Optimierung und Regularisierung von Constraint Satisfaction-Problemen (CSPs)

Von der Fakultät 1 - MINT Mathematik, Informatik, Physik,
Elektro- und Informationstechnik
der Brandenburgischen Technischen Universität Cottbus-Senftenberg
genehmigte Dissertation
zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften (Dr. rer. nat.)

vorgelegt von

Sven Löffler
geboren am 06.03.1990 in Cottbus

Vorsitzender: Prof. Dr. Douglas W. Cunningham

Gutachterin: Prof. Dr. rer. nat. habil. Petra Hofstedt

Gutachter: Prof. Dr. habil. Herbert Kuchen

Tag der mündlichen Prüfung: 21.09.2022

https://doi.org/10.26127/BTUOpen-6149

Zusammenfassung

Mit Hilfe der Constraint-Programmierung können komplexe, häufig NP-vollständige Probleme, wie zum Beispiel Graphfärbungs-, Optimierungs-, Konfigurations- sowie Schichtplanungs-, Raum- und Zeitplanungsprobleme modelliert und gelöst werden. Im Vordergrund der Constraint-Programmierung steht dabei die deklarative Modellierung, also eine Beschreibung des eigentlichen Problems und nicht dessen Lösungsvorgangs. Während es Aufgabe des Entwicklers ist, das Problem zu modellieren, wird die Lösung einem separat implementierten Solver (Löser) überlassen. Im Idealfall löst dieser Solver das Problem, unabhängig von der konkreten Modellierung, immer schnellstmöglich.

In der Praxis kann von diesem Idealfall in der Regel nicht ausgegangen werden und somit hat die Art und Weise der Modellierung ein und desselben Problems und dessen Remodellierung teilweise einen erheblichen Einfluss auf die Lösungsgeschwindigkeit. Bisher bestehende Remodellierungsverfahren transformieren entweder Constraint-Probleme im Ganzen (zum Beispiel beim Umwandeln in SAT- oder binäre Constraint-Probleme) oder erfordern eine präzise Angabe und Steuerung seitens des Nutzers der Constraint-Programmierung (zum Beispiel bei der Tabularisierung). Während die erste Variante in der Regel nur bei sehr speziellen Problemen zu einer Beschleunigung des Lösungsvorganges führt, benötigt die zweite Variante vom Constraint-Modellierer zusätzliches Expertenwissen über das Lösungsverfahren des verwendeten Solvers. Die bisherigen Verfahren erlauben somit keine (voll)-automatisierte Transformation von Constraint-Problemen, bei der Transformationen nur durchgeführt werden, wenn diese auch zu einer Beschleunigung führen.

Für das dieser Arbeit übergeordnete Ziel der automatisierten Optimierung von Constraint-Problemen mittels Remodellierung, ohne dass dafür zusätzliches Expertenwissen notwendig ist, werden ausreichend viele verschiedene, gute und gut untersuchte Remodellierungen benötigt. In dieser Arbeit werden daher sowohl bestehende Verfahren für die Transformation von Constraint-Problemen weiterentwickelt, als auch völlig neue entworfen und deren Korrektheit nachgewiesen. Die in dieser Arbeit entwickelten Transformationen werden bezüglich ihrer beschleunigenden Wirkung auf Constraint-Probleme untersucht. Anhand von generierten und von realen Praxisbeispielen wird die Wirksamkeit der Transformationen evaluiert und es werden Schlussfolgerungen darüber gezogen, wann welche Substituierungen besonders vielversprechend sind.

Abstract

Constraint programming allows modeling and solving complex, often NP-complete problems, such as graph coloring, optimization, configuration, shift and room planning, and scheduling problems. The focus of constraint programming is declarative modeling, i.e. describing the problem rather than how to solve it. While it is up to the user to model the problem, its solution is left to a separately implemented solver. In a perfect declarative world, this solver always solves the problem as quickly as possible, regardless of its concrete modeling.

Unfortunately, this is often not the case and the different ways one and the same problem can be modeled and remodeled can have a significant influence on its solution speed. Already existing remodeling procedures transform either the whole constraint problem (i.e. transformations into SAT- or binary constraint problems) or are applied only when the modeler explicitly controls the optimization process (i.e. by tabulation). While the first method improves the solution speed only for very specific problems, the second method requires a great deal of expert knowledge on the part of the modeler regarding the solution processes of a constraint problem. Therefore, the existing methods do not provide a fully automated transformation of constraint problems, in which transformations are only carried out if they also lead to an acceleration.

For the higher goal of automatic optimization of constraint problems with remodeling methods without the necessity of extra expert knowledge, many different, well performing and well researched transformation methods are necessary. In this dissertation, existing methods for such model optimizations will be developed further and completely new ones will be designed, and their correctness proven. The transformations developed in the process of this dissertation will be investigated regarding their accelerating effect on constraint problems. Using both generated and real practical examples, the effectiveness of the transformations will be evaluated, and conclusions will be drawn at which time which substitutions are particularly promising.

Danksagung

An dieser Stelle möchte ich mich bei all denen bedanken, die es mir ermöglicht haben, meine Fähigkeiten mit dieser Arbeit unter Beweis zu stellen. Auf wissenschaftlicher Seite ist hier im besonderen Maße meine Vorgesetzte und Betreuerin Petra Hofstedt zu erwähnen. Dein Vertrauen in mich und meine Fähigkeiten, deine Kritiken und Ratschläge gaben mir erst die Möglichkeit, diese Dissertation zu bearbeiten und niederzuschreiben. Mein Dank gilt zwar besonders Petra Hofstedt, aber auch dem gesamten Fachgebiet Programmiersprachen und Compilerbau, das mir in Person von Ilja Becker, Ke Liu, Franz Kroll und Denny Schneeweiß wissenschaftlich beratend zur Seite stand. Aber auch Katrin Ebert, Gudrun Pehle und Daniela Schramm müssen hier zusätzlich zu den bereits Genannten erwähnt werden, die alle zusammen eine sehr familiäre und äußerst sympathische Atmosphäre am Lehrstuhl geschaffen haben und mich, wenn möglich, immer unterstützt haben.

Noch größerer Dank als meinen Kollegen gilt aber meiner Familie, die während meiner ganzen schulischen und akademischen Laufbahn immer hinter mir stand und mich stets in meinen Vorhaben unterstützt hat. Meinen Eltern danke ich für den Rückhalt, den sie mir immer gegeben haben, und gleichzeitig auch für den Freiraum, mit dem sie mich meinen eigenen Weg haben gehen lassen. Meiner Schwester danke ich an dieser Stelle noch einmal besonders für die reichliche Unterstützung, die sie mir gerade gegen Fertigstellung der Arbeit hat zukommen lassen. Selbstverständlich gilt der größte Dank auch meiner Frau, die sich das ein oder andere mal mit meinen Ausführungen zu Constraints und Automaten beschäftigen musste und geduldig zugehört und mitgedacht hat.

Inhaltsverzeichnis

1 Einleitung

In diesem Kapitel werden zunächst Idee und Vorgehen der Constraint-Programmierung vorgestellt und am Beispiel kurz erläutert. In Abschnitt 1.2 folgt die Zielsetzung dieser Arbeit, bevor in Abschnitt 1.3 eine Übersicht über die einzelnen Kapitel gegeben wird. Zum Abschluss dieser Einleitung folgen in Abschnitt 1.4 Hinweise zum besseren Verständnis der Arbeit.

1.1 Motivation

Spätestens mit der Einführung und Durchsetzung des Smartphones hat das Informationszeitalter, in dem wir uns aktuell befinden, einen Punkt erreicht, an dem nahezu jeder Mensch unabhängig von seiner Herkunft oder seinem Stand ständig von Hardware und Software umgeben ist. Der Kontakt und Umgang mit Computerprogrammen (zum Beispiel Apps) ist für viele längst alltäglich und normal. Computerprogramme helfen uns dabei, reale Probleme besser zu verstehen und zu lösen. Stellt die Lösung von Problemen, für die eine passende Software vorliegt, oftmals kein Problem dar, so sieht dies für Problemstellungen, für die noch keine darauf zugeschnittene Software vorliegt, oftmals ganz anders aus.

Wird an dieser Stelle einmal außen vor gelassen, dass nicht jeder die mathematischen Voraussetzungen oder das nötige Verständnis für Programmiersprachen mitbringt, so muss für gewöhnlich für das Schreiben eines problemlösenden Programms Wissen darüber vorhanden sein, wie dieses Problem zu lösen ist. Dieses Verhalten entspricht dem imperativen Programmierparadigma, welches auf einer Folge von Anweisungen basiert, die den Lösungsweg, also das „Wie“ etwas gelöst werden soll, in den Vordergrund stellt. Folglich muss zunächst bekannt sein, wie ein Problem gelöst werden kann, um in der Folge ein Programm zu schreiben, welches dieses löst. Das deklarative Programmierparadigma probiert dieses Paradoxon, dass zum Lösen eines Problems erst Wissen darüber vorhanden sein muss, wie das Problem zu lösen ist, zu vermeiden. Statt dem „Wie“ etwas gelöst werden soll, wird dabei die Beschreibung des Problems, also das „Was“ gelöst werden soll, in den Vordergrund gestellt. Dies ermöglicht das Entwickeln von Programmen für Probleme, die zwar gut beschrieben werden können, für die aber nicht klar ist, wie diese effektiv oder effizient gelöst werden können. Für Menschen, die vor einem Problem stehen, das sie nicht lösen können, erscheint das deklarative Vorgehen

Schicht | Tag	Mo	Di	Mi	Do	Fr	Sa	So
Frei	3	3	3	3	2	2	2
Früh	2	2	2	2	3	2	2
Spät	1	1	2	1	2	1	1
Nacht	2	2	1	2	1	3	3

Tabelle 1.1: Personalanforderungen für jeden Wochentag und jede Schicht.

möglicherweise einfacher und intuitiver zu sein als das imperative. Zu den bekanntesten deklarativen Sprachen gehören die funktionalen (z.B. ML, Haskell oder Erlang), die logischen (z.B. Prolog), die funktional-logischen (Curry), die Abfragesprachen (z.B. SQL) oder Constraint-basierte Sprachen (häufig als Bibliotheken für existierende Sprachen, z.B. GECODE oder CHOCO).

Die Constraint-Programmierung ist ein mächtiges Werkzeug zur deklarativen Beschreibung und Lösung von komplexen, kombinatorischen Problemen (NP-vollständige Probleme). Sowohl die deklarative Modellierung, dass also nur das Problem, nicht aber der Lösungsweg beschrieben werden muss, als auch die Tatsache, dass Probleme gelöst werden können, für die kein effizienter Lösungsalgorithmus bekannt ist, tragen dazu bei, dass die Constraint-Programmierung als Teilgebiet der Künstlichen Intelligenz (KI) angesehen wird. Die Constraint-Programmierung setzt sich selbst aus einer Vielzahl von Techniken und Verfahren aus den Bereichen künstlicher Intelligenz, Informatik, Datenbanken, Programmiersprachen, Operations Research und vielen weiteren zusammen. Typische Problemstellungen der Constraint-Programmierung sind Graphfärbungs-, Optimierungs-, Konfigurations- sowie Schichtplanungs-, Raum- und Zeitplanungsprobleme [109, 136]. Im Zentrum der Constraint-Programmierung stehen die Constraints (Einschränkungen), welche prädikatenlogische Formeln zur deklarativen Beschreibung mathematischer Probleme darstellen. Beispiele für Constraints sind $c_1 = (x < y)$, $c_2 = (x \in \{1, 2, 3, 4\})$ oder $c_3 = (x \land y \lor z)$.

Zur Veranschaulichung der Problemarten, welche mittels Constraint-Programmierung gelöst werden können, soll das folgende Schichtplanungsproblem dienen [73].

Beispiel 1.1: Ein Schichtplanungsproblem. *Betrachtet wird ein Schichtplanungsproblem mit den folgenden Anforderungen:*

1. *Der Planungszeitraum umfasst 8 Wochen mit jeweils 7 Tagen für 8 Mitarbeiter.*
2. *Es gibt 3 verschiedene Schichttypen und den zusätzlichen Typ „Freischicht“ mit folgender Zuordnung: Frei = 0, Früh = 1, Spät = 2, Nacht = 3.*
3. *Es gibt für jeden Wochentag und für jede Schicht eine exakte Vorgabe, wie viele Mitarbeiter benötigt werden, siehe dazu Tabelle 1.1.*

	Mo.	Di.	Mi.	Do.	Fr.	Sa.	So.
1	Früh	Früh	Spät	Spät	Spät	Nacht	Nacht
2	Frei	Frei	Frei	Früh	Früh	Früh	Früh
3	Frei	Frei	Frei	Frei	Früh	Früh	Früh
4	Spät	Spät	Spät	Frei	Frei	Frei	Frei
5	Früh	Früh	Früh	Nacht	Nacht	Nacht	Nacht
6	Frei	Frei	Früh	Früh	Früh	Nacht	Nacht
7	Nacht	Nacht	Frei	Frei	Spät	Spät	Spät
8	Nacht	Nacht	Nacht	Nacht	Frei	Frei	Frei

Tabelle 1.2: Eine mögliche Lösung des Schichtplanungsproblems.

4. *Jeder Mitarbeiter soll am Sonntag die gleiche Schicht wie am vorherigen Samstag haben.*
5. *Jeder Mitarbeiter soll immer an mindestens zwei aufeinander folgenden Tagen in der gleichen Schicht arbeiten.*
6. *Jeder Mitarbeiter soll immer an nicht mehr als vier aufeinander folgenden Tagen in der gleichen Schicht arbeiten.*
7. *Die Schichtfolge soll sich an das Prinzip der Vorwärtsrotation halten. Das bedeutet, dass auf eine frühere Schicht nur eine gleiche, eine spätere oder eine Freischicht folgen darf. Auf eine Frühschicht darf also z.B. eine Spätschicht folgen, auf eine Spätschicht aber keine Frühschicht.*
8. *Innerhalb von zwei Wochen sollen immer mindestens zwei freie Tage liegen.*

Das beschriebene Problem erfüllt einige (aber nicht alle) gesetzlichen Vorgaben [6] und arbeitswissenschaftlichen Empfehlungen [7]. Es kann für verschiedene konkrete Schichtplanungsprobleme angepasst oder erweitert werden.

Eine mögliche Lösung für das Problem in einer speziellen Form ist in Tabelle 1.2 dargestellt. Die Lösung kann sowohl den Dienstplan für einen Mitarbeiter für acht Wochen repräsentieren, dabei entspricht die x-te Zeile der x-ten Woche für den ersten Mitarbeiter, als auch den Dienstplan für acht Mitarbeiter für eine Woche darstellen, dabei entspricht die x-te Zeile der Schicht des x-ten Mitarbeiters für die erste Woche.

Es wurde dabei ein Rotationsprinzip verwendet, sodass dem ersten Mitarbeiter in der zweiten Woche jeweils der Plan des zweiten Mitarbeiters in der ersten Woche bzw. dem ersten Mitarbeiter in der dritten Woche der Plan des zweiten Mitarbeiters in der zweiten Woche und des dritten Mitarbeiters in der ersten Woche zugewiesen wird. Dem Mitarbeiter $m \in \{1, ..., 8\}$ *ist dementsprechend in der Woche* $n \in \{1, ..., 8\}$ *der Schichtplan zugeordnet, der in Zeile* $k = ((n + m - 2)$ *modulo* $8) + 1$ *dargestellt ist.*

Die potentielle Größe des Schichtplanungsproblems bzw. seinen Suchraumes (ohne das spezielle Rotationsprinzip zu verwenden) umfasst für jeden der acht Mitarbeiter und für jeden Tag (acht Wochen mit jeweils sieben Tagen) vier verschiedene mögliche Schichten, also $4^{8*8*7} \approx 5,283 * 10^{269}$ Möglichkeiten. Durch Verwendung des erwähnten Rotationsprinzips kann sowohl der Suchraum als auch der Lösungsraum signifikant verringert werden, so dass in der Regel deutlich schneller Lösungen gefunden werden können. Die Suchraumgröße beläuft sich dann „nur noch“ auf $4^{8*7} \approx 5,192 * 10^{33}$ Möglichkeiten, was in Anbetracht der geringen Größe und Komplexität des vorgestellten Planungsproblems immer noch gewaltig ist.

Die Lösungssuche bei Problemen dieser Größe kann schnell ein akzeptables Maß an Zeit überschreiten. Die dargestellten Zahlen sollen veranschaulichen, wie groß Constraint-Probleme in der Regel sind, und verdeutlichen, wie wichtig optimierte Problemmodellierungen sind. Ziel der deklarativen Programmierung ist es unter anderem, dass die Beschreibung des Problems und nicht die des Lösungsvorganges im Vordergrund steht. Wünschenswert ist es dabei, unabhängig von der Art und Weise, wie ein Problem beschrieben ist, schnellstmöglich eine, mehrere, alle oder eine bestmögliche Lösung für das modellierte Problem zu finden. In der Realität ist dies allerdings meist nicht gegeben. Dort hat die Art und Weise, wie ein Problem modelliert wird, oftmals einen sehr großen Einfluss auf die Lösungsgeschwindigkeit. Dieses Verhalten tritt natürlich auch in der Constraint-Programmierung auf. Aus diesem Grund werden in dieser Arbeit Verfahren zum Optimieren und Regularisieren von Constraint-Problemen erarbeitet und vorgestellt. Dabei wird unter einem Verfahren zum Optimieren von Constraint-Problemen eine semantikerhaltende Umformung des Constraint-Modells verstanden, die die Lösungszeit des Constraint-Problems im Durchschnitt verringert. Das Regularisieren stellt dabei eine besondere Form dieser Umformungen dar, bei der Teile des Constraint-Modells durch *Regular*-Constraints ersetzt werden. Ziel ist es, beliebige Constraint-Modellierungen (automatisch) in äquivalente Modellierungen umzuformen, die anschließend schneller gelöst werden können.

Das Vorgehen zum Lösen eines Constraint-Problems lässt sich grundlegend in zwei Phasen einteilen:

1. Die deklarative Modellierung eines Constraint-Problems.
2. Das Lösen eines Constraint-Problems mittels eines Solvers (Lösers).

Die Modellierung des Problems obliegt dabei dem (in der Regel menschlichen) Programmierer, wohingegen das Lösen des Problems einem Algorithmus (Solver), der für den Entwickler häufig eine Art Blackbox darstellt, überlassen wird. Durch diese Zweiteilung ist es möglich, die Beschreibung des Problems, also das „Was“gelöst werden soll, in den Vordergrund und das „Wie“ gelöst werden soll, hintenan zu stellen.

Insbesondere erlaubt die Constraint-Programmierung den Umgang mit unvollständigem Wissen und ermöglicht dabei eine für den Menschen sehr natürliche Wissensrepräsentation [46]. Diese natürliche Wissensrepräsentation ist allerdings nur dann gewinnbringend, wenn die Probleme, die mit ihr dargestellt werden, auch in akzeptabler Zeit gelöst werden können. Leider existiert oftmals ein sehr großer Unterschied zwischen für den Menschen verständlich dargestelltem Wissen und für den Computer schnell zu verwertenden Informationen. Aus diesem Grund ist es wichtig, leicht verständliche Problembeschreibungen automatisch und schnell in solche zu überführen, die ein Computer schnellstmöglich lösen kann. Denn nur so kann in akzeptabler Zeit eine Lösung gefunden und der Nutzen, der durch die als natürlich empfundene Wissensrepräsentation entsteht, ausgeschöpft werden.

1.2 Zielsetzung

Ziel dieser Arbeit ist die Untersuchung von Constraint Satisfaction-Problemen hinsichtlich Optimierung und Regularisierung ihrer zugrunde liegenden Modelle. Die herkömmliche Herangehensweise in der Constraint-Programmierung gliedert sich in zwei Phasen: erstens, das deklarative Modellieren der Problemstellung und zweitens, das Lösen des Modells mittels eines Constraint Solvers (Lösers).

Für die Modellierung eines realen Problems stehen dem Anwender der Constraint-Programmierung viele verschiedene Constraints zur Verfügung. Diese Constraints sind teilweise sehr stark auf einzelne Problemstellungen zugeschnitten und erlauben oftmals eine intuitive Modellierung der Problemstellung. Aufgrund der verschiedenen Constraints, die der Nutzer der Constraint-Programmierung zur Auswahl hat, kann ein reales Problem oftmals auf sehr viele, verschiedene Arten modelliert werden. Das fertige Modell wird anschließend dem Solver übergeben, der das Problem löst.

