schlecht geplante Suche verloren, da schlechte Entscheidungen mit Backtracking rückgängig gemacht werden müssen. Ein Problem welches sich unter Umständen durch eine vorgeschaltetes Value-Ordering lösen lassen würde. Das Zweite Problem entsteht, wenn mögliche Lösungen eng beieinander liegen.

Klassische Methoden, wie Tiefensuche, eignen sich für Probleme bei denen die möglichen Lösungen gleichmäßig über den Suchbaum verteilt sind. Egal für welchen Teilbaum man sich entscheidet, die Wahrscheinlichkeit eine Lösung darin zu finden ist hoch. Sind die Lösungen jedoch in wenigen Teilbäumen konzentriert, besteht die Möglichkeit das die Tiefensuche viele Teilbäume erschöpfend durchsuchen muss.

Im ersten Schritt des Minimizing Conflicts Hill Climbing Algorithmus wird eine vollständige, jedoch inkonsistente Lösung erzeugt. Danach beginnt der Algorithmus nach und nach diese zu verbessern. Dazu werden die Variablen die Konflikte erzeugen gesammelt und von diesen wird eine zufällig ausgewählt um sie mit einem neuen Wert zu

belegen. Dieser Wert wird aus allen möglichen Belegungen so ausgewählt, das danach entweder keine oder weniger Konflikte vorhanden sind. Stehen mehrere mögliche Werte zur Auswahl, wird zufällig entschieden.

```
algorithmus MIN-CONFLICTS
  eingabe:    csp
                 max_schritte, eine maximale Anzahl von Schritten
                 zustand, eine Initial-Belegung für die Variablen des
   CSP
  ausgabe: eine Lösung für das csp oder Misserfolg
  for i=1 to max_schritte do
      if zustand ist eine Lösung für csp then return zustand
      var <-- eine zufällige Variable die Konflikte erzeugt
      wert <-- ein Wert w der die Konflikte für var reduziert
      set var = wert
  return Misserfolg
```

Listing 4.1: Min Conflict Hill Climbing Pseudo Code

Es kann vorkommen das der Algorithmus, wie bei allen Hill Climbing Techniken, in einem lokalen Minimum stecken bleibt. Ein Weg dem zu entkommen ist ein Neustart der Suche [Bea+04].

Ob der Algorithmus optimal arbeitet, hängt stark von der Initial-Belegung ab mit der er startet. Sind dort schon weniger Konflikte vorhanden kann eine vollständige Lösung natürlich einfacher gefunden werden. Um das zu erreichen muss man entweder versuchen mit einem kleineren Set von Constraints eine möglichst gute Belegung zu erstellen oder man lernt aus der Vergangenheit. Bei dem uns vorliegenden Problem, der Veranstaltungsplanung an der HAW, haben wir eine lange Historie von validen Vorlesungsplänen auf man aufbauen könnte. So könnte man mit einen Veranstaltungsplan auswählen der mit den zur Zeit bestehenden Anforderungen am wenigstens in Konflikt steht und versuchen diesen zu reparieren.

5. Realisierung

5.1. Konzept

Um sich das Problem einfacher begreiflich zu machen kann man es sich als abgewandeltes n-queens Problem vorstellen.

Beim n-queens Problem geht es darum 8 Damen auf einem 8x8 Schachfeld zu platzieren, ohne das diese sich gegenseitig schlagen können. Ähnlich verhält es sich auch bei dem Stundenplan Problem der HAW. Anstatt eines Schachfeldes haben wir unsere Wochenstruktur, auf der wir versuchen unsere Veranstaltungen, wie Figuren, nach verschiedenen Regeln zu platzieren.

Einer der Unterschiede ist, das wir nicht nur ein Schachfeld haben, sondern sehr viele. Für jede Ressource eins.

Die Wochen sind zweidimensionale Arrays, jeder dieser Integer-Werte, zu sehen in Abbildung 5.2, gibt an wie verfügbar die Ressource zu dem jeweiligen Zeitpunkt ist. Momentan wird durch jeden Wert jeweils ein Viertel abgebildet, möglich sind beliebige Strukturen wie zB. einen Eintrag pro Minute zu haben. Die Realisierung als Arrays bzw. als Tabellen erlaubt es nicht nur schnell, sondern auch homogen auf die Werte zuzugreifen.

Der Algorithmus startet mit einer initial Belegung, in der alle Vorlesungen zufällig auf dem Stundenplan verteilt werden. Nachdem diese Art der Belegung natürlich nicht frei von Kollisionen ist, beginnt der Algorithmus den Stundenplan zu reparieren. Er wählt bei jedem Schritt eine Vorlesung aus, die in Konflikt mit anderen Vorlesungen steht und sucht für sie einen neuen Platz. Dabei schaut er in die Woche jeder Ressource (Vorlesung, Semester, Dozenten, sowie alle in Frage kommenden Räume) und listet alle möglichen Plätze auf, in dem er für jede Zeitpunkt/Raum Kombination die vorhandenen Werte addiert.

Die Ergebnisse werden nach ihren Werten sortiert und der Algorithmus wählt eine der Möglichkeiten mit dem niedrigsten Wert aus, dadurch kann sichergestellt werden das die neue Belegung gleich viele oder weniger Konflikte verursacht als die ursprüngliche. Anschließend modifiziert er die Wochen der involvierten Ressourcen entsprechend der

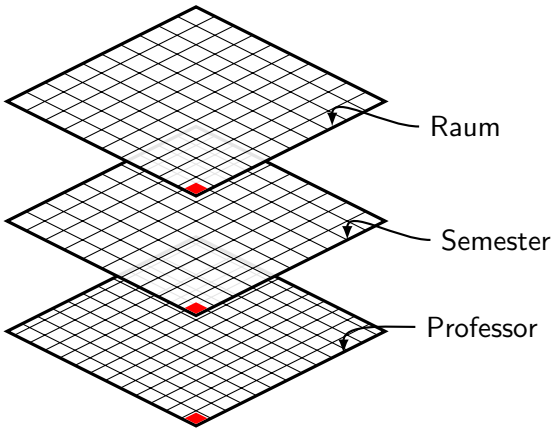


Abbildung 5.1.: Veranschaulichung des n-queen Beispiels in der jede Ressource hat Ihr eigenes Brett hat. Betrachtet man einen möglichen Platz für eine Veranstaltung, so schaut man sich alle übereinander liegenden Felder an Felder.

```
{
{ 0, 0, 0, 0, 0, 0 }
{ 0, 0, 0, 0, 0, 0 }
{ 0, 0, 0, 0, 0, 0 }
{ 0, 0, 0, 0, 0, 0 }
{ 0, 0, 0, 0, 0, 0 }
{ 0, 0, 0, 0, 0, 0 }
}
```

Abbildung 5.2.: Woche als Array

```
{
{ 0, 0, 0, 0, 0, 0 }
{ 0, 0, 0, 0, 0, 0 }
{ 0, 0, 0, 0, 0, 0 }
{ 0, 0, 0, 0, 0, 0 }
{ 1, 1, 1, 1, 1, 1 }
{ 1, 1, 1, 1, 1, 1 }
}
```

Abbildung 5.3.: Woche mit Soft Constraint

neuen Belegung.

Hard und Soft Constraints werden dabei als Zahlenwerte in die Arrays geschrieben. Liegt in einem Feld eine Vorlesung, so wird dieser Platz mit einer 2 belegt. Will man beispielsweise den Soft Constraint hinzufügen, nachdem keine Vorlesungen in den Nachmittag gelegt werden darf, so belegt man bereits bei der Initialisierung alle entsprechenden Felder im Semester mit einer 1, siehe Abbildung 5.3.

Beim Sammeln der möglichen neuen Plätze für eine Veranstaltung, rücken die Plätze bei denen Soft Constraints verletzt werden würden automatisch weiter nach hinten und werden erst benutzt wenn keine andere Möglichkeit mehr besteht.

Auf diese Weise kann man nicht nur bequem Soft Constraints hinzufügen, man könnte die verschiedenen Constraints bei Bedarf auch beliebig gewichten.

Um festzustellen ob eine vorhandene Planung valide ist, reicht es zu überprüfen ob irgendein Wert in den Wochen-Arrays eine festgelegte Höchstgrenze überschreitet. In der momentanen Implementation ist dieser Wert 3, die Summe aus einem Hard (2) und einem Soft Constraint (1). Dieses Verfahren funktioniert nicht nur sehr schnell, es erlaubt auch über eine Bewertungsfunktion die verschiedenen Spezialfälle zu behandeln.

Alle zur Verfügung stehenden Ressourcen können in einem XML Dokument festgehalten werden, in dem man mit einem Set aus Attributen verschiedene Constraints definieren kann. Dieses Dokument stellt die Schnittstelle nach außen dar und kann von anderen Programmen, wie zum Beispiel Grafischen Nutzeroberflächen, eingelesen werden.