Für das Lösen eines Constraint-Problems wird nach dessen Modellierung ein Solver verwendet. Dieser stellt im Allgemeinen eine Art Blackbox dar, die für den Anwender der Constraint-Programmierung nicht einsehbar ist. Allerdings kann dieser häufig durch Eingaben des Nutzers konfiguriert und teilweise gesteuert werden. Hinzu kommt, dass es verschiedene Solver für verschiedene Constraint-Probleme gibt. Es obliegt dem Nutzer der Constraint-Programmierung einen geeigneten Solver auszuwählen.

Idealerweise würde jede Modellierung einen besten Lösungsprozess anstoßen und so in kürzester Zeit, je nach Nutzerwunsch, eine, mehrere, alle oder eine beste Lösung für die Problemstellung finden. In der Praxis wird diese Wunschvorstellung allerdings nicht erfüllt. Daher bietet die Remodellierung von Constraint-Problemen einen interessanten Ansatzpunkt, um die Lösungsgeschwindigkeit von Constraint-Problemen zu erhöhen. In der Vergangenheit [24, 26, 111] und auch parallel zu dieser Arbeit [10] wird weiterhin intensiv in diese Richtung geforscht und publiziert. Bisher wurden vor allem Ansätze zur Umwandlung von Constraints in boolesche, *Regular*- oder *Table*-Constraints entwickelt. Allerdings erlauben es die bisherigen Ansätze nicht, jedes Constraint-Problem mittels Remodellierung zu beschleunigen, was einerseits daran liegt, dass nicht jede Transformation jedes Problem beschleunigt und andererseits daran, dass die benötigten Transformationen noch nicht existieren. Aus diesem Grund werden in dieser Arbeit die Ansätze zur Remodellierung von Constraints durch das *Regular-Language-Membership*-Constraint erweitert und neue Ansätze zur Umwandlung von Constraints in boolesche *Scalar*-Constraints entwickelt.

Im Gegensatz zu dem in der Vergangenheit und Gegenwart viel untersuchten *Table-Constraint* [50, 87, 90, 103, 104] erlaubt das *Regular-Language-Membership*-Constraint oft eine wesentlich kompaktere Repräsentation von Teillösungen, wodurch sich die Chance bietet, größere Teile eines Constraint-Problems mit ihm zu substituieren, als das mit dem *Table-Constraint* möglich ist. Das Substituieren von einzelnen Constraints durch andere kann zu einer Erhöhung des Propagationslevels, einer Verringerung der Redundanz und einer Verringerung der benötigten Backtracks innerhalb des Lösungsvorganges eines Constraint-Problems führen und somit dessen Lösungsgeschwindigkeit erhöhen.

Für die zweite Phase, dem Lösen des modellierten Problems mittels eines Constraint-Solvers, ergeben sich aus der Remodellierung verschiedene interessante Schlussfolgerungen. Durch das Substituieren aller Constraints durch Constraints einer bestimmten Klasse besteht die Chance, speziellere Constraint-Solver zu verwenden, die speziell auf die verwendete Klasse von Constraints spezialisiert sind. Das Verwenden von spezialisierten Constraint-Solvern hat sich in der Vergangenheit als sehr erfolgreich herausgestellt. Als Beispiele dafür seien SAT- und Simplex-Solver erwähnt.

Selbst, wenn eine vollständige Substitution der Constraints eines Modells nicht möglich sein sollte, können durch das möglicherweise besonders häufige Auftreten von Constraints einer bestimmten Klasse Konfigurationen für den Constraint-Solver abgeleitet werden, die dessen Lösungsgeschwindigkeit erhöhen.

Als Ergebnis dieser Arbeit entsteht somit ein Drei-Phasen-Modell, wie es in Abbildung 1.1 zu sehen ist. Zwischen den beiden bestehenden Phasen Modellierung und Lösung wird dabei eine Remodellierungs- und Konfigurationsphase eingefügt, in der mögliche Substitutionen ermittelt und besonders vorteilhaft erscheinende Transformationen durchgeführt werden.

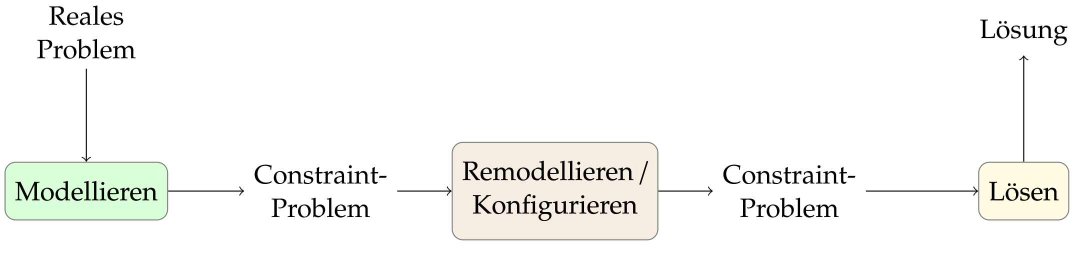


Abbildung 1.1: Das neue Drei-Phasen-Modell der Constraint-Programmierung.

Forschungsfragen

In dieser Arbeit wird die folgende Hauptforschungsfrage betrachtet:

HF Können bestehende Remodellierungsverfahren erweitert und neue entwickelt werden, um eine (automatische) Remodellierung von Constraints oder Constraint-Systemen, die zu einer Beschleunigung des Lösungsvorgangs führen, zu gewährleisten?

Zur Bearbeitung und Beantwortung dieser Frage ergeben sich eine Fülle an Teilforschungsfragen:

TF1 Welche allgemeinen Transformationen für beliebige Constraints in Constraints einer anderen Art existieren bereits?

TF2 Kann ein allgemeines Konzept für Constraints, die alle anderen FD-Constraints substituieren können, entwickelt werden?

TF3 Welche Vorteile ergeben sich für CSPs durch das Substituieren einzelner oder von Mengen von Constraints?

TF4 Können weitere allgemeine Transformationsverfahren von beliebigen Constraints in Constraints einer bestimmten Art entwickelt werden?

TF5 Können speziellere Substituierungsverfahren entwickelt werden, die eine schnellere Transformation und das Verwenden kompakterer Datentypen erlauben?

TF6 Führen die erforschten Substituierungsverfahren bei realen Anwendungsbeispielen zu Beschleunigungen?

TF7 Kann eine Heuristik entwickelt werden, die prognostiziert, wann die Vereinigung mehrerer Constraints zu einem neuen Constraint eine beschleunigende Wirkung auf das zu lösende Constraint-System hat?

TF8 Kann ein Klassifikator entwickelt werden, der zuordnet, welche Substituierungen sich für welche Constraints am besten eignen?

Im folgenden Abschnitt wird eine Übersicht über die einzelnen Kapitel der Arbeit gegeben und jeweils erläutert, wie diese zur Beantwortung der Forschungsfragen beitragen.

1.3 Kapitelübersicht

Die nachfolgende Arbeit gliedert sich in sechs weitere Kapitel.

Zur Bearbeitung der Forschungsfragen wird zunächst in Kapitel 2 grundlegendes Wissen über die Constraint-Programmierung zusammengetragen. Dabei wird insbesondere auf die Begriffe Constraint, Constraint Satisfaction-Problem (CSP) und Finite Domain-Constraint eingegangen. Im Anschluss werden verschiedene Constraint-Domänen und Solver vorgestellt. Insbesondere wird auf die Finite Domain-vollständigen Solver detailliert eingegangen. Abschließend wird in diesem Kapitel das Konzept der globalen Constraints erläutert und ausgewählte Vertreter dieser werden eingeführt.

Zur Klärung der Frage TF1, ob und welche allgemeinen Transformationen von Constraints bisher existieren, werden in Kapitel 3 verwandte Arbeiten betrachtet. Insbesondere wird dabei auf binäre Transformationen und Umwandlungen in SAT-Probleme eingegangen, welche Ähnlichkeiten mit den in dieser Arbeit neu entwickelten Verfahren der booleschen Skalarisierung aufweisen. Im Anschluss daran wird auf die Tabularisierung eingegangen, die vergleichbar mit der in dieser Arbeit vorgestellten Regularisierung ist. Zum Ende des Kapitels wird der aktuelle Stand der Forschung noch einmal zusammenfassend veranschaulicht und es werden die wesentlichen Bestandteile dieser Arbeit von der bisherigen Forschung abgegrenzt.

In Kapitel 4 wird sowohl ein allgemeines Konzept für Constraints, die alle anderen FD-Constraints substituieren können (FD-vollständige Constraints), als auch eines für CSPs, die alle anderen FD-CSPs substituieren können (FD-vollständige CSPs), entwickelt (TF2). Zusätzlich werden sowohl für FD-vollständige Constraints als auch für FD-vollständige CSPs Beispiele gegeben und es wird erläutert, welche Vorteile sich durch das Substituieren einzelner oder von Mengen von Constraints ergeben (TF3).

In Kapitel 5 werden neuartige Substituierungen entwickelt und vorgestellt (boolesche Skalarisierungen) und bestehende weiterentwickelt (Regularisierung). Es werden dabei zunächst mit der allgemeinen Regularisierung, der konfliktbasierten und der Support-basierten booleschen Skalarisierung drei neue allgemeine Transformationsmöglichkeiten geschaffen (TF4), bevor im Anschluss daran spezielle Substituierungen für ausgewählte globale Constraints erarbeitet werden (TF5). Zusätzlich zu den allgemeinen Verfahren und der Transformation spezieller globaler Constraints wird auf die Substituierung logischer Meta-Constraints eingegangen und eine Möglichkeit geschaffen, jedes logische Meta-Constraint zu regularisieren oder in boolesche *Scalar*-Constraints umzuwandeln (TF5). Abschließend werden in diesem Kapitel Schlussfolgerungen bezüglich der verschiedenen Substituierungen gezogen und weitere mögliche Substituierungen abgeleitet.

In Kapitel 6 werden die neu geschaffenen Substituierungsmöglichkeiten evaluiert. Dafür werden die Substituierungen einzeln auf CSPs mit nur einem globalen Constraint betrachtet und bewertet, inwieweit diese zu einer Beschleunigung führen. Anhand dieser Einzelbetrachtungen werden Rückschlüsse darauf gezogen, wann Substituierungen globaler Constraints voraussichtlich zu einer Beschleunigung führen. Im zweiten Teil dieses Kapitels werden reale Anwendungsbeispiele und der Einfluss der verschiedenen Substituierungen auf sie betrachtet (TF6). Schlussendlich werden Hypothesen darüber aufgestellt, wann die verschiedenen Substituierungen und wann die Vereinigung verschiedener Constraints zu einer beschleunigten Lösung von Constraint-Systemen führen (TF7 und TF8).

Das abschließende Kapitel 7 beginnt mit einer Zusammenfassung der Arbeit, zieht ein Fazit, in dem auf die Hauptforschungsfrage (HF) eingegangen wird und gibt einen Ausblick über zukünftige Arbeiten.

1.4 Anmerkungen zur Arbeit

In diesem Abschnitt werden Anmerkungen bezüglich der Arbeit und der Art und Weise, wie diese gelesen werden soll, getroffen. Darüber hinaus werden die technischen Voraussetzungen (Soft- und Hardware) für das Erstellen und Durchführen der späteren Testreihen zur Evaluation aufgeführt.

Gendergerechte Sprache

In dieser Arbeit wird auf die einzelne Benennung aller Geschlechter verzichtet und stattdessen immer nur die männliche Form verwendet. Dies geschieht ohne diskriminierende Absicht und lediglich zum Zwecke der besseren Lesbarkeit der Arbeit. Als Beispiel stehen Mitarbeiter immer sowohl für Mitarbeiter als auch für Mitarbeiterinnen.

Synonyme Verwendung

In dieser Arbeit agieren die Begriffe *Regular-Language-Membership*-Constraint, *Regular-Membership*-Constraint und *Regular*-Constraint als Synonyme füreinander.

Vereinfachte Bezeichnungen

Ab Kapitel 3 wird unter einem Automaten immer ein levelbasierter deterministischer endlicher Automat (levelbasierter DFA) verstanden (Vergleich Definition 2.22).

Englische Begriffe

In dieser Arbeit werden wissenschaftliche Begriffe wie z.B. Constraint, Solver etc. in der Regel im Englischen belassen und nicht deren deutsche Übersetzung verwendet. Das ist dem geschuldet, dass sich die englischen Begriffe in der internationalen wie nationalen Literatur etabliert haben.

Verwendete Soft- und Hardware

Für die gesamte Arbeit und alle damit in Zusammenhang durchgeführten Versuchsreihen wurde ein DELL Laptop mit einem Intel i7-4610M Prozessor (quad core) mit einer Taktfrequenz von 3.00GHz und einem 16 GB DDR3-Arbeitsspeicher mit einer Taktfrequenz von 1600 MHz eingesetzt. Als Betriebssystem wurde Microsoft Windows 10 professional verwendet.

Es wurde ein Java mit JDK-Version 1.8.0_191 und der Choco-Solver mit Version 4.0.4 [129] verwendet. Als Standardsuchstrategie wurde die *DomOverWDeg*-Suchstrategie verwendet, welche in [35] erklärt ist und als Standardsuchstrategie im Choco-Solver verwendet wird [129].

2 Grundlagen der Constraint-Programmierung

In diesem Kapitel werden zunächst die Grundlagen der Constraint-Programmierung erläutert (Abschnitt 2.1), bevor die Probleme charakterisiert und klassifiziert werden (Abschnitt 2.2), die mit Hilfe der Constraint-Programmierung gelöst werden können. Anschließend werden zunächst in Abschnitt 2.3 verschiedene Constraint-Solver und Lösungsverfahren vorgestellt, bevor in Abschnitt 2.4 auf die für diese Arbeit essentiellen Finite Domain-Solver detailliert eingegangen wird. Abschließend wird das Konzept der globalen Constraints eingeführt, es werden verschiedene globale Constraints exemplarisch vorgestellt und erläutert (Abschnitt 2.5).

2.1 Constraint Satisfaction-Probleme

Die Beschreibung eines Problems in der Constraint-Programmierung erfolgt deklarativ über Variablen $X = \{x_1, x_2, ..., x_n\}$, Domänen $D = \{D_1, D_2, ..., D_n\}$ und Constraints $C = \{c_1, c_2, ..., c_m\}$. Zu jeder Variablen $x_i \in X$ existiert eine entsprechende Domäne $D_i \in D$, die den Wertebereich der Variable beschränkt. Ein Tripel aus Variablen, zugehörigen Domänen und Constraints wird als Constraint Satisfaction-Problem (CSP) $P = (X, D, C)$ bezeichnet.[1]

Definition 2.1: *Constraint Satisfaction-Problem (CSP).* Ein Constraint Satisfaction-Problem P ist ein Tripel (X, D, C) mit:

$$\begin{array}{rcll} X & = & \{x_1, ..., x_n\} & \text{ist eine geordnete Menge von } n \text{ Variablen,} \\ D & = & \{D_1, ..., D_n\} & \text{ist eine zu der Variablenmenge } X \text{ zugehörige, geordnete} \\ & & & \text{Menge von } n \text{ Domänen,} \\ C & = & \{c_1, ..., c_m\} & \text{ist eine Menge von } m \text{ Constraints (nach [14]).} \end{array}$$

2.1.1 Constraints und ihre Eigenschaften

Ein Constraint $c \in C$ ist ein Tupel (X', R) aus einer Relation R und einer geordneten Teilmenge der Variablen $X' \subseteq X$ eines CSPs $P = (X, D, C)$, über denen die Relation R

---

[1] Je nach Literatur wird anstelle von D_i für die Domäne der Variable x_i auch die Notation $D(x_i)$ verwendet. In dieser Arbeit wird bei einfach indexierten Variablen mit gleichem Namen die erste Notation verwendet. In den anderen Fällen wird auf die zweite Notation zurückgegriffen.

definiert ist. Präziser beschreibt die Relation R die gültigen Wertebelegungen, die die Variablen in X' annehmen dürfen. Die Relation R drückt somit eine Teilmenge des kartesischen Produkts der Domänenwerte $D_1 \times ... \times D_r$ der entsprechenden Variablen $X' = \{x_1, ..., x_r\} \subseteq X$ des Constraints c aus. Die beschriebene Variablenmenge X' wird auch als *scope* (auf deutsch Anwendungsbereich) des Constraints bezeichnet.

Definition 2.2: ***Constraint*.** Ein Constraint $c = (X', R) \in C$ ist durch eine Relation R über eine geordnete Teilmenge der Variablen $X' \subseteq X$ definiert [45].

Definition 2.3: ***Scope*.** Der *scope* eines Constraints $c = (X', R)$ entspricht den Variablen des Constraints: $scope(c) = X'$ [45].

Beispiele für Constraints sind $c_1 = (\{x, y\}, (x > y)), c_2 = (\{R, U, I\}, (R = U/I)), c_3 = (\{I, Q, t\}, (I = Q/t))$ mit entsprechenden reellwertigen Domänen oder $c_4 = (\{A, B, C\}, (A \wedge B \rightarrow C))$ über booleschen Werten. Im Fall von endlichen Domänen kann die Relation eines Constraints sowohl intensional, durch das implizite Darstellen der Tupel als Relation R, als auch extensional, über das explizite Auflisten von Tupeln T dargestellt werden. Dafür wird zusätzlich zur intensionalen Darstellung $c = (X, R)$ die Notation $c = (X, T)$ (extensional) eingeführt. Dabei ist T die explizite, endliche Auflistung aller zulässigen Tupel für die Variablen X, die sonst implizit durch die Relation R ausgedrückt werden. Sind den Variablen x und y beispielsweise die folgenden endlichen Domänen $D_x = \{1, 2, 3\}$ und $D_y = \{1, 2, 3\}$ zugeordnet, so kann das Constraint c_1 auch wie folgt dargestellt werden: $c_1 = (X = \{x, y\}, T = \{(2, 1), (3, 1), (3, 2)\})$.

Typische Eigenschaften von Constraints sind nach Barták [25]:

1. Constraints können Teilinformationen beschreiben
2. Constraints sind ungerichtet
3. Constraints sind deklarativ
4. Constraints sind additiv
5. Constraints können sich überlappen

Anhand der Constraints $c_2 = (\{R, U, I\}, (R = U/I))$ und $c_3 = (\{I, Q, t\}, (I = Q/t))$ werden diese Eigenschaften im Folgenden verdeutlicht. Es wird an dieser Stelle davon ausgegangen, dass die Domänen der Variablen R, U, I, Q und t der Menge der reellen Zahlen $\mathbb{R}$ entsprechen ($D(R) = D(U) = D(I) = D(Q) = D(t) = \mathbb{R}$).

Die Möglichkeit eines Constraints *Teilinformationen zu beschreiben*, erlaubt es den Variablen eines solchen Constraints, nicht nur konkrete Werte zuzuordnen, sondern auch Mengen oder Intervalle. Constraint c_2 kann zum Beispiel die Informationen, dass I gleich eins und U kleiner 230 ist, dazu nutzen, um den Wertebereich von R auf kleiner 230 zu reduzieren. Es müssen also nicht alle anderen Variablen eines Constraints instantiiert sein, um einen Informationsgewinn für weitere Variablen zu erzielen.

Das ohmsche Gesetz, das in c_2 dargestellt ist, verdeutlicht, dass Constraints *ungerichtet* sind. Constraints gelten immer in alle Richtungen. Das bedeutet, dass aus $R = U/I$ immer auch hervorgeht, dass $U = R * I$ und $I = U/R$ gilt, ohne dass dies explizit angegeben werden muss.

Die *deklarative* Eigenschaft eines Constraints ist gleichbedeutend damit, dass nur angegeben werden muss, welche Beziehungen gelten, nicht aber, wie diese berechnet werden bzw. mit ihnen gerechnet wird. Dies kann am Beispiel der Constraints c_2 und c_3 erläutert werden, welche zusammen ein Gleichungssystem ergeben, für das nicht angegeben werden muss, wie dieses zu lösen ist. Der Constraint-Solver übernimmt den Lösungsvorgang, ohne dass der Anwender der Constraint-Programmierung dies näher spezifizieren muss.