Der Algorithmus ist in Lua geschrieben und teilt sich in 3 Module auf: der Auswahl, der Bewertung und dem XML Parser/Writer.

5.2. Lua

Nachdem Geschwindigkeit einer der Hauptschwerpunkte eines Constraint Solver ist, sollte dieser in einer Sprache implementiert werden die in der Lage ist diese Anforderung zu erfüllen.

Nach einigen Vergleichen fiel die Wahl auf Lua [1] [2] Lua ist eine in C geschriebene funktionale dynamische Skriptsprache, entwickelt an der Universität von Rio de Janeiro in Brasilien. Es ist eine leichtgewichtige, schnelle Sprache die sich, dank Erzeugung von Bytecode, plattformübergreifend nutzen lässt.

Sie ist dazu konzipiert als Skriptsprache in größere Projekte integriert zu werden und wird dafür auch in vielen industriellen Bereichen erfolgreich eingesetzt.

[1] Lua ist kein Akronym, es ist das portugiesische Wort für Mond.
[2] http://www.lua.org/

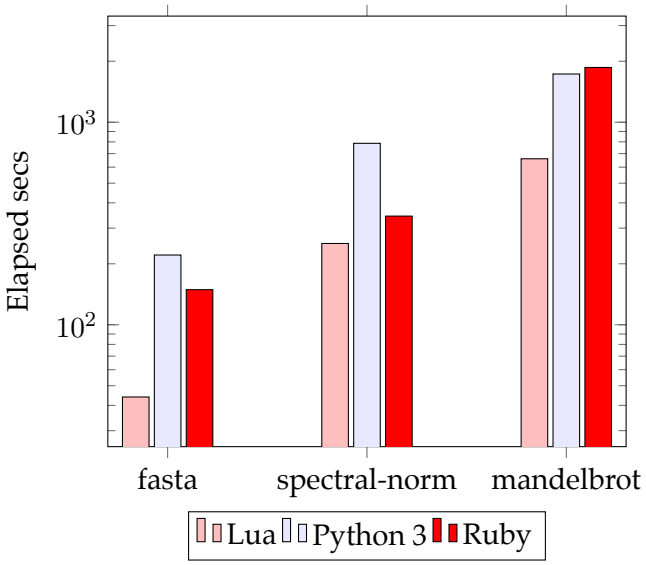


Abbildung 5.4.: Laufzeitvergleich im Vergleich

Einige Beispiele für Anwendungen die Lua nutzen:

prosody ist ein XMPP-Server der in Lua geschrieben ist.

Adobe Photoshop Lightroom nutzt Lua für sein Interface.

Apache HTTP Server unterstützt die Ausführung von Lua im request process.

Cisco nutzt Lua für seine Dynamic Access Policies, zur Konfiguration von VPNs.

iClone ein 3D Echtzeit Animationssoftware nutzt Lua zur Physik Simulation.

FarCry ein Spiel aus dem Hause Crytek/UbiSoft nutzt Lua für alle Spiel internen Events und KI-Logik.

Nmap scripting language nutzt den Lua interpreter mit zusätzlichen Bibliotheken als Interface für Nmap.

Weitere Beispiele finden sich unter: www.lua.org/uses.html

Besonders in der Spielindustrie ist Lua wegen seiner Geschwindigkeit und der leichten Integration beliebt, so kann man aus C und Lua heraus Funktionen der jeweils andere Programmiersprache nativ aufrufen [IFC07].

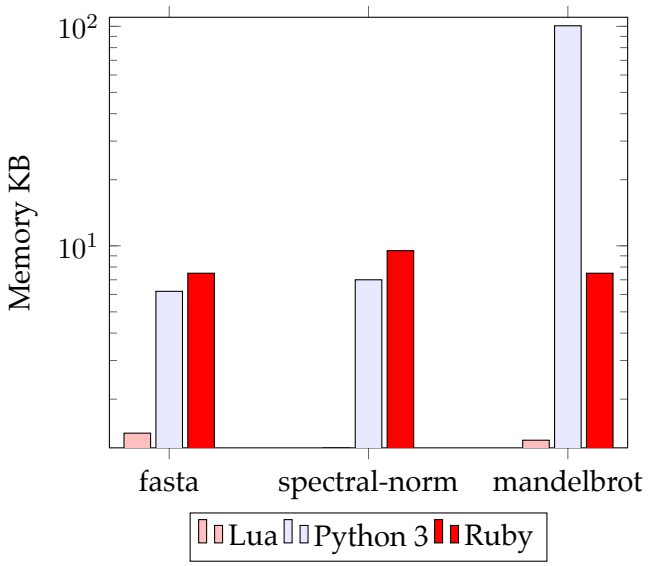


Abbildung 5.5.: Speicherverbrauch im Vergleich

Vergleicht man Lua mit anderen Skriptsprachen anhand von Microbenchmarks, wie in Abbildung 5.4 [3], lässt dies darauf schließen das Lua einen Geschwindigkeitsvorteil gegenüber anderen Skriptsprachen verspricht. Ein weiterer Grund Lua zu nutzen ist der geringe Speicherverbrauch, Abbildung 5.5 zeigt das Lua eine leichtgewichtige Sprache ist und von sich aus ohne viel Overhead auskommt.

Als zusätzliche Verbesserung gibt es den von Mike Pall seit 2005 entwickelten LuaJIT [4], den Lua Just In Time Compiler. Dabei führt LuaJIT viele Optimierungen, wie das Erzeugen von Maschinencode und eigenem Speichermanagement durch, was die Performance besonders bei längerer Laufzeit verbessert.

Lua (ab Version 5) und LuaJIT steht unter der MIT-Lizenz [5] und kann daher ohne Einschränkungen gemäß der Lizenz genutzt werden.

5.3. Variablen Auswahl

Die Hauptaufgabe der Variablen Auswahl ist es eine der vorhandenen Veranstaltungen, die in Konflikt mit anderen Veranstaltungen steht, auszuwählen und dafür einen neuen Platz zu finden. Hier kommt das breits erwähnte Hill Climbing zum Einsatz, das von

[3] Quelle: http://benchmarksgame.alioth.debian.org
[4] http://luajit.org/
[5] siehe: http://opensource.org/licenses/mit-license.php

der Bewertungsfunktion unterstützt wird.

```
local conflicts = find_conflicts()
local v = random_lecture(conflicts)
list = {}
for r in pairs(rooms) do
  for d=1, wochedays do
    for q=1, quater do
      conflicts = hits(v, d, q, r)
      insert = false
      if not (list[1]) then
        insert = true
      elseif (conflicts < list[1].c) then
        insert = true
        auswahl = {}
      elseif (conflicts == list[1].c) then
        insert = true
      end
      if (insert) then
        item = {}
        item.c = conflicts
        item.r = r
        item.d = d
        item.q = q
        table.insert(list, item)
      end
    end
  end
end

local random = math.random(#list)
change_timespot(v, list[random].d, list[random].q, list[random]
   .r)
```

Ein grundlegender Vorteil der Entkopplung von Auswahl und Bewertung ist, das man den Solver-Algorithmus einfach austauschen kann. Bei Bedarf könnte man das hier eingesetzte Hill Climbing gegen einen systematischen Ansatz austauschen. Die einzige

Bedingung ist, das sich der Algorithmus an der Sortierung der Werte orientiert.

5.4. Bewertung

Das eigentliche Herzstück des Algorithmus ist die Bewertungsfunktion. Hier werden die Wochen der Ressourcen betrachtet und je nach Constraints dynamisch die Werte angepasst.

Als Beispiel kann jeder Professor eine Präferenz liste mit möglichen Raumtypen besitzen, die er bevorzugt.

```
local right_room_type = false
for i, rtyp in ipairs(v.roomtyp) do
  if (rooms[r].roomtyp == rtyp) then
    right_room_type = true
    break
  end
end
if not (right_room_type) then return 10 end

right_room_type = false
for i, p in ipairs(v.professor) do
  for i, rtyp in ipairs(staff[p].roomtyp) do
    if (rooms[r].roomtyp == rtyp) then
      right_room_type = true
      break
    end
  end
  if not (right_room_type) then nr_conflicts = nr_conflicts + 1
    end
end
```

Listing 5.1: Bewertung von Räumen

In Listing 5.1 sieht man den Teil des Codes der das erledigt. Hier wird jeder Raum der in Frage kommt bewertet, einmal aus der Sicht der Veranstaltung (v) und aus der Sicht des Dozenten (p). Nachdem festgestellt wurde ob die Vorlesung in diesem Raum stattfinden darf, wird die Präferenz liste für jeden Dozenten in dieser Vorlesung angeschaut. Ist der

Raumtyp darauf nicht vorhanden, so wird eine 1 zu der Bewertung hinzu addiert. Die jeweilige Zeit/Raum Kombination rutscht demnach beim Sortieren weiter nach hinten und Räume auf der Präferenz liste werden automatisch bevorzugt. Hier sieht man auch das die Veranstaltung-Raum Beziehung viel höher gewichtet ist als die Dozent-Raum Beziehung.