Unter der *Additivität* ist zu verstehen, dass die Reihenfolge der Eingabe der Constraints keine Rolle spielt, sondern sie immer alle zur gleichen Zeit erfüllt sein müssen. Im Fall eines CSPs mit c_2 und c_3 hat das zur Folge, dass, egal ob c_2 zeitlich vor c_3 angelegt wird oder umgekehrt, dies keinen Einfluss auf die Lösungsmenge des CSPs hat. Es besteht aber die Möglichkeit, dass eine unterschiedliche Reihenfolge der Constraints einen Einfluss auf die Lösungsgeschwindigkeit hat. Mehr Informationen dazu lassen sich Kapitel 2.4.1 entnehmen.

Zwei Constraints *überlappen* sich, wenn sie wie im Fall von c_2 und c_3 mindestens eine gleiche Variable enthalten (in diesem Fall die Variable I). Als Konsequenz daraus können Abhängigkeiten entstehen. So kann z.B. eine Veränderung der Variable R dazu führen, dass der Wert t geändert werden muss, obwohl beide nicht direkt über ein Constraint verbunden sind. Durch Überlappungen von Constraints können somit größere Zusammenhänge und Abhängigkeiten geschaffen werden, als aus der Beschreibung jedes einzelnen Constraints für sich betrachtet hervorgeht.

Eine Änderung von R könnte durch Auswerten von c_2 zu einer Änderung von I führen, was wiederum durch Auswerten von c_3 zu einer Änderung von t führen kann.

2.1.2 Belegungen und Lösungen

Für die folgenden Definitionen wird von einem CSP $P = (X, D, C)$ mit Variablen $X = \{x_1, ..., x_n\}$, Domänen $\{D_1, ..., D_n\}$ und Constraints $C = \{c_1, ..., c_m\}$ ausgegangen. Werden den Variablen $x_i \in X$ Werte d_i aus ihren jeweiligen Domänen D_i zugeordnet, so wird von einer Variablenbelegung ϕ gesprochen. Erfüllt eine solche Belegung die Relation eines Constraints, so ist die Belegung eine gültige Belegung des Constraints.

Definition 2.4: ***Gültige Belegung eines Constraints.*** Eine Variablenbelegung $\phi : X \to D$ für die Variablen aus $X = \{x_1, ..., x_n\}$ mit $D = \bigcup_{i \in \{1,...,n\}} D_i$ und $\phi(x_i) \in D_i$ wird gültige Belegung eines Constraints $c = (X', R)$ mit $X' \subseteq X$ genannt, wenn die Relation R mit den Werten $\phi(x_i), \forall x_i \in X'$ erfüllt ist (nach [45]).

Aufbauend auf der Definition für eine gültige Belegung eines Constraints wird die Lösung eines Constraint-Problems definiert. Wenn jeder Variablen eines Constraint-Problems ein Wert ihrer Domäne zugeordnet ist, so dass alle Constraints erfüllt sind, so stellt diese Zuordnung eine Lösung des Constraint-Problems dar.

Definition 2.5: ***Lösung eines Constraint-Problems.*** Die Variablenbelegung $\phi : X \rightarrow D$ mit $\phi(x_1) = d_1, ..., \phi(x_n) = d_n, \forall d_i \in D_i, i \in \{1, ..., n\}$ für die Variablen $X = \{x_1, ..., x_n\}$ eines Constraint-Problems $P = (X, D, C)$ wird Lösung dieses Constraint-Problems genannt, wenn ϕ eine gültige Belegung für jedes Constraint $c \in C$ ist (nach [14, 45]).

Neben Constraint Satisfaction-Problemen, bei denen nach einer beliebigen Lösung, n-vielen oder allen Lösungen gesucht wird, besteht auch die Möglichkeit, nach einer Lösung zu suchen, die zusätzlich eine gegebene Zielfunktion optimiert. Ein solches Constraint-Problem wird Constraint Satisfaction Optimization-Problem (CSOP oder COP) genannt und ist eine Erweiterung eines CSPs um eine Optimierungsfunktion f.

Definition 2.6: ***Constraint Satisfaction Optimization-Problem (CSOP).*** Ein Constraint Satisfaction Optimization-Problem (CSOP) P_{opt} ist ein Quadrupel (X, D, C, f) mit:

$X = \{x_1, ..., x_n\}$	ist eine Menge von n Variablen,
$D = \{D_1, ..., D_n\}$	ist eine Menge von n Domänen,
$C = \{c_1, ..., c_m\}$	ist eine Menge von m Constraints,
$f \in D_1 \times ... \times D_n \rightarrow \mathbb{R}$	ist eine Zielfunktion, die jeder Lösung einen numerischen Wert zuordnet.

Ziel ist die Minimierung (bzw. Maximierung) des Ergebnisses der Zielfunktion f [70, 152].

Ohne Beschränkung der Allgemeinheit kann bei einem CSOP von einem Minimierungsproblem ausgegangen werden. Sollte ein Maximum gesucht werden, so kann die Zielfunktion mit -1 multipliziert werden und somit wieder nach einem Minimum gesucht werden.

2.1.3 Constraint-Netze

In der Literatur werden die Begriffe Constraint Satisfaction-Probleme und Constraint-Netze teilweise als Synonyme füreinander verwendet. In dieser Arbeit wird der Begriff Constraint Satisfaction-Problem für die mathematische Beschreibung des Problems und der Begriff Constraint-Netz ausschließlich für die Visualisierung eines CSPs verwendet.

Das Visualisieren von Constraint-Problemen stellt für die Nachvollziehbarkeit und Fehlerdiagnostik ein wichtiges Forschungsgebiet innerhalb der Constraint-Programmierung dar. Eine Repräsentationsmöglichkeit stellen dabei Graphen dar.

Definition 2.7: ***Graph.*** Ein Graph G ist ein Tupel (V, E), wobei V eine Menge von Knoten (vertices) und E eine Menge von Kanten (edges) bezeichnet. Dabei ist E eine Teilmenge aller 2-elementigen Teilmengen von V.

Constraint-Probleme $P = (X, D, C)$ mit nur unären und binären Constraints können als Graphen $G = (X^G, E^G)$ repräsentiert werden. Dabei existiert für jede Variable x in X ein Knoten x^G in X^G. Die Beschriftung (das Label) jedes Knotens x^G kann dabei den Namen der Variable x und deren Wertebereich $D(x)$ widerspiegeln. Für jedes Constraint c in C existiert eine Kante e^G in E^G.

Handelt es sich bei c um ein unäres Constraint mit $scope(c) = \{x\}$, so stellt die Kante e^G eine Schleife am Knoten x^G dar ($e^G = \{x^G, x^G\} \in E^G$). Ist das Constraint c binär mit $scope(c) = \{x_1, x_2\}$, so ist die zugehörige Kante e^G eine ungerichtete Kante zwischen x_1^G und x_2^G in G, d.h. $e^G = \{x_1^G, x_2^G\} \in E^G$. Die Beschriftung der Kanten E^G spiegelt die Bezeichnung oder die Relation des Constraints c wider.

Für die Visualisierung von Constraint-Problemen, die mindestens ein Constraint c enthalten, das mehr als zwei Variablen beinhaltet ($|scope(c)| > 2$), können Hypergraphen verwendet werden. Hypergraphen unterscheiden sich von den zuvor vorgestellten Graphen dahingehend, dass ihre Kanten beliebig viele Knoten umfassen können. Für jedes Constraint c in C mit $\{x_1, ..., x_n\} = scope(c)$ ergibt sich die zugehörige Hyperkante $e^G = \{x_1^G, ..., x_n^G)\} \in E^G$. Hyperkanten werden häufig als Sphären um die betroffenen Knoten visualisiert. Die Beschriftung der Hyperkanten E^G spiegelt die Bezeichnung oder die Relation des Constaints c wider.

Definition 2.8: ***Hypergraph.*** Ein Hypergraph G ist ein Tupel (V, E), wobei V eine Menge von Knoten und E eine Menge von Kanten bezeichnet. Dabei ist E eine Menge von Teilmengen der Knotenmenge $E = \{S_1, ..., S_m\}, S_j \subseteq V \mid \forall j \in \{1, ..., m\}$ [45].

Aufbauend auf Definition 2.8 ist ein Constraint-Netz ein Hypergraph, der ein gegebenes CSP P repräsentiert.

Definition 2.9: ***Constraint-Netz.*** Das Constraint-Netz N zu einem CSP $P = (X, D, C)$ ist ein Hypergraph $N = (X^G, E^G)$, bestehend aus $|X|$ vielen Knoten X^G und $|C|$ vielen Kanten E^G. Jede Kante $e_i^G \in E^G$ verbindet genau die Knoten $x_1^G, ..., x_j^G \in X^G$, die die Variablen $scope(c_i) = \{x_1, ..., x_j\}$ des von $e_i^G \in E^G$ repräsentierten Constraints $c_i \in C$ darstellen.

Hypergraphen sind für die mathematische Beschreibung von Constraints geeignet. Die Darstellung der Hyperkanten als Sphären, insbesondere bei überlappenden Hyperkanten, wird allerdings schnell unübersichtlich. Aus diesem Grund wird an dieser Stelle eine zweite Darstellungsform eines Constraint-Problems $P = (X, D, C)$ als bipartiter Graph $G = (V_1, V_2, E)$ eingeführt.

Definition 2.10: *Bipartiter Graph.* Ein Graph $G(V, E)$ mit Knoten V und Kanten E ist bipartit, wenn die Knotenmenge V in zwei disjunkte Mengen V_1 und V_2 aufgeteilt werden kann, so dass keine Kanten zwischen Knoten der selben Knotenmenge (V_1 oder V_2) existieren. Ein bipartiter Graph kann auch als Tripel $G = (V_1, V_2, E)$ angegeben werden [19].

Dabei repräsentiert jeder Knoten v_i der ersten Knotenmenge V_1 eine Variable $x_i \in X$ und jeder Knoten v_j der zweiten Knotenmenge V_2 ein Constraint $c_j \in C$ von P. Eine Kante $e \in E$ existiert genau dann zwischen einem Knoten $v_i \in V_1$ und $v_j \in V_2$, wenn die Variable x_i Teil des Constraints c_j, also $x_i \in scope(c_j)$, ist.

Im Beispiel 2.1 werden zwei CSPs vorgestellt und die verschiedenen Visualisierungsmöglichkeiten veranschaulicht.

Beispiel 2.1: Visualisierung von CSPs mit Constraint-Netzen. *Seien die beiden CSPs* $P^1 = (X^1, D^1, C^1)$ *und* $P^2 = (X^2, D^2, C^2)$ *wie folgt gegeben.*

CSP 1: $P^1 = (X^1, D^1, C^1)$ mit:
$X^1 = \{x_1, x_2, x_3\}$
$D^1 = \{D_1, D_2, D_3 \mid D_1 = D_2 = D_3 = \{0, 1\}\}$
$C^1 = \{c_1 = (x_1 \neq x_2), c_2 = (x_1 \neq x_3), c_3 = (x_2 \neq x_3)\}$ (paarweise Ungleichheit)

CSP 2: $P^2 = (X^2, D^2, C^2)$ mit:
$X^2 = \{x_4, x_5, x_6\}$
$D^2 = \{D_4, D_5, D_6 \mid D_4 = D_5 = \{0, 1, 2, 3\}, = D_6 = \{0, 1, 2, 3, 4, 5\}\}$
$C^2 = \{c_4 = (x_4 + x_5 = x_6), c_5 = (x_4 \neq x_5)\}$

Das CSP P^1 *drückt ein sehr prominentes Ungleichheitsproblem aus, bei dem drei Variablen jeweils die Werte null oder eins annehmen können aber gleichzeitig paarweise verschieden sind. Für den Menschen ist leicht ersichtlich, dass dieses CSP keine Lösung haben kann. Nichts desto trotz gibt es für jedes Constraint* c *in* C^1 *(individuell betrachtet) mindestens eine gültige Belegung.*

Das zweite CSP P^2 *beinhaltet ein Constraint* c_4 *mit drei Variablen und ein weiteres* c_5 *mit zwei Variablen. Bei diesem CSP werden Wertetripel gesucht, sodass die ersten beiden Werte ungleich zueinander sind und die Summe der ersten beiden Werte gleich dem dritten Wert ist.*

In Abbildung 2.1 sind die unterschiedlichen Visualisierungsmöglichkeiten für das CSP P^1 *gegenübergestellt. Die Abbildung als Graph (a) kann die Informationen am kompaktesten und eindeutigsten darstellen. Das liegt zum einen daran, dass, wie bei der Hypergraphdarstellung (b), die geringste Anzahl an Knoten benötigt wird. Zum anderen ist, wie bei der bipartiten Darstellung (c), eine klare Linienverfolgung möglich. Allerdings hat diese Darstellungsform das Problem, dass sie für CSPs mit Constraints, die mehr als zwei Variablen enthalten, nicht anwendbar ist. Aus diesem Grund kann sie auch nicht für das CSP* P^2 *verwendet werden. Es ist zu erkennen, dass bei der Hypergraphvariante (b) in Abbildung 2.1 Knoten mit vielen Kanten sehr viele Überlagerungen zur Folge haben, die für den Menschen schwer nachvollziehbar sind.*

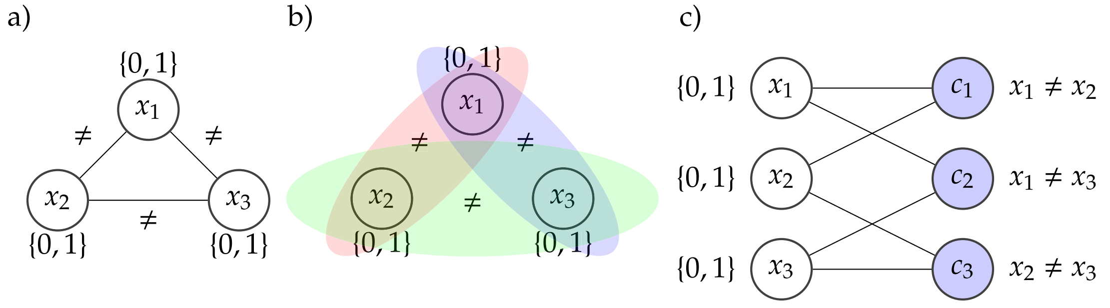


Abbildung 2.1: Das CSP P^1 als klassischer (*a*), Hyper- (*b*) und bipartiter Graph (c).

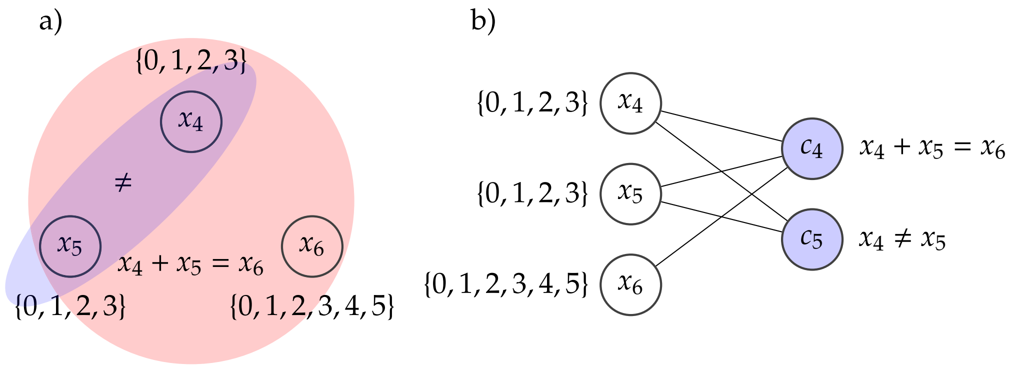


Abbildung 2.2: Das CSP P^2 als Hypergraph (*a*) und bipartiter Graph (b).

Die bipartite Variante (c) ist demgegenüber verständlicher, benötigt dafür aber zusätzliche Knoten für die Constraints (hier blau dargestellt).

In Abbildung 2.2 sind die Constraint-Netze des CSP P^2 als Hypergraph und als bipartiter Graph zu sehen. Im Zuge dieser Arbeit werden alle drei Visualisierungsformen Anwendung finden. Es wird dabei immer die jeweils übersichtlichste Variante gewählt.

2.2 Klassifikation von Constraint-Problemen

Zum Lösen eines Constraint-Problems wird ein sogenannter Solver (Löser) verwendet. Ein Solver ist ein Lösungsalgorithmus für Constraint-Probleme bzw. ein Programm, das diesen Algorithmus umsetzt. Die wesentlichen Aufgaben eines Solvers sind im Allgemeinen das Ermitteln, ob Lösungen existieren und das Ermitteln einer, mehrerer oder aller Lösungen eines Constraint-Problems. Ein vom Programmierer deklarativ modelliertes Constraint-Problem wird dem Solver übergeben, welcher das Problem selbstständig löst, falls eine Lösung vorliegt. Über verschiedene Konfigurationen kann das Verhalten eines Solvers durch den Programmierer beeinflusst werden. In Abschnitt

2.3 wird detailliert auf Solver im Allgemeinen und in Abschnitt 2.4 speziell auf FD-Solver eingegangen.

Constraint-Probleme können nach ihren Domänen und den verwendeten Constraints klassifiziert werden. Dies ist unter anderem notwendig, weil in der Constraint-Programmierung sehr viele verschiedene Konzepte und Algorithmen aus verschiedenen Bereichen der Mathematik und der Informatik vereint werden. Da die verschiedenen Bereiche teilweise nur sehr schwer kombinierbar sind, gibt es keinen allgemeinen Solver, der jede Art von Constraint-Problem am schnellsten löst. Auf der einen Seite muss der Nutzer der Constraint-Programmierung damit im Vorfeld wissen, welche Art von Constraint-Problem vorliegt, um einen geeigneten Solver zu verwenden, auf der anderen Seite ermöglicht dieser Sachverhalt aber auch das Verwenden von sehr spezialisierten Solvern, die schneller und effektiver arbeiten können als es ein allgemeiner Solver je könnte. In diesem Abschnitt wird zunächst auf die Klassifikation von Constraint-Problemen in verschiedene Constraint-Domänen eingegangen, bevor im Anschluss daran, in Abschnitt 2.3, zugehörige Constraint-Solver diskutiert werden.

Die Constraint-Domäne, die ein Solver verwenden kann, wird in der Regel durch Vorschriften zur Bildung von Constraints beschrieben. Solche Vorschriften können unter anderem vorgeben, welche Konstanten, Funktionen und Constraint-Relationen in der jeweiligen Domäne erlaubt sind. Des Weiteren kann die Arität, der Typ und die Funktionalität von Constraints in verschiedenen Constraint-Domänen eingeschränkt werden (nach [37]).

In der Literatur (z.B.[16, 46, 109]) wird zwischen einer Vielzahl von Constraint-Domänen unterschieden. An dieser Stelle wird nur auf die für diese Arbeit relevanten eingegangen. Zusätzliche Informationen zu den vorgestellten und weiteren Constraint-Domänen können unter anderem der zuvor angegeben Literatur entnommen werden.

2.2.1 Boolesche Constraint-Probleme

Boolesche Constraint-Probleme zeichnen sich dadurch aus, dass deren Variablen nur die Werte *Wahr* oder *Falsch* (bzw. 1 oder 0) annehmen können. Die Constraints werden dabei durch boolesche Ausdrücke (z.B. in konjunktiver Normalform) angegeben. Eine Formel der Aussagenlogik ist in konjunktiver Normalform, wenn sie eine Konjunktion von Disjunktionstermen ist. Disjunktionsterme sind dabei Disjunktionen von Literalen und Literale sind wiederum nicht negierte oder negierte Variablen.

Definition 2.11: ***Konjunktive Normalform (KNF).*** Eine Formel der Aussagenlogik ist in konjunktiver Normalform, wenn sie die folgende Form hat: $\bigwedge_i (\bigvee_j (\neg) x_{ij})$.

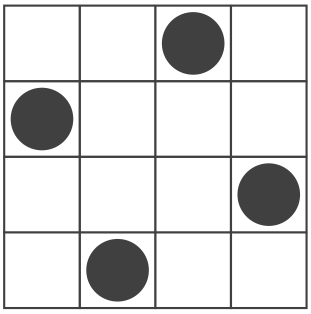


Abbildung 2.3: Eine Lösung des 4-Damen-Problems.