Sollte der Fall eintreten das die Räume die der Dozent bevorzugt gar nicht in Frage kommen, wie zB. wenn eine Veranstaltung in einem PC Pool stattfinden muss und für den Dozenten Seminarräume mit Fenstern eingetragen sind, so ändert das an der Bewertung nichts. Für die eigentliche Auswahl spielt nur die sortierte Reihenfolge eine Rolle, die absolut gleich bleibt wenn für alle Räume dasselbe auf die Bewertung addiert wird.

```
for i, p in ipairs(v.professor) do
  for i=0, v.length-1 do
    nr_conflicts = nr_conflicts + staff[p].week[day][slot+i].
   value
  end
end
```

Listing 5.2: Bewertung von Räumen

```
for i=0, v.length-1 do
  for s=1, #v.semester do
    local number = 1
    for k=1, #semester[v.semester[s]].week[day][slot+i].events
   do
      event = semester[v.semester[s]].week[day][slot+i].events[
   k]
      if not (event.name == v.name) then
        if not (event.typ == v.ignore) then
          if (event.typ == v.parallel and number <= v.intervall
    and event.intervall == v.intervall) then
            number = number+1
          else
            nr_conflicts = nr_conflicts + semester[v.semester[s
   ]].week[day][slot+i].value
          end
```

```
13          end
14        end
15      end
16    end
17  end
```

Listing 5.3: Bewertung von Vorlesungen

Beim einfachen Fall, zu sehen in Listing 5.2, wird für jeden Professor der der Vorlesung zugeteilt ist, der Wert jedes seiner Viertel das die Vorlesung belegen würde zu einer Gesamtbewertung hinzu addiert. Da es bei den Vorlesungen einige Spezialfälle gibt ist die Bewertung die in Listing 5.3 zu sehen ist etwas aufwendiger, folgt aber derselben grundlegenden Logik. Die beiden Spezialfälle die hier abgehandelt werden, sind einmal die Möglichkeit, dass Veranstaltungen sich ignorieren dürfen oder parallel laufen können. Dabei werden die Regeln angewandt die in Abschnitt 4.3 beschrieben sind.

5.5. XML

Um das vorhandene Problem zu beschreiben und die Lösung dafür zu speichern wurde die XML-Beschreibungssprache gewählt. Die Daten aller Ressourcen werden in einer Datei festgehalten, zusammen mit den verbundenen Constraints.

Beim Konzipieren der Struktur wurde versucht die Beschreibung des Problems so einfach und intuitiv wie möglich zu halten, um auch Benutzern ohne Informatik Kenntnisse den Zugang zu erleichtern.

Hat der Solver eine Planung abgeschlossen so schreibt er die Datei erneut und hält in den Attributen der Veranstaltungen das Ergebnis des Planungsvorganges fest. Dieser Ansatz wurde gewählt damit Ergebnis und Problem nicht voneinander getrennt werden.

Sollte ein anderes Programm die Datei einlesen, so kann es entweder nur die Lösung extrahieren oder auch das Problem mit verarbeiten, um es bei weiteren etwaigen Verarbeitungsschritten nutzen zu können.

Im Nachfolgenden werden die einzelnen Bestandteile des XML Dokuments beschrieben.

5.5.1. Wochenstruktur

Am Anfang wird die Wochenstruktur beschrieben, man kann festlegen wie viele Tage und wie viele Zeiteinheiten der Planung zur Verfügung stehen.

```
<wochenStruktur>
    <tag lange="7">
      <zeiteinheit wert="0"/>
      <zeiteinheit wert="0"/>
      <zeiteinheit wert="0"/>
      <zeiteinheit wert="0"/>
      <zeiteinheit wert="1"/>
      <zeiteinheit wert="1"/>
    </tag>
    <tag lange="7">
      <zeiteinheit wert="0"/>
      <zeiteinheit wert="0"/>
      <zeiteinheit wert="0"/>
      <zeiteinheit wert="0"/>
      <zeiteinheit wert="1"/>
      <zeiteinheit wert="1"/>
    </tag>
</wochenStruktur>
```

Listing 5.4: Beispiel für die Wochen Struktur mit 2 Tagen und je 7 Zeiteinheiten

Aus dieser Struktur speisen sich alle Wochen der folge Ressourcen und man kann hier bereits allgemeingültige Werte eintragen. In Listing 5.7 kann man sehen, das für alle Viertel am Nachmittag bereits eine 1 eingetragen ist.