Definition 2.12: *Boolesches CSP.* Ein boolesches Constraint Satisfaction-Problem P^b ist ein Tripel (X^b, D^b, C^b) mit:

$X^b = \{x_1^b, ..., x_n^b\}$ ist eine Menge von n Variablen,

$D^b = \{D_1^b, ..., D_n^b\}$ ist eine Menge von n binären Domänen ($D_1^b = ... = D_n^b = \{False, True\}$ bzw. $= \{0, 1\}$),

$C^b = \{c_1, ..., c_m\}$ ist eine Menge von m Constraints, die sich in konjunktiver Normalform (KNF) darstellen lassen (nach [14, 45]).

Beispiele für aussagenlogische Formeln in KNF (für die booleschen Variablen A, B und C) sind u.a. $A \wedge (\neg B \vee C)$ oder $(A \vee \neg B) \wedge (\neg A \vee C)$. Es existieren weitere binäre und unäre Operationen auf booleschen Ausdrücken, wie z.B. das exklusive Oder ($\oplus$), die Implikation ($\rightarrow$) oder die Äquivalenz ($\leftrightarrow$), allerdings lassen sich alle aussagenlogischen Formeln über booleschen Variablen in KNF überführen.

Das Beispiel 2.2 gibt eine boolesche CSP-Repräsentation des n-Damen-Problems an.

Beispiel 2.2: Das (boolesche) n-Damen-Problem (nach [14]). *Das n-Damen-Problem stellt die Frage, wie n Damen auf einem $n \times n$ Schachbrett positioniert werden müssen, ohne dass eine Dame eine andere attackieren kann. Eine Dame kann beim Schach andere Figuren in beliebiger horizontaler, vertikaler oder diagonaler Richtung attackieren. In Abbildung 2.3 ist eine Lösung des 4-Damen-Problems dargestellt.*

Das Problem lässt sich als boolesches CSP modellieren. Dafür wird für jedes der $n \times n$ Felder auf dem Schachbrett eine boolesche Variable $x_{i,j}^b$ angelegt, die genau dann den Wert Wahr annimmt, wenn sich eine Dame auf dem Feld in Reihe i und Spalte j befindet und ansonsten den Wert Falsch. Für die Beschreibung der Constraints, die das Nicht-Attackieren ausdrücken, legen wir zunächst einen booleschen Ausdruck oneOf($x_1, ..., x_n$) an, der dafür sorgt, dass nur exakt eine der eingegebenen Variablen ($x_1, ..., x_n$) Wahr sein darf. In Gleichung 2.1 ist der boolesche Ausdruck für oneOf angegeben. Mit seiner Hilfe lässt sich im booleschen CSP 3 beschreiben, dass eine Dame nicht in der gleichen Reihe oder in der gleichen Spalte wie eine andere Dame positioniert werden darf.

$$\mathit{oneOf}(x_1, ..., x_n) = (x_1 \wedge \neg x_2 \wedge ... \wedge \neg x_n) \vee (\neg x_1 \wedge x_2 \wedge ... \wedge \neg x_n) \vee ... \vee (\neg x_1 \wedge \neg x_2 \wedge ... \wedge x_n) \quad (2.1)$$

Zwei Damen stehen auch dann zueinander im Konflikt, wenn sie sich eine Diagonale teilen. Dies kann mit Hilfe einer Negation des logischen UND *(auch NAND) formuliert werden. Zwei Variablen können dann nicht gleichzeitig den Wert Wahr annehmen. Dafür muss jedes Variablenpaar* $x_{i,j}$ *und* $x_{k,l}$*, das den gleichen Reihen- wie Spaltenabstand hat, wenn* $|i-k|$ *also gleich* $|j-l|$ *ist, das zuvor beschriebene nicht-gleichzeitig-Wahr-Constraint erfüllen. Das boolesche CSP 3 beschreibt das geschilderte n-Damen-Problem.*

CSP 3: $P^b = (X^b, D^b, C^b)$ mit:

$X^b = \{x^b_{i,j} \mid \forall i, j \in \{1, ..., n\}\}$ (n^2 Variablen)

$D^b = \{D^b_{i,j} = \{\mathit{False}, \mathit{True}\} \mid \forall i, j \in \{1, ..., n\}\}$ (n^2 Domänen)

$C^b = \{\text{oneOf}(x^b_{i,1}, ..., x^b_{i,n}) \mid \forall i \in \{1, ..., n\}\} \cup$ (genau eine Dame pro Reihe)

$\{\text{oneOf}(x^b_{1,j}, ..., x^b_{n,j}) \mid \forall j \in \{1, ..., n\}\} \cup$ (genau eine Dame pro Spalte)

$\{\neg(x^b_{i,j} \wedge x^b_{k,l}) \mid \forall i, j, k, l \in \{1, ..., n\}$ so dass $i \neq k$ und $|i-k| = |j-l|\}$

(maximal eine Dame pro Diagonale)

Es gibt viele verschiedene Ansätze zum Lösen von booleschen CSPs (auch SAT-Probleme genannt, vom Englischen satisfiability problem). Die Frage, ob ein SAT-Problem erfüllbar ist oder nicht, ist eines der bekanntesten NP-vollständigen Probleme [14]. NP-vollständige Probleme stellen Probleme dar, für die es bisher keine Algorithmen gibt, die diese deterministisch in Polynomzeit lösen können. Sie sind also **n**ichtdeterministisch in **P**olynomialzeit (NP) lösbar. Vollständig bedeutet in diesem Zusammenhang, dass alle Probleme in *NP* in Polynomialzeit in ein SAT-Problem überführt werden können. Sollte also ein Algorithmus gefunden werden, der SAT-Probleme deterministisch in Polynomialzeit löst, so wurde gezeigt, dass die gesamte Klasse der NP-Probleme in Polynomialzeit lösbar ist.

Auf der einen Seite macht die Überführbarkeit anderer Probleme in SAT-Probleme diese für die Wissenschaft sehr interessant. Auf der anderen Seite können zum Beispiel elektrische Schaltungen sehr gut mit SAT-Problemen beschrieben werden, was diese auch für die Industrie zu einer interessanten Problemstellung macht.

Pseudo-Boolesche Constraints

SAT-Probleme haben in der Regel das Problem, dass sie über Klauseln von Literalen definiert werden müssen. Manche Solver lassen zusätzlich noch *atLeastM*- oder *atMostM*-Constraints zu, die für eine geordnete Menge von booleschen Variablen $X = \{x_1, ..., x_n\}$ fordern, dass mindestens bzw. maximal $m, m \in N, m \leq n$ viele Variablen den Wert 1 (Wahr) annehmen müssen bzw. dürfen. Ein Problem, das die Modellierung von SAT-Problemen erschwert, ist die eingeschränkte Verwendung der vorgestellten Constraint-Typen.

Eine Constraint-Domäne, die eine einfachere Modellierung als durch das pure Angeben von Klauseln erlaubt, stellt die Domäne der linearen Pseudo-Booleschen-Constraints dar. Ein lineares Pseudo-Boolesches-Constraint ist eine lineare Relation über booleschen Variablen.

Definition 2.13: *Lineare pseudo-boolesche Constraints.* Ein lineares pseudo-boolesches Constraint ist ein Constraint über booleschen Variablen $x_1, ..., x_n$, das über die Relation $\sum_{i=1}^{n}(l_i * c_i)\ \mathfrak{R}\ r$ definiert ist mit $\forall c_i, r \in \mathbb{Z}$, $\mathfrak{R} \in \{<, \leq, =, \geq, >, \neq\}$ und l_i ist ein Literal, das entweder gleich x_i oder $\bar{x}_i$ ist ($\bar{x}_i = 1 - x_i$) [23].

Beim Lösen von linearen pseudo-booleschen CSPs werden diese häufig in SAT-CSPs überführt. Lineare pseudo-boolesche CSPs erweitern und erleichtern somit die Möglichkeiten der Modellierung von SAT-Problemen.

2.2.2 Finite Domain Constraint-Probleme

Eine besondere Bedeutung in der Constraint-Programmierung haben Probleme mit endlichen Domänen (*engl.* finite domain constraint problems, kurz FD-Probleme). Für deren Berechnung werden häufig ganze Zahlen in den Domänen verwendet, diese können aber auch beliebig durch andere Symbole ersetzt werden. Ein Finite Domain Constraint-Problem unterscheidet sich von der Definition eines herkömmlichen Constraint Satisfaction-Problems lediglich durch die Einschränkung, dass alle Domänen eine endliche Größe haben müssen. Für die Modellierung eines Constraint-Problems macht es aus Nutzersicht in der Regel keinen Unterschied, ob die Domänen endlich sind oder nicht. Für den verwendeten Solver hat dies allerdings einen signifikanten Unterschied zur Folge, wie in Abschnitt 2.3 verdeutlicht wird.

Definition 2.14: *Finite Domain-CSP.* Ein Finite Domain Constraint Satisfaction-Problem (FD-CSP) P ist ein Tripel (X, D, C) mit:

$X = \{x_1, ..., x_n\}$ ist eine Menge von n Variablen,
$D = \{D_1, ..., D_n\}$ ist eine Menge von n *endlichen* Domänen,
$C = \{c_1, ..., c_m\}$ ist eine Menge von m Constraints (nach [45]).

Die im vorherigen Abschnitt vorgestellten booleschen CSPs sind eine Variante der Finite Domain-CSPs, bei der die Variablen-Domänen auf zwei Werte (Wahr und Falsch bzw. 1 und 0) beschränkt sind. Prinzipiell kann jedes boolesche CSP mit einem FD-Solver gelöst werden. In der Regel sind FD-Solver bei der Lösung von Modellen, die für einen booleschen Solver angefertigt wurden, allerdings langsamer als entsprechende boolesche Solver. Dafür ermöglichen FD-Solver durch das Verwenden von größeren Domänen andere Modellierungen, die ihrerseits gegebenenfalls schneller gelöst werden können. Anhand des n-Damen-Problems wird gezeigt, wie ein Problem für einen FD-Solver schneller lösbar modelliert werden kann (siehe Beispiel 2.3).

Beispiel 2.3: Das (FD) n-Damen-Problem (nach [14]). *Es wird die gleiche Problemstellung wie in Beispiel 2.2 betrachtet. An dieser Stelle dürfen Domänen nicht nur binär, sondern auch nicht-binär, aber endlich sein. Aus der Problemstellung des n-Damen-Problems geht hervor, dass* *nur eine Dame pro Spalte auftreten darf. Das erlaubt eine Abstraktion des Problems auf lediglich eine Variable x_i für jede der n Spalten, deren Wert $d_i \in D_i$ angibt, in welcher Zeile innerhalb der i-ten Spalte sich die Dame befindet. Jede Variable hat dabei eine Domäne mit Werten von 1 bis n. Die Constraints unterscheiden sich von der Bedeutung her nicht von denen aus der booleschen Implementierung. Es muss wieder garantiert werden, dass in jeder Spalte, jeder Zeile und jeder Diagonalen nur maximal eine Dame vorkommt.*

Die Bedingung, dass sich nur eine Dame pro Spalte befinden darf, wird bereits durch die Art der Variablen und Domänen sichergestellt, deswegen ist dafür kein zusätzliches Constraint notwendig. Für das Kriterium, dass nur eine Dame pro Zeile platziert werden darf, müssen die Variablenwerte paarweise disjunkt sein. Um zu vermeiden, dass die Damen sich diagonal attackieren, muss sichergestellt werden, dass der Betrag der Differenz der Indizes zweier Variablen (der Spaltenabstand der entsprechenden Damen auf dem Spielfeld) nicht gleich dem Betrag der Differenz der Werte dieser Variablen (der Zeilenabstand der entsprechenden Damen auf dem Spielfeld) ist. Das FD-CSP 4 zeigt eine mögliche Modellierung des Problems.

CSP 4: $P = (X, D, C)$ mit:

$X = \{x_1, ..., x_n\}$ (n Variablen)
$D = \{D_1, ..., D_n | D_1 = ... = D_n = \{1, ..., n\}\}$ (n Domänen)
$C = \{x_i \neq x_j \mid \forall i, j \in \{1, ..., n\}, i \neq j\} \cup$ (genau eine Dame pro Reihe)
$\{|x_i - x_j| \neq |i - j| \mid \forall i, j \in \{1, ..., n\}, i \neq j\}$ (maximal eine Dame pro Diagonale)

Zur Unterstreichung der Ausdrucksstärke von FD-CSPs ist am Ende dieses Kapitels in Abschnitt 2.5 ein weiteres Beispiel 2.5 vorgestellt. In diesem wird das in Kapitel 1 vorgestellte Schichtplanungsproblem (Beispiel 1.1) aufgegriffen und als FD-CSP modelliert.

2.2.3 Constraint-Probleme über reellen Zahlen

Constraint-Probleme über reellen Zahlen (reelle CSPs) stellen von den vorgestellten Problemen diejenigen dar, die potentiell den größten Suchraum aufspannen, da jede Domäne für sich bereits über unendlich viele Werte verfügt. Boolesche- und FD-CSPs stellen eine Variation von reellen CSPs dar, allerdings sind keine effektiven Solver bekannt, die für beliebige CSPs (mit beliebigen Constraints) über reellen Zahlen Lösungen finden. Das Hauptproblem liegt darin, dass für die unterschiedlichen Constraint-Domänen unterschiedliche Algorithmen existieren, die teilweise nicht in die jeweils anderen Constraint-Domänen überführt werden können.

Eine weitere Schwierigkeit von Constraint-Problemen über reellen Zahlen ist der Umgang mit unendlich großen Domänen, die ein vollständiges Absuchen des Lösungsraumes nicht ermöglichen. Aus diesem Grund müssen zur Lösung von reellen CSPs spezielle Solver verwendet werden. Diese speziellen Solver sind entweder unvollständig, finden also nicht immer eine Lösung, falls eine solche existiert, oder erlauben nur eine sehr beschränkte Auswahl an Constraints bei der Modellierung der Constraint-Probleme.

Eine sehr bekannte Klasse der reellen CSPs ist die Klasse der CSPs mit ausschließlich linear arithmetischen Constraints (linear arithmetische CSPs), also Constraints, die nur lineare Gleichungen und Ungleichungen repräsentieren.

Definition 2.15: ***Lineares arithmetisches CSP.*** Unter einem linear arithmetischen Constraint Satisfaction-Problem (lineares arithmetisches CSP) P^R wird ein Tripel (X^R, D^R, C^{la}) verstanden, mit:
$X^R = \{x_1^R, ..., x_n^R\}$ ist eine Menge von n Variablen,
$D^R = \{D_1^R, ..., D_n^R\}$ ist eine Menge von n Domänen mit $D_i \subseteq \mathbb{R}\ \forall i \in \{1, ..., n\}$,
$C^{la} = \{c_1^{la}, ..., c_m^{la}\}$ ist eine Menge von m Constraints mit $c_j^{la} = (\sum_{i \in \{1,...,n\}} a_i * x_i^R)\ \Re\ r \mid \forall a_i \in \mathbb{R}, i \in \{1, ..., n\}, \Re \in \{=, <, \leq\}, r \in \mathbb{R}$ [106].

Die Erfüllbarkeit eines linear arithmetischen Constraint Satisfaction-Problems kann mit Hilfe der Simplex-Methode [51, 120, 163] oder anderen Verfahren, wie der Gauß-Jordan-Elimination oder der Fourier-Motzkin-Elimination ermittelt werden [74].

2.2.4 Weitere Constraint-Domänen

Neben den bereits vorgestellten Constraint-Domänen existieren noch weitere, welche für diese Arbeit allerdings weniger von Bedeutung sind. Exemplarisch wird an dieser Stelle lediglich auf zwei weitere Vertreter eingegangen, die sowohl in der Forschung als auch in der Praxis Anwendung finden.

Mixed Integer-Programmierung

In der Mixed Integer-Programmierung (MIP) werden CSPs behandelt, die gleichzeitig sowohl FD-Variablen als auch Realzahl-Variablen beinhalten. Diese hybride Variante stellt eine vollkommen andere Problemstellung dar, die sich weder direkt mit den Methoden der Finite Domain Constraint-Probleme noch mit den Methoden der reellen Constraint Satisfaction-Probleme lösen lässt. Aus diesem Grund stellt die Mixed-Integer-Programmierung ein eigenes, interessantes Forschungsfeld dar. Weitere Informationen zur MIP können zum Beispiel [8] entnommen werden.

Eine Spezialisierung der Mixed Integer-Programmierung ist die lineare Mixed Integer-Programmierung (MILP von Mixed Integer Linear Programming), die nur linear arithmetische Constraints erlaubt [8].

Constraints über Mengenvariablen

Constraints über Mengenvariablen (Set-Variablen) erlauben das Einschränken von Variablen, die jeweils eine Menge von Werten repräsentieren. Ziel ist es nicht, jeder Variablen genau einen Wert ihrer Domäne, sondern eine Teilmenge ihrer ursprünglichen Domäne zuzuordnen. Insbesondere erlaubt das den Einsatz von Operationen aus der Mengenlehre, wie zum Beispiel den Schnitt oder die Vereinigung von zwei Mengen [9].

Weitere erwähnenswerte Constraint-Domänen sind u.a. die Baum- und Graph-Constraints [106] sowie String-Constraints [11], auf welche in dieser Arbeit allerdings nicht weiter eingegangen wird.

2.3 Constraint-Solver

Viele Lösungsverfahren für Constraint-Probleme basieren auf Algorithmen, die spezielle Einschränkungen an den Wertebereich oder die zu verwendenden Constraints haben. Aus diesen lassen sich die bereits vorgestellten Constraint-Domänen ableiten. Abhängig von den verwendeten Constraint-Domänen und den damit verbundenen Lösungsstrategien existieren viele verschiedene Solver, die für das Lösen von Constraint-Problemen verwendet werden können. In diesem Abschnitt wird zunächst der Begriff des Solvers eingeführt, bevor verschiedene Solver genannt werden und im folgendem Abschnitt 2.4 ausführlich auf Lösungstechniken von Finite Domain-Solvern (FD-Solvern) eingegangen wird.

2.3.1 Klassifikation von Solvern

Lösungsalgorithmen für Constraint-Systeme werden Solver genannt. Bei diesen wird, neben der Unterteilung nach ihren Constraint-Domänen, unter anderem auch zwischen *vollständigen* und *unvollständigen*, *allgemeinen* und *domänenspezifischen* sowie *polynomialen* und *nicht polynomialen* Solvern unterschieden [16, 46, 109].

Ein Solver ist *vollständig*, wenn für jedes Eingabe-CSP eindeutig festgestellt werden kann, ob es erfüllbar ist (Ausgabe *True*) oder nicht (Ausgabe *False*) und im Fall der Erfüllbarkeit eine Lösung des CSPs ermittelt werden kann. Ein *unvollständiger* Solver hingegen kann zusätzlich den Wert unbekannt (*Unknown*) ausgeben und garantiert nicht, dass eine oder sogar alle existierenden Lösungen gefunden werden. Da der Suchraum von Constraint-Problemen oftmals riesig, teilweise sogar unendlich ist und für manche Probleme keine vollständigen Solver existieren, kommt *unvollständigen* Solvern in der Constraint-Programmierung eine ebenso große Bedeutung zu wie vollständigen Solvern.

Allgemeine Constraint Solver, wie zum Beispiel Brute-Force-Solver, die jeder Variablen einen zufälligen Wert ihrer Domäne zuordnen, sind für verschiedene Constraint-Domänen nutzbar, im Gegensatz dazu sind *domänenspezifische* Solver nur für eine bzw. wenige Domänen einsetzbar. Da für viele Domänen oder Fragestellungen keine vollständigen Lösungsalgorithmen bekannt sind, müssen allgemeine Solver immer unvollständig sein, wohingegen bei domänenspezifischen Solvern die Vollständigkeit möglich ist.

Polynomiale Solver können Lösungen von CSPs in polynomialer Zeit finden. *Nicht-polynomiale* Solver können dies nicht gewährleisten. Da CSPs im allgemeinen NP-schwer sind, treten *polynomiale* Solver in der Regel nur für sehr beschränkte Domänen oder als unvollständige Solver auf.

Constraint-basierte Programmiersysteme

Um einen Constraint Solver nutzen zu können, muss zunächst eine Modellierung in einem Constraint-basierten Programmiersystem erstellt werden. Constraint-basierte Programmiersysteme können als Constraint-logische Programmiersprachen, Constraint-basierte Modellierungssprachen oder Constraint-Bibliotheken vorliegen [73, 71, 72].

Constraint-logische Sprachen

Constraint-logische Sprachen stellen eine Generalisierung der logischen Sprachen dar, Vertreter sind zum Beispiel ECL^{i}PS^e-Prolog [17, 115], SICStus-Prolog [38] und SWI-Prolog [161].

Constraint-basierte Modellierungssprachen

Constraint-basierte Modellierungssprachen wie OPL [68], Comet [153] oder der vielleicht bekannteste Vertreter MiniZinc [147, 149] stellen ebenfalls eine sehr erfolgreiche Entwicklung der Constraint-Programmierung dar.

MiniZinc [147] strebt z.B. eine Standardisierung der Constraint-Programmierung an, integriert verschiedene Löser und unterstützt hybride Lösungstechniken. Mittels MiniZinc können Spezifikationen von CSPs und Optimierungsproblemen über ganzen und reellen Zahlen formuliert werden. Mittels Annotationen können den Constraint-Modellen Solver-spezifische Informationen hinzugefügt und ein unterliegender Solver angesteuert werden. MiniZinc-Modelle werden nach FlatZinc transformiert, das eine Spezialisierung für individuelle Backend Solver unterstützt. Dies ermöglicht es, verschiedene Constraint-Löser auf einfache Weise als Backend zu integrieren [73].

Constraint-Bibliotheken

Constraint-Bibliotheken sind funktionale und syntaktische Erweiterungen existierender Sprachen um Constraints. Häufig wird dieser Ansatz für die Erweiterung objektorientierter Sprachen wie Java und C++, aber auch in funktionalen Sprachen, beispielsweise durch das MCP-Framework (Monadische Constraint-Programmierung, [142]) umgesetzt.

Die grundlegenden Konzepte der Constraint-Programmierung (wie zum Beispiel Constraints, Constraint-Löser, Suchmechanismen, Heuristiken etc.) werden dabei als Klassen bzw. Objekte direkt in der Host-Sprache modelliert. Da objektorientierte Sprachen wie Java und C++ in der Praxis anerkannt und verbreitet sind, erfahren diese Bibliotheken eine hohe Nutzerakzeptanz und sind ein Grundstein des Erfolgs Constraint-basierter Bibliotheken.

2.3.2 Eine Solver-Auswahl

Anhand der vielen verschiedenen Klassifikationsmöglichkeiten von Constraint-Solvern unter anderem nach Constraint-Domäne, Vollständigkeit, Allgemeingültigkeit, Komplexität und Constraint-basiertem Programmiersystem lässt sich ableiten, dass es nicht einen perfekten Solver zum Lösen aller Constraint-Probleme gibt. Besonders die Einteilung nach Constraint-Domänen erfordert den Gebrauch verschiedener Solver, weswegen an dieser Stelle Vertreter für die zuvor genannten Domänen angegeben werden, bevor im nächsten Kapitel die Funktionsweise der für diese Arbeit essentiellen FD-Solver erläutert wird.

FD-Solver

Zu den bekanntesten Finite Domain-Solvern zählen die C++-Bibliothek Gecode [42] und die Java-Bibliotheken Choco [129] und JaCoP [3], die durch ihre objektorientierte Programmierung eine hohe Akzeptanz finden. Ein weiterer Vertreter sind die Google OR-Tools [2], die durch Erfolge bei der jährlich stattfindenden MiniZinc-Challenge [148] auf sich aufmerksam machen konnten. Die Google OR-Tools stellen eine C++-Bibliothek dar, die aber auch Python, C# und Java unterstützt.

Etwas unbekannter sind dagegen die Constraint-basierten Modellierungssprachen Picat [165] und OscaR [119], die als eigenständige Solver fungieren. Als FD-Solver für Constraint-logische Sprachen sei noch der clp(FD)-Solver erwähnt, der in [82] eingeführt und in den Sprachen ECL^{i}PS^e-Prolog [17, 115], SICStus-Prolog [38] und SWI-Prolog [161] verwendet wird.

Boolesche Solver

Der MiniSat-Solver [52] stellt die Grundlage für den ECL^{i}PS^e-Prolog- und den Choco-Solver dar. Demgegenüber verwenden SICStus-Prolog und SWI-Prolog den clp(B)-Solver, der in [82] beschrieben ist und auf der Reduktion und Ordnung von binären Entscheidungsdiagrammen basiert. *SAT4J* [5] ist eine Java-Bibliothek, die sich ebenfalls auf das Lösen von booleschen CSPs spezialisiert hat. OscaR [119] und Picat [165] stellen zwei weitere boolesche Solver bereit. Boolesche Solver, deren Modelle über Klauselmengen definiert werden, werden auch SAT-Solver genannt.

Lineare pseudo-boolesche Solver

Lineare pseudo-boolesche Solver basieren auf der Idee, linear-Constraints in Klauselmengen zu überführen. Aus diesem Grund sind alle SAT-Solver unter Annahme einer Transformation von linearen pseudo-booleschen Constraints in eine Klauselmenge auch lineare pseudo-boolesche Solver. Vor der Transformation können im Allgemeinen Verfahren zur Reduzierung der Gleichungen und Ungleichungen angewendet werden.

Der MiniSat-Solver stellt mit dem MiniSat+-Solver, ebenso wie die *SAT4J*-Java-Bibliothek, einen Solver für boolesche CSPs mit pseudo-booleschen Constraints zur Verfügung.

Solver über reellen Zahlen

Ein sehr populärer Vertreter für Constraint-logische Sprachen ist der ebenfalls in [82] vorgestellte clp(Q,R)-Solver, welcher in den Prolog-Varianten SICStus-Prolog, SWI-Prolog und ECL^{i}PS^e-Prolog verwendet wird. Der IBEX-Solver [80] und die Google OR-Tools enthalten zwei weitere C++-Bibliotheken zur Lösung von CSPs über reellen Zahlen.

In dieser Arbeit wird primär der Choco-Solver zum Lösen von Finite Domain-CSPs verwendet. Die im Folgenden vorgestellten Algorithmen sind allerdings Solver-unabhängig gestaltet. Da FD-Solver die Grundlage für diese Arbeit darstellen, werden im folgenden Abschnitten die grundlegenden Komponenten von FD-Solvern erläutert.

2.4 FD-Solver

Vollständige FD-Solver wie zum Beispiel Choco, Gecode, JaCoP, Google OR-Tools oder auch clp(FD) arbeiten mit einer auf Backtracking-basierenden Tiefensuche, verzahnt mit der Propagation der einzelnen Constraints. In den folgenden Abschnitten werden daher zunächst die Konzepte der Suche in Abschnitt 2.4.1 und der Propagation und Konsistenz in Abschnitt 2.4.2 und anschließend die Interaktion der beiden Konzepte in Abschnitt 2.4.3 detailliert beschrieben.

2.4.1 Suche

Eine Methode, ein CSP mit endlichem Suchraum (zum Beispiel ein FD- oder boolesches CSP) zu lösen, besteht darin, mittels einer vollständigen, naiven Suche den gesamten Lösungsraum nach der ersten, n vielen, allen oder der besten Lösung zu durchsuchen. In der Regel wird dabei eine Tiefensuche verwendet. Eine Breitensuche kommt auf Grund des dafür benötigten immensen Speicherbedarfs (b^d wobei b der Verzweigungsfaktor und d die Tiefe des Baumes darstellt) in der Regel nicht in Frage.

An dieser Stelle wird ein CSP $P = (X, D, C)$ mit n Variablen $\{x_1, ..., x_n\} = X$ und den zugehörigen n Domänen $\{D_1, ..., D_n\} = D$ sowie m Constraints $\{c_1, ..., c_m\} = C$ als gegeben betrachtet. Bei der Verwendung der naiven Tiefensuche weist der Constraint-Solver jeder Variablen $x_i \in X$ nacheinander einen Wert aus ihrer Domäne $d_i \in D_i$ zu. Wurde allen Variablen auf diese Weise ein Wert zugeordnet, so wird überprüft, ob die resultierende Variablenbelegung $\phi \in X \rightarrow \mathbb{N}^n$ eine Lösung des CSPs P darstellt. Dafür muss überprüft werden, ob alle Constraints mit der Belegung ϕ erfüllt sind. Ist dies der Fall, so wurde ϕ als erste Lösung gefunden.

Wurde keine Lösung gefunden, ist durch die Belegung ϕ also mindestens ein Constraint nicht erfüllt, so wird Backtracking durchgeführt. Das bedeutet, dass die letzte Zuweisung $\phi(x_i) = d_i$ rückgängig gemacht wird und der Variablen x_i ein anderer in ihrer Domäne vorkommender Wert $d'_i \in D, d_i \neq d'_i$, zugeordnet wird. Sollte kein solcher Wert $d'_i \in D_i$ mehr verbleiben, der der Variablen x_i noch nicht zugeordnet wurde, so breitet sich das Backtracking zur vorherigen Variable x_{i-1} aus und widerruft auch deren Zuweisung. Das Backtracking wiederholt sich solange, bis für eine Variable eine noch nicht probierte Zuweisung gefunden wurde oder für mindestens eine Variable keine Zuweisung existiert, die Teil einer gültigen Belegung ist. Im zweiten Fall folgt, dass der gesamte Suchraum durchsucht worden ist. Wurde bis dahin keine Lösung gefunden, so hat das CSP P keine Lösung.

Anhand von CSP 5 soll das Vorgehen bei der Suche veranschaulicht werden. Das CSP 5 mit $P = (X, D, C)$ umfasst drei Variablen x_1, x_2 und x_3, wovon die ersten beiden die Domäne $D_1 = D_2 = \{0, 1\}$ haben und die dritte die Domäne $D_3 = \{0, 1, 2\}$ hat. Weiterhin beinhaltet das CSP drei Constraints, die widerspiegeln, dass die Variablen x_1 und x_2 sowie x_2 und x_3 unterschiedliche Werte und die Variable x_1 einen kleineren Wert als die Variable x_3 annehmen müssen.

CSP 5: $P = (X, D, C)$ mit:
$X = \{x_1, x_2, x_3\}$
$D = \{D_1 = \{0, 1\}, D_2 = \{0, 1\}, D_3 = \{0, 1, 2\}\}$
$C = \{c_1 = (x_1 \neq x_2), c_2 = (x_1 < x_3), c_3 = (x_2 \neq x_3)\}$

In Abbildung 2.4 ist der vollständige Suchbaum für das CSP dargestellt. Im Wurzelknoten befinden sich dabei die Variablen (x_1, x_2, x_3) des Problems mit den zugehörigen Domänen. Von jedem Knoten wird über eine Variablenzuweisung zum jeweiligen Kindknoten gelangt. Es wird in dieser Arbeit davon ausgegangen, dass die Suchbäume von oben nach unten und links nach rechts durchsucht werden. Demzufolge werden in Abbildung 2.4 auf dem Weg vom Wurzelknoten hin zu einem Blattknoten nacheinander den Variablen x_1, x_2 und x_3 Werte zugeordnet.

Der erste Blattknoten hat die Variablenbelegung $\phi_1(x_1) = 0, \phi_1(x_2) = 0, \phi_1(x_3) = 0$. Für diese Belegung wird geprüft, ob alle Constraints $c \in C$ erfüllt sind. Da dies nicht der Fall ist, stellt die Belegung ϕ_1 keine Lösung des Problems dar, was mit einem roten „X“ in der Abbildung gekennzeichnet ist. Für die weitere Suche wird die letzte Zuweisung (für x_3) verworfen (Backtracking) und der Variablen x_3 ein anderer Wert ihrer Domäne zugeordnet (in diesem Fall der Wert 1). Dieses Vorgehen wird wiederholt, bis alle Domänenwerte für x_3 geprüft wurden. Anschließend muss auch die vorherige Zuweisung (für x_2) durch eine noch nicht geprüfte Zuweisung ersetzt werden. Im dargestellten Beispiel wurden die Variablenbelegungen $\phi_6(x_1) = 0, \phi_6(x_2) = 1, \phi_6(x_3) = 2$ und $\phi_9(x_1) = 1, \phi_9(x_2) = 0, \phi_9(x_3) = 2$ als einzige Lösungen des CSPs ermittelt.

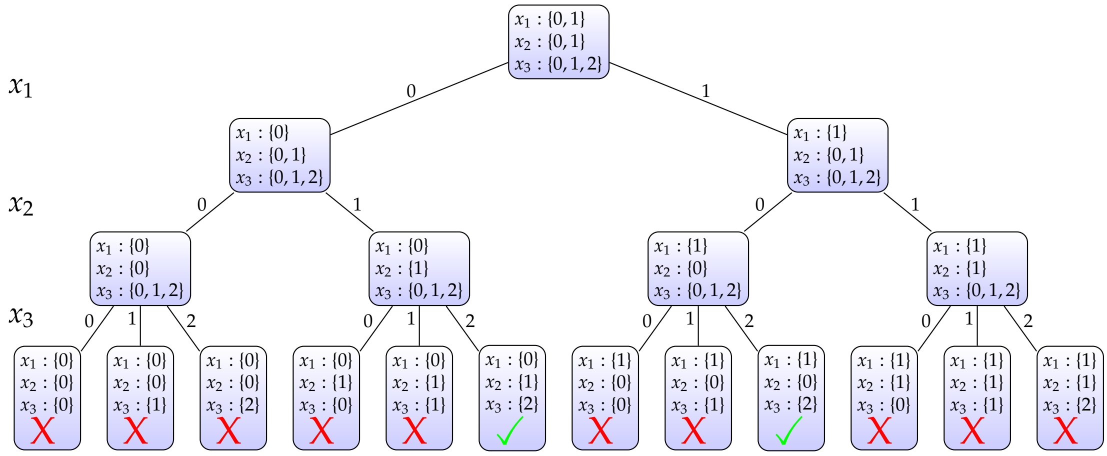


Abbildung 2.4: Ein vollständiger Suchbaum für das CSP 5.

Für die Suche können Variablenauswahl- und Werteauswahlheuristiken verwendet werden. Sie legen fest, in welcher Reihenfolge die Variablen belegt werden und in welcher Reihenfolge die Domänenwerte zugewiesen und geprüft werden. Im Suchbaum in Abbildung 2.4 wurde als Variablenauswahlkriterium die Indexreihenfolge der Variablen und als Werteauswahlkriterium der kleinste in der Domäne verbleibende Wert gewählt. Bei der naiven, Backtracking-basierten Tiefensuche, wie sie zuvor vorgestellt worden ist, hat die Wahl der Variablenauswahl- und Werteauswahlheuristiken keinen Einfluss auf die Anzahl der traversierten Suchbaumknoten, wenn nach allen Lösungen gesucht wird. Wird die Suche allerdings in Kombination mit Propagation eingesetzt, wie sie in Abschnitt 2.4.3 vorgestellt wird, so hat die verwendete Variablenauswahlheuristik einen teils signifikanten Einfluss auf die Größe des Suchbaumes.

Die Werteauswahlheuristik beeinflusst, in welcher Reihenfolge die einzelnen Knoten traversiert werden. Damit hat sie hauptsächlich bei der Suche nach einer, *n* vielen oder einer besten Lösung einen Einfluss auf die Anzahl der zu traversierenden Knoten im Suchbaum. Wird zum Beispiel vom gleichen Variablenauswahlkriterium ausgegangen, wie zuvor angegeben, das Werteauswahlkriterium allerdings ausgetauscht, so dass zunächst jeder Variablen der größte Wert ihrer Domäne zugeordnet wird, so wäre der neue Suchbaum eine vertikale Spiegelung des Suchbaumes aus Abbildung 2.4. Bei der Suche nach der ersten Lösung würden dabei nur drei vollständige Variablenbelegungen (Blätter) überprüft werden müssen, die keine Lösung des CSPs darstellen, wohingegen es bei der ursprünglichen Variante fünf solcher Belegungen sind.

Variablenauswahlheuristiken

Grundsätzlich wird bei den Variablenauswahlheuristiken zwischen statischen und dynamischen Verfahren unterschieden [74]. Bei der statischen Variablenauswahl steht die Reihenfolge der Variablen beim Start der Suche bereits fest, wohingegen bei der dynamischen Variablenauswahl die Reihenfolge der Variablen erst während der Suche unter Berücksichtigung der Informationen, die aus der bisherigen Suche und den Propagationen gewonnen werden konnten, ermittelt wird.

Statische Variablenauswahlheuristiken benötigen während der Suche keine (oder nur sehr wenig) Rechenzeit zur Ermittlung der nächsten zu belegenden Variablen. Aus diesem Grund können die einzelnen Suchschritte sehr schnell durchgeführt werden. Dynamische Variablenauswahlheuristiken benötigen zwischen den einzelnen Suchschritten Rechenzeit zur Ermittlung der nächsten zu belegenden Variablen. Auf den ersten Blick mag das nachteilig wirken, allerdings kann dieses Vorgehen zu einer Reduktion des Suchbaumes (Verringerung der Anzahl der Knoten) führen, wenn die Suche nicht naiv, sondern verzahnt mit Propagation durchgeführt wird.

Vertreter der statischen Variablenauswahlheuristiken treffen ihre Wahl zum Beispiel auf Basis der Eingabereihenfolge, einer vorgegebenen Indexreihenfolge, der lexikographischen Reihenfolge der Variablennamen, der Variablen sortiert nach ihrer Domänengröße zu Beginn des Suchvorganges, ihres größten Domänenwertes oder der Anzahl der Constraints, an denen sie beteiligt sind.

Dynamische Vertreter basieren zum Beispiel auf der Berechnung der Variablen sortiert nach ihrer Domänengröße oder ihres größten Domänenwertes zum aktuellen Zeitpunkt der Suche, oder basieren auf komplexeren Berechnungen wie zum Beispiel die in [35] vorgestellten Konflikt-gerichteten Variablenauswahlheuristiken *wdeg* und *dom/wdeg*. Die letztgenannte Heuristik ist zum Beispiel die Standard-Variablenauswahlheuristik im Choco-Solver.

Werteauswahlheuristiken

Werteauswahlheuristiken spielen auf den ersten Blick eine untergeordnetere Rolle als Variablenauswahlheuristiken, was vor allem daran liegt, dass sie keinen Einfluss auf die Suchbaumgröße haben, wenn nach allen Lösungen mittels naiver Backtracking-basierter Suche oder Backtracking-basierter Suche verzahnt mit Propagation gesucht wird. Allerdings sind Constraint-Probleme in der Regel so groß, dass meist nur nach der besten, einer oder einer bestimmten Anzahl von Lösungen gesucht wird.

Generell beeinflusst die Werteauswahlheuristik die Reihenfolge der Kindknoten in jeder Ebene des Suchbaums. Somit können durch Auswahl der Werteauswahlheuristik bestimmte Bereiche des Suchbaumes, in denen zum Beispiel besonders viele Lösungen liegen, eher oder später durchsucht werden.

Wird während der Suche zusätzliches Wissen gewonnen und ausgenutzt, zum Beispiel durch Lernen mittels Nogoods [47, 28], oder handelt es sich um ein COP, so kann die Wertauswahlheuristik auf die Größe und die Struktur des Suchbaumes und somit auch auf die Lösungsgeschwindigkeit des Constraint-Problems einen erheblichen Einfluss haben.

Für ein zu minimierendes COP kann beispielsweise aus einer ersten Lösung eine obere Schranke gewonnen werden. Mit dieser können bei der Suche nach besseren Lösungen ggf. bestimmte Zweige des Suchbaums, die keinen besseren Zielwert mehr erreichen können, erkannt und ausgeschlossen werden [73]. Weitere Informationen zu Suchstrategien können unter anderem aus [29, 35] entnommen werden.

Varianten der Suche

Die vollständige Suche mittels Backtracking (in Kombination mit Propagation) stellt eine der häufigsten Vorgehensweisen zum Lösen von Constraint-Problemen dar. Es existieren allerdings weitere Möglichkeiten und Abwandlungen.

Zum Beispiel kann das Erzeugen der Kindknoten in der Suche auf verschiedene Art und Weise erfolgen. In Abbildung 2.4 wurde ein Aufzählungsverfahren gewählt, bei dem ein Knoten mit als nächster zu belegender Variable x_i mit zugehöriger, noch verbliebener Domäne $D_i = \{d_1, ..., d_j\}$ genau $|D_i|$ viele Kindknoten hat, die über die Belegung der Variable x_i mit den Werten d_1 bis d_j erreicht werden können.