Will man Modifikationen an den spezifischen Wochen der verschiedenen Ressourcen vornehmen, kann man das Element 'woche' nutzen. Darin gibt man den Tag, die entsprechende Zeiteinheit bzw. das Viertel an und abschließend den Wert den man in diesem Feld stehen haben möchte.

```
<woche tag="1" slot="3" wert="1"/>
```

Listing 5.5: Beispiel für ein 'woche' Element

5.5.2. Dozent

Neben den Vorlesungen ist der Dozent der einzige der das Element 'raumtyp' kennt. Hier kann man angeben welche Raumtypen der jeweilige Dozent bevorzugt.

```
<dozent name="Klauck">
    <woche tag="1" slot="3" wert="1"/>
    <raumtyp name="Fensterraum"/>
</dozent>
```

Listing 5.6: Beispiel für den Eintrag eines Dozenten

5.5.3. Vorlesung

```
<vorlesung name="PR1" typ="Vorlesung" intervall="1" dauer="2"
   beginInWoche="1" lock="false" day="5" slot="4" raum="RS11">
    <dozent name="SML"/>
    <woche tag="1" slot="3" wert="1"/>
    <raumtyp name="Seminar"/>
    <semester name="Semester 1"/>
    <parallel name="nil"/>
    <ignore name="nil"/>
</vorlesung>
```

Listing 5.7: Beispiel für den Eintrag der PR1 Vorlesung

5.5.4. Attribute

name Der Name der Vorlesung und der eindeutige Schlüssel mit dem sie referenziert werden kann.

typ Typ der Veranstaltung, wie zB. 'Vorlesung' oder 'Praktika'.

intervall Anzahl der Wochen in denen sich die Vorlesung wiederholt.

dauer Anzahl der Viertel über die sich die Veranstaltung erstreckt.

beginInWoche Wochenzahl in der die Vorlesung beginnen soll.

lock Ist dieser Wert true gesetzt, soll der Solver diesen Eintrag nicht mehr verändern und übernimmt die vorhandene Belegung.

day & slot & raum Der Tag, das Viertel und der Raum auf dem die Veranstaltung bei der letzten Lösung lag. Diese Werte werden aktualisiert wenn der Solver einen neuen Stundenplan erstellt.

5.5.5. Elemente

dozent Eindeutige Referenz des Dozenten.

woche Hier kann die Wochendomäne der Veranstaltung modifiziert werden.

raumtyp Raumtypen in denen diese Veranstaltung stattfinden kann.

semester Semester mit denen keine Kollisionen vorkommen dürfen.

parallel Veranstaltungstypen bei denen die Veranstaltungen parallel verlaufen dürfen. Beispiel: Praktika.

ignore Veranstaltungstypen die bei der Planung ignoriert werden können. Beispiel: Wahlpflichmodule.

5.6. Laufzeit

Zur Messung der Laufzeit wurden 100 Stundenpläne für den Studiengang Angewandte Informatik erstellt, jeweils mit zufälliger Initialbelegung.

	Lua	Luajit
Max ms (Iterationen)	3968 (1524)	16494 (1470)
Min ms (Iterationen)	478 (214)	2469 (218)
Avg ms (Iterationen)	1232 (499)	6192 (484)

Tabelle 5.1.: Laufzeit Vergleich in ms

Die durchschnittliche Laufzeit zum Erstellen eines Stundenplanes beträgt unter Verwendung von Luajit 1232m, während Lua durchschnittlich 6192ms benötigt.

6. Fazit und Ausblick

Hauptziel dieser Arbeit war es einen speziellen Constraint Solver zu konstruieren der in der Lage ist Stundenpläne für Hochschulen zu erstellt. Dafür wurden die Stundenpläne an der HAW analysiert um Constraints zu identifizieren die einen solchen Stundenplan beschreiben. Dabei wurden Erkenntnisse gesammelt, wie das vorhandene Problem vereinfacht werden kann, ohne dabei wichtige Informationen zu verlieren. Ein Benutzer kann in einer XML Datei flexibel seine Ressourcen und Veranstaltungen eintragen und hat dabei ein Set von leicht verständlichen Constraints zur Auswahl.