Alternativ wäre an dieser Stelle auch eine binäre Unterteilung (*Binary Choice*) oder eine Aufteilung der Domäne (*Domain Splitting*) möglich gewesen. Bei der binären Unterteilung hat ein Knoten in der Regel zwei Kinder, das eine wird erreicht, indem die Variable x_i auf den Wert $d \in D_i$ gesetzt wird und das andere, indem der Wert d aus der Domäne D_i von der Variablen x_i entfernt wird. In der Abbildung 2.5 ist ein Ausschnitt eines Suchbaumes dargestellt, der durch binäre Unterteilung erzeugt wurde.

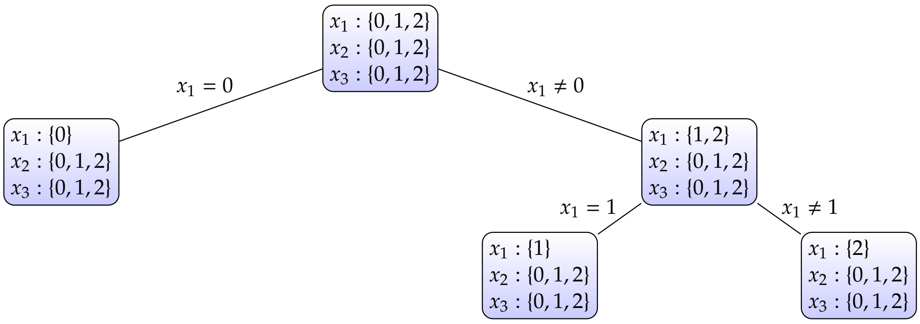


Abbildung 2.5: Ein Ausschnitt eines Suchbaumes, der durch binäre Unterteilung entsteht.

Bei der Zerlegung der Domänen kann die maximale Anzahl der Kindknoten b angegeben werden. In der Regel wird dafür der Wert zwei gewählt und die Domäne D_i in der Mitte geteilt. Das linke Kind enthält dann die ersten $h_1 = \lceil \frac{|D_i|}{2} \rceil$ verbleibenden Domänenwerte

$(d_1, ..., d_{h_1})$ und das rechte Kind die restlichen $h_2 = \lfloor \frac{|D_i|}{2} \rfloor$ verbleibenden Domänenwerte $(d_{h_1+1}, ..., d_{|D_i|})$ der Variablen x_i. In Abbildung 2.6 ist ein Ausschnitt eines Suchbaumes dargestellt, der durch Domain Splitting erzeugt wurde.

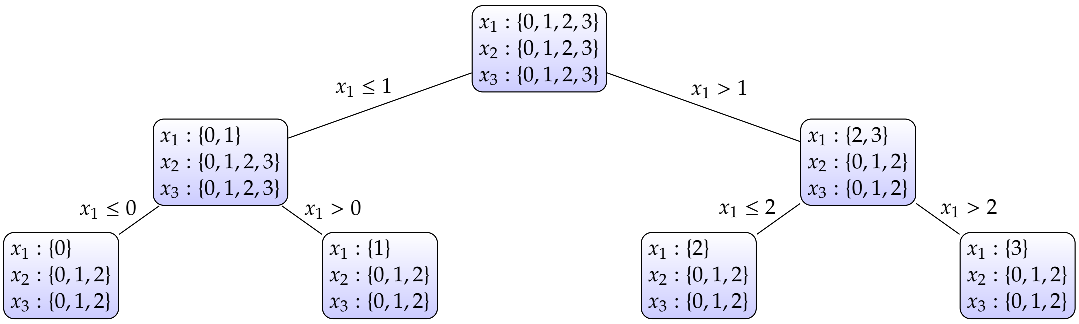


Abbildung 2.6: Ein Ausschnitt eines Suchbaumes, der durch Domain Splitting entsteht.

Backtracking-Varianten

Neben unterschiedlichen Varianten der Suchraumzerlegung existieren auch Alternativen zur Art und Weise, wie das Backtracking während der Suche ausgeführt wird. Bisher wurde der klassische, naive Backtracking-Ansatz vorgestellt, bei dem jede Zuweisung einzeln widerrufen wird.

Alternative Ansätze sind das graphbasierte Zurückspringen (Graph Based Backjumping, [55]) oder das konfliktgerichtete Zurückspringen (Conflict-Directed Backjumping, [128]). Beide Verfahren ermöglichen ein Zurückspringen über mehr als eine Suchbaumebene, widerrufen also mehr als nur die letzte Zuweisung.

Das schnellere Zurückspringen wird in beiden Fällen durch aufwendigere Berechnungen der Rücksprungposition ermöglicht. Zum Beispiel müssen für das Conflict-Directed Backjumping-Verfahren sogenannte Rücksprungmengen (Jumpback Sets) ermittelt werden. Alle Verfahren haben Vor- und Nachteile. In der weiteren Arbeit wird allerdings das naive Backtracking verwendet, wie es zum Beispiel im Choco-Solver zum Einsatz kommt.

Parallelisierung

Die Parallelisierung von Constraint-Problemen ist ein vielversprechender Ansatz, um das Lösen von CSPs zu beschleunigen. Populäre Ansätze der Parallelisierung sind dabei die parallele Suche (parallel search, [133]), Portfolio-Ansätze, Problemzerlegung, Suchraumzerlegung (alle drei [131]), die parallele Konsistenzberechnung (parallel consistency, [66, 114, 134]), die kombinierte parallele Suche und parallele Konsistenzberechnung (combining parallel search and parallel consistency, [134]) sowie verteilte CSPs (distributed CSPs, [54, 162]).

Mit manchen Ansätzen ist dabei eine superlineare Verbesserung der Lösungsgeschwindigkeit (Speedup) möglich (z.B. bei der Suchraumzerlegung), bei anderen Ansätzen, wie z.B. der parallelen Konsistenzberechnung, nicht. Die Parallelisierung von CSPs ist aufgrund der hohen Interaktion der einzelnen Komponenten innerhalb eines CSPs sehr schwierig. Da die Entwicklung von Computern allerdings dahin geht, immer mehr Prozessoren zu nutzen, wird dieses Forschungsfeld auch in Zukunft weiter an Bedeutung gewinnen. An dieser Stelle wird auf die einzelnen Verfahren nicht weiter eingegangen, statt dessen wird auf die angegebene Fachliteratur verwiesen [63, 132].

Weitere Lösungsverfahren

Neben den bisher vorgestellten verschiedenen Varianten der globalen Suche besteht auch die Möglichkeit, Constraint-Probleme mittels *lokaler Suche* [153], *Agenten-basierten Ansätzen* [135] oder dynamischer Programmierung (Dynamic Programming) [31] zu lösen.

Die lokale Suche lässt in der Regel keine Traversierung des gesamten Lösungsraumes zu. Sie garantiert damit also nicht eine global optimale Lösung oder überhaupt eine gültige Lösung zu finden, falls eine solche existiert. Sie ist somit nicht-vollständig. Nichts desto trotz kann sie für eine Vielzahl von Problemen schnell eine Lösung finden, die „hinreichend" gut ist.

Bei der lokalen Suche werden in der Regel zunächst alle Variablen $x_i \in X$ mit einem Wert ihrer Domäne $d_j \in D_i$ belegt und anschließend ein Fehlermaß berechnet, das angibt, in welchem Maße Constraints durch diese Variablenbelegung verletzt sind. Das Maß der Verletzung wird durch eine Zielfunktion $f : D_1 \times ... \times D_n \rightarrow \mathbb{R}$, welche Wissen über alle Constraints C hat, ermittelt. Während der Suche wird die Variablenbelegung schrittweise verändert und das Maß der Verletzung auf der Suche nach einem minimalen Zielwert für f neu berechnet. Abhängig vom konkreten Algorithmus werden dabei nur bessere oder aber auch vorübergehend schlechtere Zielwerte für f akzeptiert.

Lokale Suche kann unter anderem in nachbarschaftsbasierte Ansätze wie *Hill-Climbing*, *Tabu-Suche* oder *Simulated Annealing* und evolutions- oder populationsbasierte Ansätze wie *Genetische Algorithmen* oder den *Ameisenalgorithmus* unterschieden werden [58, 110, 144, 160].

2.4.2 Konsistenz und Propagation

Das im vorherigen Abschnitt gezeigte Lösen von Constraint-Problemen mittels naiver, Backtracking-basierter Suche ist sehr aufwendig, da alle möglichen Belegungen vollständig erzeugt und erst danach auf ihre Gültigkeit überprüft werden. Um den Lösungsvorgang zu beschleunigen, werden deshalb in der Regel Konsistenztechniken und Propagationen verwendet.

Basierend auf dem Prinzip, dass ein CSP $P = (X, D, C)$ unerfüllbar ist, wenn eine Domäne D_i einer Variablen x_i leer ist, wird versucht das ursprüngliche CSP P in ein äquivalentes $P' = (X, D', C)$ umzuformen, bei dem die Domänen für die Variablen verkleinert werden ($D'_i \subseteq D_i \ \forall \ i \in \{1, ..., |D|\}$), indem immer solche Werte entfernt werden, die in keiner Lösung vorkommen können. Beinhaltet durch die Umformung des CSPs eine Domäne einer Variable keinen Wert mehr, so ist das ursprüngliche CSP unerfüllbar. Umgekehrt kann allerdings nicht direkt davon ausgegangen werden, dass eine Domäne leer wird, wenn das ursprüngliche CSP unerfüllbar ist.

Wird mittels lokaler Konsistenztechniken keine Unerfüllbarkeit festgestellt, so wird der im Allgemeinen sehr große Suchraum des CSPs zumindest eingeschränkt. Meist wird der Suchraum auf diese Weise so stark verkleinert, dass danach globale Verfahren, wie z. B. Suche mit Backtracking, die verbleibenden Lösungen wesentlich schneller ermitteln können. In der Praxis werden lokale Konsistenztechniken und globale Suche miteinander verzahnt verwendet. Das heißt, nach jedem Suchschritt werden lokale Konsistenztechniken zur Verringerung des Suchraumes angewendet, bevor der nächste Suchschritt durchgeführt wird. Auf diese Weise entstehen effiziente vollständige Solver (vgl. Abschnitt 2.4.3).

Im Folgenden werden verschiedene Konsistenzniveaus vorgestellt. Die ersten beiden Konsistenzniveaus (*Knotenkonsistenz* und *Kantenkonsistenz*) basieren auf der Graphrepräsentation von binären CSPs (CSPs, die nur Constraints beinhalten, deren Relation maximal zwei Variablen einschränkt), sind aber auch für CSPs anwendbar bzw. adaptierbar, deren Constraints mehr als zwei Variablen umfassen.

Knotenkonsistenz

Die Knotenkonsistenz bezieht sich in erster Linie auf alle unären Constraints eines CSPs. Die anderen Constraints, also die, deren Relation mehr als eine Variable umfasst, gelten per Definition als knotenkonsistent. Ein unäres Constraint ist dann knotenkonsistent, wenn alle Domänenwerte der Variablen des Constraints auch gültige Zuweisungen für dieses Constraint sind. Der Begriff der Knotenkonsistenz ist analog auf ein CSP anwendbar. Ein CSP ist genau dann knotenkonsistent, wenn alle seine Constraints knotenkonsistent sind.

Definition 2.16: *Knotenkonsistenz*. Ein Constraint c ist knotenkonsistent, wenn entweder $|scope(c)| \neq 1$ oder für jeden Domänenwert $d_i \in D(x)$ der Variablen x, mit $scope(c) = \{x\}$ gilt, dass die Zuweisung des Wertes d_i an Variable x das Constraints c erfüllt. Ein CSP mit Constraints $C = \{c_1, ..., c_m\}$ ist knotenkonsistent, wenn alle Constraints $c_i \ \forall \ i \in \{1, ..., m\}$ knotenkonsistent sind [107].

Beim Herstellen der Knotenkonsistenz für ein CSP $P = (X, D, C)$ werden demzufolge alle Werte d_j aus den Domänen D_i entfernt, deren zugehörige Variable x_i mit der Zuweisung d_j ein unäres Constraint $c = (\{x_i\}, R) \in C$ verletzt.

Kantenkonsistenz

Analog zur Knotenkonsistenz bezieht sich die Kantenkonsistenz in erster Linie auf alle binären Constraints eines CSPs. Die anderen Constraints, also die, deren Relation genau eine oder mehr als zwei Variablen umfasst, gelten per Definition als kantenkonsistent. Ein binäres Constraint $c = (\{x, y\}, R)$ ist genau dann kantenkonsistent, wenn es zu jedem möglichen Domänenwert $d_x \in D_x$ der Variablen x mindestens ein Domänenwert $d_y \in D_y$ der Variablen y gibt, so dass eine Belegung ϕ mit $\phi(x) = d_x$ und $\phi(y) = d_y$ das Constraint c erfüllt und umgekehrt.

Definition 2.17: ***Kantenkonsistenz.*** Für den Fall, dass ein Constraint c mehr oder weniger als zwei Variablen umfasst ($|scope(c)| \neq 2$), so ist dieses kantenkonsistent. Ein Constraint c mit $scope(c) = \{x, y\}$ ist kantenkonsistent, wenn für jeden Domänenwert $d_x \in D(x)$ der Variablen x gilt, dass ein Wert $d_y \in D(y)$ der Variablen y existiert, so dass mindestens eine Belegung ϕ_1 mit $\phi_1(x) = d_x$ und $\phi_1(y) = d_y$, als auch für jeden Domänenwert $d_y \in D(y)$ ein Wert $d_x \in D(x)$ existiert, so dass mindestens eine Belegung ϕ_2 mit $\phi_2(x) = d_x$ und $\phi_2(y) = d_y$ existiert, die das Constraint c erfüllt. Ein CSP mit Constraints $C = \{c_1, ..., c_m\}$ ist kantenkonsistent, wenn alle Constraints $c_i \; \forall \; i \in \{1, ..., m\}$ kantenkonsistent sind [107].

Beim Herstellen der Kantenkonsistenz für ein CSP $P = (X, D, C)$ wird für jedes binäre Constraint $c = (\{x, y\}, R)$ untersucht, ob alle Werte in D_x mindestens einen zugehörigen Wert in D_y haben, sodass das Constraint c erfüllt ist. Andernfalls wird der entsprechende Wert aus der Domäne von x entfernt. Analog wird für alle Werte in D_y vorgegangen.

Lokale Konsistenz

Eine Erweiterung der Kantenkonsistenz für Constraints mit mehr als zwei Variablen ist die so genannte generalisierte Kantenkonsistenz (generalized arc-consistency, GAC) oder auch Hyperkanten Konsistenz (hyper-arc consistency) oder auch lokale Konsistenz.

Definition 2.18: ***Lokale Konsistenz.*** Ein Constraint $c = (X, R)$, mit $X = \{x_1, ..., x_n\}$ ist genau dann lokal konsistent, wenn für jeden Wert $a \in D_i$ jeder Variablen x_i mit $i \in \{1, ..., n\}$ ein Tupel $t \in R$ existiert, so dass $t[i] = a$ gilt. Das Tupel t wird auch *support* von a genannt. Ein CSP $P = (X, D, C)$ ist genau dann lokal konsistent, wenn alle Constraints $c \in C$ lokal konsistent sind [44].

Beim Herstellen der lokalen Konsistenz für ein CSP $P = (X, D, C)$ wird für jedes Constraint $c \in C$ überprüft, ob jeder Wert $a \in D_i$ jeder Variablen $x_i \in scope(c)$ in mindestens einem *support* t von c mit $t[i] = a$ vorkommt. Ist dies für mindestens ein Constraint nicht der Fall, so wird der Wert a aus der jeweiligen Domäne D_i entfernt. Vereinfacht ausgedrückt bleiben keine Werte in den Domänen der Variablen zurück, die eine Relation in einem Constraint verletzen. In [44] ist ein Algorithmus zur Herstellung von lokaler Konsistenz aufgeführt.

Globale Konsistenz

Lokale Konsistenz garantiert für ein CSP $P = (X, D, C)$, dass für alle Werte $a \in D_i$ aller Variablen $x_i \in X$ für jedes Constraint $c \in C$ mit $x \in scope(c)$ mindestens eine Belegung ϕ mit $\phi(x_i) = a$ existiert, die c erfüllt. Das garantiert allerdings nicht, dass jeder Domänenwert a jeder Variablen $x_i \in X$ auch Teil einer Lösung des CSPs ist. Wird zum Beispiel das CSP 1 aus Abschnitt 2.1.3 betrachtet, so kann festgestellt werden, dass das CSP lokal konsistent ist, es aber über keine Lösung verfügt.

Aus diesem Grund wird das höhere Konsistenzniveau der globalen Konsistenz eingeführt. Ein CSP $P = (X, D, C)$ ist global konsistent, wenn jede Variable $x_i \in X$ nur noch Werte a_j in ihrer Domäne D_i hat, so dass mindestens eine Variablenbelegung ϕ mit $\phi(x_i) = (a_j)$ existiert, die alle Constraints C erfüllt.

Definition 2.19: ***Globale Konsistenz.*** Gegeben sei ein CSP $P = (X, D, C)$. Sei ϕ eine Belegung mit $\phi(x_{i_1}) = d_{i_1}, ..., \phi(x_{i_{k-1}}) = d_{i_{k-1}}$ mit $d_{i_j} \in D_{i_j}, i \in \{1, ..., n\}, j \in \{1, ..., k\}$, die alle Constraints zwischen den Variablen $x_{i_1}, ..., x_{i_{k-1}}$ erfüllt. Das CSP ist k-konsistent, wenn durch Hinzunahme einer weiteren Variable $x_{i_k}, k \in \{1, ..., n\}$ eine Zuordnung $\phi(x_{i_k}) = d_{i_k}$ mit $d_{i_k} \in D_{i_k}$ existiert, so dass ϕ alle Constraints zwischen den k-Variablen erfüllt.
Ein CSP $P = (X, D, C)$ mit $X = \{x_1, ..., x_n\}$ ist genau dann global konsistent, wenn es k-konsistent für alle k in $\{1, ..., n\}$ ist [44].

Zur Herstellung globaler Konsistenz eines CSPs $P = (X, D, C)$ werden aus allen Domänen $D_i \in D$ diejenigen Werte $a_j \in D_i$ entfernt, bei denen die Zuweisung $\phi(x_i) = a_j$ nicht Teil mindestens einer Lösung des CSPs P ist. In [44] sind zwei Algorithmen zur Herstellung der globalen Konsistenz aufgeführt.

Grenzenkonsistenz

Die globale Konsistenz scheint immer erstrebenswert, da durch sie die Anzahl der Backtracking-Schritte in der darauffolgenden Suche stark verringert werden kann. Allerdings ist die Herstellung globaler Konsistenz in vielen Fällen zu zeit- und rechenaufwändig, weswegen ein weiteres Konsistenzniveau, die Grenzenkonsistenz, vorgestellt wird, das weniger aussagekräftig aber schneller berechenbar als die globale und auch die lokale Konsistenz ist.

Die Grenzenkonsistenz garantiert nicht für jeden Domänenwert $a \in D_i$ einer Variablen x_i, dass die Zuweisung $\phi(x) = a$ Teil einer gültigen Belegung für jedes Constraint $c \in C$ mit $x \in scope(c)$ ist, sondern konzentriert sich lediglich auf den kleinsten $min(D_i)$ und größten $max(D_i)$ Domänenwert der Variablen x_i. Dadurch kann im Vergleich zur lokalen Konsistenz oftmals viel Zeit bei der Konsistenzherstellung eingespart werden.

Definition 2.20: ***Grenzenkonsistenz.*** Eine Variable x ist grenzenkonsistent relativ zu einem Constraint $c = (X, R)$ mit $x \in scope(c)$, wenn der Wert $min(D_x)$ (bzw. $max(D_x)$) zu einem vollständigen Tupel $t \in R$ erweitert werden kann. Ein Constraint c ist grenzenkonsistent, wenn für alle Variablen $x \in scope(c)$ gilt, dass diese grenzenkonsistent relativ zu c sind. Ein CSP ist grenzenkonsistent, wenn all seine Constraints grenzenkonsistent sind [44].