Zudem wurde eine Grundlage geschaffen in der gewichtete Zeit- oder Attribut-Basierte Soft Constraints flexibel angewandt und weitere einfach implementiert werden können.

Durch die Trennung von Variablen Auswahl und Bewertung ist es möglich einzelne Teile im System auszutauschen oder diese an anderer Stelle wiederzuverwenden. So ist das eingesetzte Hill Climbing auf Geschwindigkeit und Variantenreichtum ausgelegt, es soll den Planer schnell mit vielen verschiedenen Lösungsvorschlägen versorgen können. Wenn man sich die durchschnittliche Laufzeit von 1232ms pro Planungsvorgang anschaut, ist von der Laufzeit gesehen noch genug Spielraum um ggf. auch andere Algorithmen einsetzen zu können. Auch könnte das Bewertungsverfahren in einer grafische Nutzeroberfläche zum Einsatz kommen, um den Benutzer beim manuellen Anpassungen direkt auf Kollisionen hinzuweisen.

Als mögliche Erweiterung das kann das vorhandene System auf eine Minutengenaue Planung erweitern werden. Dafür könnte man noch weitere Constraints hinzufügen, wie zB das Vorlesungen über Pausen hinweg oder auch so aufgeteilt werden dürfen das sie vor und nach einer Pause stattfinden.

Ferner ist es möglich qualitativ hochwertige Stundenpläne aus der Vergangenheit in die vorhandene XML Form zu übertragen und dem Algorithmus als Initial-Belegung zur Verfügung zu stellen. Ist eine entsprechende Datenbank mit alten Stundenplänen aufgebaut, kann der Algorithmus eine Belegung auswählen die am wenigsten Kollisionen verursacht. Auf diese weise würden sich neu erstellte Stundenpläne an alten orientieren.

Selbst wenn die autonom erstellten Stundenpläne die theoretischen Rahmenbedienungen erfüllen, so hat jede Hochschule ihre eigenen Regeln und Verfahren. Auch an der

HAW muss vor einem Praxiseinsatz eine weitere Evaluation stattfinden um das Verfahren noch weiter zu verfeinern und seine Nützlichkeit steigern zu können. Dazu kommt das die Menschliche Komponente immer ein manuelles anpassen notwendig machen wird, so das das hier vorgestellte Programm erst dann sein vollen nutzen entfalten kann, wenn es mit einer entsprechenden grafischen Nutzeroberfläche ausgestattet wird.

Das im Rahmen dieser Arbeit entstandene Programm ist die Basis für ein Assistenz-System, das die Entscheidungsfindung beim erstellen von Stundenplänen stark unterstützen kann.

7. Abkürzungsverzeichnis

CP Constraint Programming

CSP Constraint Satisfaction Problem

DCSP Dynamic Constraint Satisfaction Problem

FCSP Flexible Constraint Satisfaction Problem

PCSP Partial Constraint Satisfaction Problem

GT Generate and Test

BT Backtracking

BJ Backjumping

BC Backchecking

FC Forward Checking

PLA Partial Look-Ahead

DAC Directional Arc Consistency

MC Min-Conflict

TS Tabu Search

COP Constraint-Optimisation-Problem

BnB Branch and Bound

PCS Partial Constraint Satisfaction

FH Fachhochschule

HAW Hochschule für Angewandte Wissenschaften Hamburg

SWS Semesterwochenstunden

WP Wahlpflichtmodul

GW Gesellschaftswissenschaften

GDS Guarded Discrete Stochastic Network

8. Bibliografie

[Bar99] R. Barták. „Constraint Programming: In Pursuit of the Holy Grail“. In: *Proceedings of the Week of Doctoral Students* Part IV (1999), Part IV.

[Bea+04] Matthew Beaumont u. a. „Solving Over-Constrained Temporal Reasoning Problems Using Local Search.“ In: *PRICAI*. Bd. 3157. Springer, 2004, S. 134–143.

[BFW92] Alan Borning, Bjorn Feldman-Benson und Molly Wilson. „Constraint Hierarchies“. In: *LISP AND SYMBOLIC COMPUTATION*. 1992, S. 48–60.

[BS11] Roman Barták und Miguel A. Salido. „Constraint satisfaction for planning and scheduling problems“. In: *Constraints* 16.3 (2011), S. 223–227.