Beim Herstellen der Grenzenkonsistenz für ein CSP $P = (X, D, C)$ wird für jedes Constraint $c \in C$ überprüft, ob der minimale Wert $a_{min} = min(D_i)$ und der maximale Wert $a_{max} = max(D_i)$ jeder Variablen $x_i \in scope(c)$ in jeweils mindestens einem *support* t_1 und t_2 von c mit $t_1[i] = a_{min}$ und $t_2[i] = a_{max}$ auftritt. Ist dies für ein Constraint nicht der Fall, so muss der kleinste Wert $a_{min'} > a_{min}$ (bzw. der größte Wert $a_{max'} < a_{max}$) für die Variable x_i ermittelt werden, sodass ein solcher support t_1 bzw. t_2 mit $t_1[i] = a_{min'}$ bzw. $t_2[i] = a_{max'}$ existiert.

Propagation

Für die zuvor aufgeführten Konsistenzen existiert jeweils eine Version, die sich auf ein Constraint bezieht, und eine, die sich auf ein CSP bezieht. In der Theorie und in speziellen Solvern kann für ganze CSPs jeweils lokale, globale oder Grenzenkonsistenz hergestellt werden, in den meisten Solvern wird aber für jedes Constraint ein separates Konsistenzlevel berücksichtigt (zum Beispiel im Choco- und im Gecode-Solver). Das Vorgehen zum Herstellen eines bestimmten Konsistenzniveaus für ein Constraint wird *Propagation* genannt und kann je nach Constraint sehr unterschiedlich ausfallen. Die Propagation wird von einem sogenannten *Propagator* ausgeführt.

Es besteht die Möglichkeit, dass ein Constraint abhängig von seinen Eingaben verschiedene oder aber auch mehrere Propagatoren, die gegebenenfalls ein unterschiedliches Konsistenzniveau anstreben, zur Verfügung hat. Ein Beispiel für solch ein Constraint, das auf mehrere Propagatoren mit unterschiedlichem Konsistenzziel zugreifen kann, ist das *AllDifferent*-Constraint. Dieses verfügt oftmals sowohl über einen Propagator, der Grenzenkonsistenz, als auch über einen, der lokale Konsistenz herstellt. Ein Vertreter für Constraints, die in Abhängigkeit von ihren Eingaben andere Propagatoren verwenden, ist das Summen-Constraint (*Sum*-Constraint). Für boolesche oder nicht-boolesche Variablen bzw. für weniger als vier Variablen oder mehr als drei Variablen werden im Choco-Solver zum Beispiel unterschiedliche Propagatoren verwendet, welche für die jeweiligen Eingaben optimiert sind.

Auf die beiden genannten Constraints (*AllDifferent*-Constraint und *Sum*-Constraint) wird im Abschnitt 2.5 genauer eingegangen. An dieser Stelle soll exemplarisch die Propagation von einfachen arithmetischen Constraints beschrieben werden.

Zunächst werden Constraints der Form $x\mathfrak{R}y$ mit $\mathfrak{R} \in \{<, \leq, =, \geq, >, \neq\}$ und $x, y \in X$ sind Variablen eines FD-CSP $P = (X, D, C)$ mit Domänen $D_x \in D$ und $D_y \in D$ betrachtet. In Algorithmus 1 ist ein Propagationsalgorithmus für die Kleiner-Relation angegeben, der zunächst alle Werte aus der Domäne D_y entfernt, die der Variable y nicht zugeordnet werden können, da die Variable x andernfalls keinen kleineren Wert mehr in ihrer Domäne enthält (Zeilen 1 und 2). Im zweiten Schritt werden alle Werte aus der Domäne D_x entfernt, die der Variable x nicht zugeordnet werden können, da die Variable y andernfalls über keinen größeren Wert mehr in ihrer Domäne verfügt (Zeilen 3 und 4).

Algorithmus 1: Ein Propagationsalgorithmus für $x < y$

```
  Input: Variable $x$ mit Domäne $D_x$, Variable $y$ mit Domäne $D_y$
1 forall ($d \in D_y \mid d \leq min(D_x)$) do
2     remove($d, D_y$)
3 forall ($e \in D_x \mid e \geq max(D_y)$) do
4     remove($e, D_x$)
```

Für die Realisierung der Relation ≤ kann derselbe Propagator verwendet werden, statt den Variablen x und y müsste der Algorithmus lediglich x und $(y + 1)$ als Eingabe erhalten. Bei den Relationen > und ≥ können die Variablen getauscht werden und somit wieder der Propagator für die Relation < bzw. ≤ verwendet werden. Bei der Relation = könnte der Propagator zum Beispiel zweimal mit den Eingaben $x < y + 1$ und $y < x + 1$ aufgerufen werden. Lediglich zur Realisierung der Relation ≠ muss ein anderer Propagator verwendet werden, der nur dann aufgerufen werden muss, wenn eine der beiden Variablen nur noch einen Wert d beinhalten würde ($D_x = \{e\}$ oder $D_y = \{d\}$). Sei x diese Variable. Es genügt dann, den Wert e aus der Domäne von y zu entfernen.

2.4.3 Die Kombination von Suche und Propagation

Wie bereits in den vorherigen Abschnitten erwähnt wurde, wird in vielen Solvern die Suche verzahnt mit der Propagation verwendet. Statt den gesamten Suchraum direkt mit Backtracking-basierter Suche zu erkunden, werden zunächst alle Propagatoren einmal angewendet und somit der Suchraum gegebenenfalls initial verkleinert. Führt eine Propagation dazu, dass mindestens ein Wert aus der Domäne einer Variablen x entfernt wird, so müssen alle Propagatoren nochmals aufgerufen werden, deren zugehörige Constraints ebenfalls diese Variable beinhalten.

Durch dieses Vorgehen können Propagationszyklen entstehen. Da Propagatoren Domänen allerdings niemals vergrößern und weitere Propagationen nur nach sich ziehen, wenn sie eine Domäne verkleinern, während gleichzeitig die Anzahl der Variablen und die Größen der zugehörigen Domänen endlich sind, terminiert das Verfahren nach endlich vielen Schritten. Die Reihenfolge, in der Propagatoren ausgewertet werden, hat einen Einfluss auf die Lösungsgeschwindigkeit eines CSPs. Sie beeinflusst nicht das Aussehen des Suchbaumes, allerdings kann eine ungünstige Reihenfolge dazu führen, dass Propagatoren unnötig oft ausgeführt werden müssen. Die Wahl der richtigen Propagationsreihenfolge ist, wie die Auswahl der richtigen Suchstrategie, problemabhängig und heuristisch. Es gibt verschiedene Heuristiken, die im Durchschnitt eine gute Performance liefern. Gängige Constraint-Solver (wie zum Beispiel GECODE, JACOP, die GOOGLE OR-TOOLS oder der CHOCO-Solver) ordnen Propagatoren häufig nach der Komplexität, die die Berechnung des Propagationsschrittes hat.

Verändert sich keine Domäne mehr und sind alle Propagationen abgearbeitet, so kann auf dem verringerten CSP die Suche gestartet werden. Dabei wird lediglich einer Variablen x ein Wert zugewiesen, bevor wieder alle Propagatoren kaskadierend aufgerufen werden, deren zugehörige Constraints die Variable x enthalten. Dieses Vorgehen wiederholt sich, bis eine Lösung gefunden oder eine Domäne vollständig geleert wurde. Im letzteren Fall wird, wie bei der naiven Suche, Backtracking verwendet, um in andere, noch nicht-besuchte Bereiche des Lösungsraumes zu gelangen.

Anhand des CSPs 5 (vgl. S. 40) wird das Prinzip und die Effektivität dieses Vorgehens erläutert. In Abbildung 2.7 ist der vollständige Suchbaum für das CSP 5 bei Verwendung von Suche in Kombination mit Propagation dargestellt. Der Startknoten ist identisch mit dem Startknoten in Abbildung 2.4 und beinhaltet das ursprüngliche CSP P, welches im Folgenden mit P_1 bezeichnet wird. Im Gegensatz zu der Suche, die in Abbildung 2.4 dargestellt ist, müssen in diesem Fall zunächst alle Propagatoren p_1, p_2 und p_3 für die Constraints $C = \{c_1 = (x_1 \neq x_2), c_2 = (x_1 < x_3), c_3 = (x_2 \neq x_3)\}$ propagiert werden, bevor die erste Variablenzuweisung durchgeführt wird. Die Reihenfolge, in der die Propagatoren ausgewertet werden, ergibt sich in diesem Fall aus ihrer Indexreihenfolge.

Die Anwendung des Propagators p_1 hat keine Eliminierung von Domänenwerten zur Folge. Die Propagation von p_2 schließt hingegen den Wert 0 aus der Domäne D_3 aus, da die Bedingung $(d \in D_1 \mid d \leq min(D_3)) = (0 \leq 0)$ (Zeile 1 in Algorithmus 1) für diesen Wert erfüllt ist. Da die Domäne von Variable x_3 verringert wurde, müssen in der Folge alle Propagatoren p_i erneut aufgerufen werden, deren Constraints c_i die Variable x_3 enthalten. In diesem Fall umfasst das lediglich p_3, wobei es allerdings zu keiner weiteren Einschränkung von Domänenwerten kommt. Somit wurde ein lokal konsistentes CSP P'_1 erzeugt.

Für das CSP P'_1 wird entsprechend einer Suchheuristik eine Variable ausgewählt (hier x_1 analog zum Vorgehen in Abbildung 2.4) und diese mit einem Wert ihrer Domäne instantiiert ($x_1 = 0$). Auf dem so entstandenen CSP P_2 müssen daraufhin wieder alle Propagatoren p_i aufgerufen werden, deren zugehörige Constraints c_i die Variable x_1 enthalten. In diesem Fall sind das die Propagatoren p_1 und p_2. Die Propagation von p_1 hat zur Folge, dass x_2 der Wert 1 zugeordnet wird und eine anschließende Propagation von p_2 führt zu der Zuweisung des Wertes 2 zur Variable x_3. Damit sind alle Variablen instantiiert. Um sicherzugehen, dass es sich bei der Belegung ϕ mit $\phi(x_1) = 0$, $\phi(x_2) = 1$

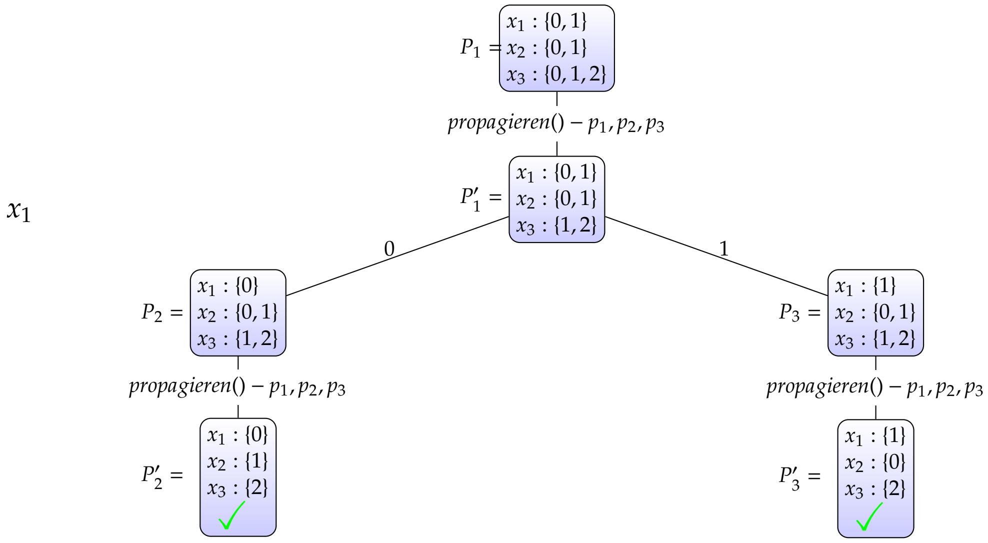


Abbildung 2.7: Ein vollständiger Suchbaum für das CSP 5 bei der Verwendung von Suche in Kombination mit Propagation.

und $\phi(x_3) = 2$ um eine Lösung handelt, muss allerdings noch überprüft werden, ob ein Propagator, der aufgrund der letzten Domänenänderungen noch ausgeführt werden muss, dafür sorgt, dass eine Domäne leer wird. In diesem Beispiel muss daher noch p_3 ausgewertet werden. Da dies aber zu keiner leeren Domäne führt, stellt die Belegung ϕ eine Lösung des CSPs dar.

Für die Suche nach weiteren Lösungen werden mittels Backtracking alle Domänenänderungen seit dem Erzeugen von CSP P_1' widerrufen und der Variablen x_1 wird ein anderer Wert ihrer Domäne zugeordnet. Anschließend wird analog verfahren.

Abbildung 2.7 zeigt, dass die Anzahl an Knoten, die durchlaufen werden müssen, um alle Lösungen zu finden, im Vergleich zu der naiven Backtracking-basierten Suche, die in Abbildung 2.4 dargestellt ist, signifikant reduziert werden konnte. Insbesondere konnte es verhindert werden, Knoten zu expandieren, die keine Lösung beinhalten. Backtracking ist in diesem konkreten Beispiel nur notwendig, um nach dem Finden einer Lösung nach weiteren Lösungen zu suchen. Auf der anderen Seite wird dafür mehr Rechenzeit benötigt um die Propagationen durchzuführen. Für die meisten Probleme ist die zweite Herangehensweise, in der die Suche mit der Propagation interagiert, die deutlich effektivere, weswegen im weiteren Verlauf von dieser ausgegangen wird. Gängige Solver wie Gecode, JaCoP, die Google OR-Tools oder der Choco-Solver (vgl. Abschnitt 2.3.2) verwenden ebenfalls diese Herangehensweise.

2.5 Globale Constraints

Nach [70] beschreiben *Globale Constraints* komplexe Bedingungen, an deren Stelle sonst eine Vielzahl einfacherer Constraints stünden. Die zwei wesentlichen Vorteile von globalen Constraints gegenüber der Kombination mehrerer primitiver Constraints sind auf der einen Seite die besser zugeschnittenen Propagationsalgorithmen, die in den meisten Fällen ein schnelleres Propagieren und genaueres Ausschließen von Domänenwerten erlauben und auf der anderen Seite die Möglichkeit, komplexe Sachverhalte mit wenigen, kompakten und verständlichen Constraints zu modellieren.

Eine besondere Eigenschaft globaler Constraints ist die, dass die Anzahl der Variablen globaler Constraints nicht von vornherein auf einen bestimmten Wert begrenzt ist. Während zum Beispiel der in Algorithmus 1 vorgestellte Propagator für das Constraint, das die $<$ Relation repräsentiert, auf genau zwei Variablen beschränkt ist, erlauben globale Constraints die Eingabe von $n \in \mathbb{N}$ vielen Variablen. Typische Vertreter von globalen Constraints sind das *AllDifferent-*, das *Sum-*, das *Scalar-*, das *Global Cardinality-*, das *Count-*, das *Regular-* und das *Table*-Constraint, welche in den weiteren Abschnitten beschrieben werden. Es wird für alle im Folgenden vorgestellten Constraints angenommen, dass ein FD-CSP mit $P = (X, D, C)$ gegeben ist.

Weitere globale Constraints und Beschreibungen dieser können dem Global Constraint Katalog [1] entnommen werden.

Das *AllDifferent*-Constraint

Das *AllDifferent*-Constraint ist eines der am stärksten erforschten Constraints [101, 154]. Es garantiert für eine Liste von Variablen $\{x_1, ..., x_n\} = X' \in X$, dass diese paarweise verschiedene Domänenwerte $\{d_1, ..., d_n\}$ zugeordnet bekommen.

$$AllDifferent(\{x_1, ..., x_n\}) := \{(d_1, ..., d_n) \mid d_i \neq d_j \forall i, j \in \{1, ..., n\}, i < j\} \tag{2.2}$$

Das *AllDifferent*-Constraint unterstreicht die beiden Hauptvorteile globaler Constraints. Auf der einen Seite vereinfacht es die Modellierung, es wird nur ein *AllDifferent*-Constraint anstelle von $n*(n-1)/2$ vielen Ungleichheit-Constraints benötigt. In Gleichung 2.3 ist die Repräsentation des *AllDifferent*-Constraints durch Ungleichheit-Constraints aufgeführt. Für ein *AllDifferent*-Constraint mit zehn Variablen wären bereits $\frac{9*10}{2} = 45$ viele primitive Ungleichheit-Constraints nötig. Auf der anderen Seite erlaubt das *AllDifferent*-Constraint den Einsatz effektiverer Propagationsalgorithmen, die ein höheres Konsistenzniveau erreichen (lokale Konsistenz). Auf die Propagationsalgorithmen des *AllDifferent*-Constraints soll an dieser Stelle nicht weiter eingegangen werden, statt dessen wird auf [101] verwiesen.

$$AllDifferent(\{x_1, ..., x_n\}) \Leftrightarrow (x_i \neq x_j)\ \forall x_i, x_j \in \{x_1, ..., x_n\}, i < j \tag{2.3}$$

Das *AllDifferent*-Constraint wird bei der Modellierung vieler Probleme wie zum Beispiel der Ressourcen-, Personal- oder Routenplanung genutzt [154]. Es eignet sich insbesondere zur Darstellung von Permutationen und von Zyklen in Graphen. Eine ganze Reihe weiterer globaler Constraints wie zum Beispiel das *Count*- oder das *Global Cardinality*-Constraint stellen eine Erweiterung des *AllDifferent*-Constraints dar [70].

Zur Veranschaulichung des Nutzens globaler Constraints und insbesondere des *AllDifferent*-Constraints wird in Beispiel 2.4 das bereits (aus Beispiel 2.2 und 2.3) bekannte n-Damen-Problem so angepasst, dass anstelle von vielen Ungleichheits-Constraints jeweils *AllDifferent*-Constraints verwendet werden.

Beispiel 2.4: Das (globale, FD-) n-Damen-Problem. *Es gilt die gleiche Problemstellung wie in den Beispielen 2.2 und 2.3. Anstelle der Ungleichheits-Constraints zur Beschreibung, dass keine zwei Damen in einer Reihe auftreten dürfen, wird nur ein* AllDifferent-*Constraint verwendet.*

Die Bedingung, die ausdrückt, dass sich nur maximal eine Dame pro Diagonale wiederfindet, kann mit zwei AllDifferent-*Constraints dargestellt werden. Für die Diagonalen, die auf dem Schachbrett von links unten nach rechts oben verlaufen, genügt es zu fordern, dass alle Werte der Variablen mit ihrem jeweiligen Indexwert verringert um eins addiert ($x_i + (i-1)$) untereinander verschieden sein müssen. Für die anderen Diagonalen (von links oben nach rechts unten) muss die Bedingung analog mit einer Subtraktion ausgedrückt werden.*

Das CSP 6 zeigt eine Modellierung des n-Damen-Problems mit Hilfe des globalen AllDifferent-*Constraints. Es ist zu erkennen, dass unabhängig von der Größe des Problems nur noch drei Constraints benötigt werden. Im Vergleich zu den Varianten aus den Beispielen 2.2 und 2.3 führt die verringerte Constraint-Anzahl zu einer größeren Übersichtlichkeit und bei großen Probleminstanzen zu einer schnelleren Lösungsfindung, da die mächtigeren* AllDifferent-*Propagatoren anstelle der schwächeren Ungleichheits-Propagatoren verwendet und somit viele Suchschritte vermieden werden können.*

CSP 6: $P = (X, D, C)$ mit:

$X = \{x_1, ..., x_n\}$ (n Variablen)

$D = \{D_1, ..., D_n \mid D_1 = ... = D_n = \{1, ..., n\}\}$ (n Domänen)

$C = \{\mathit{AllDifferent}(\{x_1, ..., x_n\}),$ (genau eine Dame pro Reihe)

$\mathit{AllDifferent}(\{x_1, x_2 + 1, ..., x_n + n - 1\}),$ (genau eine Dame pro Diagonale)

$\mathit{AllDifferent}(\{x_1, x_2 - 1, ..., x_n - (n-1)\})\}$ (genau eine Dame pro Diagonale)

Das *Count*-Constraint

Das *Count*-Constraint ist eine Abwandlung des *AllDifferent*-Constraints. Für eine geordnete Menge von Variablen $\{x_1, ..., x_n\} = X' \subseteq X$ und eine weitere Variable $occ \in X$ mit dazugehöriger Domäne $D_{occ} = \{occ_{min}, ..., occ_{max}\}$ wird gefordert, dass die Anzahl an Vorkommen des Wertes $v \in \mathbb{N}$ in einer Belegung der Variablen-Menge X' dem Wert der Variablen occ entspricht:

$$Count(X', occ, v) \Leftrightarrow (\sum_{x \in X'} \begin{cases} 0 & x \neq v \\ 1 & x = v \end{cases}) = occ \tag{2.4}$$

Beispielsweise ist das Constraint $Count(\{x_1, x_2, x_3\}, 2, 1)$ für die Belegung ϕ mit $\phi(x_1) = 1$, $\phi(x_2) = 3$ und $\phi(x_3) = 1$ erfüllt, weil der Wert $v = 1$ zweimal ($occ = 2$) in x_1 bis x_3 auftritt.