[CB11] Xinguang Chen und Peter van Beek. „Conflict-Directed Backjumping Revisited“. In: *CoRR* abs/1106.0254 (2011).

[DFP96] D. Dubois, H. Fargier und H. Prade. „Possibility Theory in Constraint Satisfaction Problems: Handling Priority, Preference and Uncertainty“. In: *Applied Intelligence* 6 (1996), S. 287–309.

[DP98] Didier Dubois und Henri Prade. *Flexible Constraints and Qualitative Decision in A.I. (Extended Abstract)*. 1998.

[Fre97] Eugene C. Freuder. „In Pursuit of the Holy Grail“. In: *Constraints* 2.1 (1997), S. 57–61.

[FW92] Eugene C. Freuder und Richard J. Wallace. *Partial Constraint Satisfaction*. 1992.

[HE79] Robert M. Haralick und Gordon L. Elliott. „Increasing tree search efficiency for constraint satisfaction problems“. In: *Proceedings of the 6th international joint conference on Artificial intelligence - Volume 1*. Morgan Kaufmann Publishers Inc., 1979, S. 356–364.

[IFC07] Roberto Ierusalimschy, Luiz Henrique de Figueiredo und Waldemar Celes. „The Evolution of Lua“. In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. ACM, 2007, pages.

[Kum92] Vipin Kumar. „Algorithms for Constraint-satisfaction Problems: A Survey“. In: *AI Mag.* 13.1 (1992), S. 32–44.

[Mac77] Alan Mackworth. „Consistency in Networks of Relations“. In: *Artificial Intelligence* 8.1 (1977). Reprinted in <I>Readings in Artificial Intelligence</I>, B. L. Webber and N. J. Nilsson (eds.), Tioga Publ. Col., Palo Alto, CA, pp. 69-78, 1981. This paper was honoured in <I>Artificial Intelligence</I> 59, 1-2, 1993 as one of the fifty most cited papers in the history of Artificial Intelligence. It also received the 2013 AIJ Classic Paper Award: http://www.journals.elsevier.com/artificial-intelligence/news/announcing-winners-of-the-2013-aij-classic-paper-award/, S. 99–118.

[Min+92] Steven Minton u. a. „Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems“. In: *Artif. Intell.* 58.1-3 (1992), S. 161–205.

[Mon04] Ugo Montanari. „Networks of constraints: Fundamental properties and applications to picture processing.“ In: *Inf. Sci.* 7 (24.2004), S. 95–132.

[Pro93] Patrick Prosser. „Hybrid Algorithms for the Constraint Satisfaction Problem“. In: *Computational Intelligence* 9 (1993), S. 268–299.

[RM02] Hana Rudová und Keith S. Murray. „University Course Timetabling with Soft Constraints“. In: *PATAT*. Bd. 2740. Springer, 2002, S. 310–328.

[Run06] Wolfgang Runte. „YACS: Ein hybrides Framework für Constraint-Solver YACS: Ein hybrides Framework für Constraint-Solver zur Unterstützung wissensbasierter Konfigurierung“. Magisterarb. Universität Bremen Fachbereich Mathematik / Informatik, 2006.

[Tsa93] Edward P. K. Tsang. *Foundations of constraint satisfaction*. Academic Press, 1993, S. I–XVIII, 1–421.

[VS94] Gérard Verfaillie und Thomas Schiex. „Solution Reuse in Dynamic Constraint Satisfaction Problems“. In: 1994, S. 307–312.

[Wal96] Richard J. Wallace. „Enhancements of Branch and Bound Methods for the Maximal Constraint Satisfaction Problem.“ In: *AAAI/IAAI, Vol. 1*. AAAI Press / The MIT Press, 1996, S. 188–195.

[Wei13] Eduard Weigandt. „Modellierung und Visualisierung einer Constraint-basierten Stundenplanung für Hochschulen“. Bachlorarbeit. HAW Hamburg, 2013.

A. CD

A.1. Inhalte

Auf der CD sind folgende Inhalte:

- Diese Arbeit im PDF-Format
- Quellcode des Solvers

A.2. Aufbau der CD

```
/
├── ba
│   └── Bachelorarbeit_J_Schaa.pdf
└── src
    ├── min_conflict.lua
    ├── ressourcen.xml
    ├── xmlparser.lua
    └── xmlwriter.lua
```

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

Hamburg, 24. März 2014 Jens Schaa