Das *Global Cardinality*-Constraint

Das *Global Cardinality*-Constraint (gcc) ist eine weitere Abwandlung des *AllDifferent*-Constraints und gleichzeitig eine Erweiterung des *Count*-Constraints für mehrere Domänenwerte. Genauer beschreibt das *Global Cardinality*-Constraint für eine geordnete Menge von Variablen $\{x_1, ..., x_n\} = X' \subseteq X$ und eine geordnete Menge von Zählvariablen $\{c_1, ..., c_m\} \in X$, dass jeder Wert $j \in \{1, ..., m\}$ genau c_j viele Male in einer Belegung der Variablenmenge X' vorkommen muss. Insbesondere lässt sich das *Global Cardinality*-Constraint durch mehrere *Count*-Constraints beschreiben:

$$GCC(X', \{c_1, ..., c_m\}) \Leftrightarrow \bigwedge_{j \in \{1, ..., m\}} count(X', c_j, j) \tag{2.5}$$

Das *Sum*-Constraint

Das *Sum*- oder Summen-Constraint fordert für eine geordnete Menge von Variablen $\{x_1, ..., x_n\} = X' \subseteq X$, ein Relationssymbol $\Re \in \{<, \leq, =, \geq, >, \neq\}$ und eine weitere Variable $x_r \in X$, dass die Summe der Werte, die den Variablen in X' zugeordnet werden, in Relation $\Re$ zu dem Wert von x_r steht:

$$Sum(X', \Re, x_r) \Leftrightarrow (\sum_{x \in X'} x) \; \Re \; x_r \tag{2.6}$$

Das *Scalar*-Constraint

Das *Scalar*-Constraint ist eine Verallgemeinerung des *Sum*-Constraints, bei dem eine Gewichtung durch einen Skalarvektor berücksichtigt wird. Es fordert für eine geordnete Menge von Variablen $\{x_1, ..., x_n\} = X' \subseteq X$, einen Vektor $\vec{c} = (c_1, ..., c_n)^T$ mit $c_i \in \mathbb{Z}$, ein Relationssymbol $\Re \in \{<, \leq, =, \geq, >, \neq\}$ und eine weitere Variable $x_r \in X$, dass das Skalarprodukt der Werte, die den Variablen in X' zugeordnet werden, mit den Skalaren in c in Relation $\Re$ zu dem Wert von x_r steht:

$$Scalar(X', \vec{c}, \Re, x_r) \Leftrightarrow (\sum_{i=1}^{n} x_i * c_i) \; \Re \; x_r \tag{2.7}$$

Das *Table*-Constraint

Das *Table*-Constraint ist ebenfalls eines der am stärksten erforschten und am vielfältigsten genutzten Constraints in der Praxis [34, 88, 89, 91, 93, 105]. Für eine geordnete Menge von Variablen $\{x_1, ..., x_n\}$ mit $x_i \in X, \forall i \in \{1, ..., n\}$ und eine Liste von Tupeln T gibt ein positives (bzw. negatives) *Table*-Constraint an, dass nur Variablenbelegungen ϕ mit $\phi(x_1) = d_1, ..., \phi(x_n) = d_n$ erlaubt (bzw. solche verboten) sind, für die gilt, dass das Tupel $(d_1, ..., d_n) \in T$ ist. Für ein positives *Table*-Constraint gilt demzufolge:

$$Table(\{x_1, ..., x_n\}, T) \Leftrightarrow (d_{j,1}, ..., d_{j,n}) \in T \text{ mit } j \in \{1, ..., |T|\} \tag{2.8}$$

Es existieren viele verschiedene Propagatoren für das *Table*-Constraint, die sowohl in verschiedenen Solvern aber auch innerhalb eines Solvers Anwendung finden. Der Choco-Solver allein verfügt bereits über mehr als zehn unterschiedliche Propagator-Implementierungen für das *Table*-Constraint. Es folgt eine Liste von Akronymen, unter denen die jeweils verwendeten Algorithmen in der Literatur bekannt sind:

- AC2001, AC3, AC3rm, AC3bit+rm, CT, FC, GAC2001, GAC2001+, GAC3rm, GAC3rm+, GACSTR+, MDD+ und STR2+.

In dieser Arbeit wird immer vom *Compact Table* (CT)-Algorithmus ausgegangen, wenn über den *Table*-Propagator gesprochen wird. Die Spezialisierung auf diesen Algorithmus erfolgte, da dieser in vielen Solvern als Standard festgelegt ist (unter anderem im Choco-Solver, in OscaR und in den Google OR-Tools).

Der CT-Algorithmus wurde in [50] eingeführt und wird an dieser Stelle nicht detailliert vorgestellt, stattdessen wird dazu eingeladen, die angegebene Literatur zu lesen.

Das *Regular*-Constraint

Für die Definition des *Regular*-Constraints wird zunächst die Definition eines *deterministischen endlichen Automaten* (engl. *deterministic finite automaton, DFA*) benötigt.

Definition 2.21: ***Deterministischer endlicher Automat.*** Ein deterministischer endlicher Automat ist ein Fünftupel $M = (Q, \Sigma, \delta, q_0, F)$ mit:

- Q ist eine endliche Menge von Zuständen,
- Σ ist ein endliches Eingabealphabet,
- δ ist eine Übergangsfunktion $Q \times \Sigma \rightarrow Q$,
- $q_0 \in Q$ ist ein Startzustand und
- $F \subseteq Q$ ist eine Menge von finalen, akzeptierenden Zuständen.

Ein DFA beginnt im Startzustand q_0 und liest Zeichen für Zeichen $w_i, i \in \{1, ..., n\}$ ein Eingabewort $w = w_1 w_2 ... w_n$ ein. Entsprechend der Übergangsfunktion δ wird vom aktuellen Zustand durch das Einlesen des nächsten Zeichens des Eingabewortes in einen Folgezustand übergegangen. Ein Eingabewort w wird genau dann von einem DFA M akzeptiert, wenn M nach Einlesen von w aus dem Startzustand q_0 in einen akzeptierenden Zustand $q_{accept} \in F$ übergegangen ist und dort zum Halten kommt. Die von M erzeugte Sprache $L(M)$ ergibt sich aus der Menge aller Wörter, die M als Eingabe erhalten kann, um ausgehend vom Startzustand in einem finalen Zustand zum Halten zu kommen[78].

Das *Regular*-Constraint erlaubt für einen DFA $M = (Q, \Sigma, \delta, q_0, F)$ und eine geordnete Menge von Variablen $\{x_1, ..., x_n\} = X' \subseteq X$ mit Domänen $\{D_1, D_2, \ldots, D_n\} = D' \subseteq D$, für die gilt, dass $D_i \subseteq \Sigma$ mit $i \in \{1, \ldots, n\}$ ist, nur solche Variablenbelegungen $\phi \in X' \rightarrow \Sigma$ mit $\phi(x_1) = d_1, ..., \phi(x_n) = d_n$ als gültig, für die das Wort $d_1 d_2 ... d_n$ von dem Automaten M akzeptiert wird:

$$Regular(\{x_1, ..., x_n\}, M) \Leftrightarrow w_1 w_2 \ldots w_n \in L(M) \text{ wobei } w_i \in D_i, \forall i \in \{1, \ldots, n\} \text{ [70].} \tag{2.9}$$

Es wird in dieser Arbeit davon ausgegangen, dass der von Gilles Pesant [124] eingeführte Propagator für das *Regular*-Constraint verwendet wird. Da ein $Regular(\{x_1, ..., x_n\}, M)$-Constraint mit n Variablen nur Wörter der Länge n repräsentieren kann, wird beim Erzeugen des *Regular*-Propagators der eingegebene DFA M transformiert, so dass dieser ebenfalls nur Wörter der Länge n akzeptiert. Solch ein transformierter DFA ist immer azyklisch und seine Zustände Q können in *Ebenen* (engl. *Level*) Q_0 bis Q_n aufgeteilt werden, so dass Übergänge nur von einem Zustand aus Level Q_i in einen Zustand aus Level Q_{i+1} übergehen. In der weiteren Arbeit werden Automaten dieser Art als levelbasierte DFAs bezeichnet.

Definition 2.22: ***levelbasierter deterministischer endlicher Automat, levelbasierter DFA.*** Ein levelbasierter deterministischer endlicher Automat M ist ein DFA (Q, Σ, δ, q_0, F), bei dem die Zustände Q in Level $Q_0, ..., Q_n$ unterteilt sind, für die gilt:

- Q_0 beinhaltet nur den Startzustand q_0
- Q_n beinhaltet nur akzeptierende Zustände
- Übergänge sind nur von einem Zustand aus Level Q_i zu einem Zustand aus Level Q_{i+1} erlaubt.

Ein levelbasierter DFA ist eine Spezialform eines *gerichteten azyklischen Graphen* (*directed acyclic graph*, *DAG*). Hat ein solcher levelbasierter DFA nur einen Endzustand, so wird er in der Literatur häufig als *mehrwertiges Entscheidungsdiagramm* (*Multi-valued decision diagram*, *MDDs*) bezeichnet [12, 123].

Die beschriebene Transformation eines DFAs M kann immer durch Schnittmengenautomatenbildung mit einem Automaten M^{all}, der alle Wörter der Länge n akzeptiert, erzeugt werden. Wird der DFA anschließend minimiert, so verfügt er zwangsläufig über maximal einen akzeptierenden Zustand. Im weiteren Verlauf der Arbeit wird sich im Zusammenhang mit *Regular*-Constraints, wenn nicht anders beschrieben, immer auf solch einen transformierten, minimierten, levelbasierten DFA bezogen.

Es existiert eine Vielzahl weiterer globaler Constraints. Der *Global Constraint Catalog* [1] gibt eine Übersicht über diese, nennt synonyme Bezeichner, unter denen diese bekannt sind und die Solver, in denen diese verwendet werden können. Zur Demonstration, dass sich mit Hilfe globaler Constraints reale Probleme gut modellieren lassen, dient das folgende Beispiel 2.5, in welchem das in der Einleitung vorgestellte Schichtplanungsproblem modelliert wird.

Beispiel 2.5: Das Schichtplanungsproblem. *Sei das Schichtplanungsproblem aus Beispiel 1.1 mit seinen acht aufgelisteten Anforderungen unter Verwendung des dort vorgestellten Rotationsprinzips gegeben. Dementsprechend wird für einen Mitarbeiter und jeden Tag eine Variable mit Domäne 0 = Frei, 1 = Früh, 2 = Spät und 3 = Nacht (Anforderung 2) benötigt. Für die angegebenen acht Wochen mit jeweils sieben Tagen ergibt das* $8 * 7 = 56$ *Variablen* $X = \{x_1, ..., x_{56}\}$ *(Anforderung 1). Die erste Variable repräsentiert dabei sowohl die Schicht am ersten Tag für den ersten Mitarbeiter als auch die erste Schicht in der letzten Woche für den zweiten Mitarbeiter und die erste Schicht in der vorletzten Woche für den dritten Mitarbeiter usw.*

Für einen einfacheren Zugriff auf die Variablen sei X^{2d} *eine* 8×7 *Matrix, die die Variablen von* X *enthält. Jede Zeile steht dabei für einen Mitarbeiter und jede Spalte für einen Wochentag. Der Eintrag in der zweiten Zeile in der dritten Spalte repräsentiert somit unter anderem die Schicht, die der zweite Mitarbeiter in der ersten Woche am Mittwoch zugeordnet bekommt. Die Variablen für* X^{2d} *ergeben sich somit nach folgender Zuweisung:* $X^{2d}_{i,j} = X_{(i-1)*7+j}\ \forall i \in \{1, ..., 8\}, j \in \{1, ..., 7\}$.

Weiterhin werden Constraints für das Erfüllen der Anforderungen 3 bis 8 benötigt. Für die Erfüllung der Personalanforderungen gibt jeder Wert $M_{s,w}$ *der in der Tabelle 1.1 dargestellten Matrix* M *an, wie viele Mitarbeiter für jede Schicht* $s \in \{0, ..., 3\}$ *an jedem Wochentag* $w \in \{1, ..., 7\}$ *benötigt werden. Für jede Schicht* s *und alle Tage* $X_{1,w}, X_{2,w}, ..., X_{8,w}$ *des gleichen Wochentages* w *muss die Anzahl des Auftretens der Schicht* s *in einer Belegung für* $X^{2d}_{1,w}, X^{2d}_{2,w}, ..., X^{2d}_{8,w}$ *gleich dem Wert* $M_{s,w}$ *sein. Dafür wird das Constraint* $count(\{X^{2d}_{1,w}, X^{2d}_{2,w}, ..., X^{2d}_{8,w}\}, s, M_{s,w})$ *verwendet. Dies drückt aus, dass die Anzahl der Vorkommen des Wertes* s *(also einer Schicht) über alle acht Mitarbeiter betrachtet am Wochentag* w *dem Wert* $M_{s,w}$ *entsprechen muss.*

Für die Gleichheit der Schichten am Wochenende (Anforderung 4) können Gleichheits-Constraints für die Variablenpaare $X^{2d}_{j,6}$ *und* $X^{2d}_{j,7}$ *festgelegt werden.*

Für die minimale Anzahl an Schichtwiederholungen an aufeinanderfolgenden Tagen (mindestens zwei, Anforderung 5) wird für jede Variable X_i *mit* $i \in \{1, ..., 56\}$ *gefordert, dass sie gleich zu ihrem Vorgänger oder ihrem Nachfolger ist. Für die maximale Anzahl an Schichtwiederholungen an aufeinanderfolgenden Tagen (maximal vier, Anforderung 6) wird für je fünf aufeinanderfolgende*

Variablen gefordert, dass die fünfte Variable einen anderen Wert zugeordnet bekommen muss, wenn die ersten 4 dieser Variablen den gleichen Wert zugeordnet bekommen($(x_i = x_{mod(i+1)} = x_{mod(i+2)} = x_{mod(i+3)}) \longrightarrow x_i \neq x_{mod(i+4)}$*). Die Funktion mod(k) berechnet dabei (*k *mod* $56) + 1$ *und garantiert somit, dass für Indizes größer 56 oder kleiner 0 ein adäquater Index im Bereich von 1 bis 56 gewählt wird.*

Für das Einhalten der Vorwärtsrotation (Anforderung 7) muss für je zwei Variablen x_i *und* x_{i+1} *immer gelten, dass* x_{i+1} *entweder den gleichen, einen größeren oder den Wert 0 zugeordnet bekommt. Dies lässt sich durch die Verknüpfung von arithmetischen und logischen Ausdrücken realisieren* $(x_i \leq x_{i+1}) \vee (x_{i+1} = 0)$*.*

Für die Anforderung 8, dass innerhalb von zwei Wochen immer mindestens zwei freie Tage liegen müssen, wird ebenfalls das Count-Constraint verwendet. Das FD-CSP 7 beschreibt das gegebene Schichtplanungsproblem und fasst die zuvor zusammengetragenen Punkte noch einmal zusammen. Hinter den Variablen, Domänen und Constraints ist angegeben, welche Anforderung (A1 bis A8) sie erfüllen.

CSP 7: $P = (X, D, C)$ mit:

$X = \{x_1, ..., x_{56}\} \mathrel{\widehat{=}} X^{2d}$ (7 ∗ 8 Variablen, A1)
mit $X^{2d}_{i,j} = X_{(i-1)*7+j}$ für $i \in \{1, ..., 8\}, j \in \{1, ..., 7\}$

$D = \{D_1, ..., D_{56} \mid D_1 = ... = D_{56} = \{0, 1, 2, 3\}\}$ (4 Schichttypen, A2)

$C = \{count(\{X^{2d}_{1,w}, X^{2d}_{2,w}, ..., X^{2d}_{8,w}\}, s, M_{s,w}) \mid \forall w \in \{1, ..., 7\}, \ \forall s \in \{0, 1, 2, 3\}\} \cup$
(Personalanforderungen, A3)

$\{X^{2d}_{j,6} = X^{2d}_{j,7} \mid \forall j \in \{1, ..., 8\}\} \cup$ (Wochenendgleichheit, A4)

$\{(x_{56} = x_{55}) \vee (x_{56} = x_1), \ (x_1 = x_{56}) \vee (x_1 = x_2)\} \cup$
$\{(x_i = x_{i-1}) \vee (x_i = x_{i+1}) \mid \forall i \in \{2, ..., 55\}\} \cup$ (min. 2 Wiederholungen, A5)

$\{(x_i = x_{mod(i+1)} = x_{mod(i+2)} = x_{mod(i+3)}) \longrightarrow x_i \neq x_{mod(i+4)} \mid \forall i \in \{1, ..., 52\}\} \cup$
(max. 4 Wiederholungen, A6)

$\{(x_{56} \leq x_0) \vee (x_1 = 0)\} \cup \{(x_i \leq x_{i+1}) \vee (x_{i+1} = 0) \mid \forall i \in \{1, ..., 55\}\} \cup$
(Vorwärtsrotation, A7)

$\{count(\{x_i, x_{mod(i+1)}, ..., x_{mod(i+13)}\}0, 2) \mid \forall i \in \{1, ..., 56\}\} \cup$ (min. 2 freie Tage, A8)

Das Beispiel 2.5 hat gezeigt, wie ein reales Schichtplanungsproblem als CSP modelliert werden kann. Durch die Verwendung des *Count*-Constraints können die Personalanforderungen und die Forderung nach zwei freien Tagen innerhalb von zwei Wochen sehr kompakt und verständlich dargestellt werden. Die Möglichkeit, Probleme mit vorgefertigten Gerüsten (globalen Constraints) zu beschreiben ohne einen Lösungsweg angeben zu müssen, macht die Constraint-Programmierung in der Wirtschaft zu einem interessanten Modellierungs- und Lösungswerkzeug.

In diesem Kapitel wurden die Grundlagen der Constraint-Programmierung beschrieben. Es wurden essentielle Begriffe wie Constraint, CSP und Variablenbelegung definiert, die in der weiteren Arbeit verwendet werden. Darüber hinaus wurden verschiedene Constraint-Domänen und Solver vorgestellt. Auf den FD-Solver wurde im Detail eingegangen und abschließend wurde der Begriff der globalen Constraints eingeführt, Vertreter der globalen Constraints wurden vorgestellt und ein Beispiel, das die Einsatzmöglichkeiten von FD-CSPs und globalen Constraints aufzeigt, wurde gegeben. Mehr Wissen über die Grundkonzepte der Constraint-Programmierung können unter anderem [16, 46, 54, 109] entnommen werden.

3 Verwandte Arbeiten

In diesem Kapitel werden verwandte Arbeiten betrachtet, insbesondere wird dabei auf Transformationsverfahren eingegangen, die ein bestehendes CSP mit dem Ziel der schnelleren Lösungsfindung in ein anderes umwandeln. Es wird in diesem Zusammenhang verdeutlicht, worin sich die existierenden Ansätze von den in dieser Dissertation vorgestellten Ansätzen unterscheiden.

Die Modellierung eines Problems als Constraint-Problem erlaubt die Verwendung verschiedener Variablen, Domänen und Constraints. Die Wahl des jeweiligen Modells kann einen sehr großen Einfluss auf den Lösungsfindungsprozess eines CSPs [15, 108, 113] haben. Aus dieser Erkenntnis heraus haben sich verschiedene Modellierungs- und Remodellierungstechniken entwickelt, wie z.B. das Hinzufügen redundanter und symmetriebrechender Constraints [64, 65], das Substituieren von Constraints [10] oder das Transformieren in ein äquivalentes CSP mit anderen Variablen, Domänen und Constraints [48, 85, 137, 159].

Die Ansätze, welche der in dieser Arbeit vorgestellten Regularisierung (Kapitel 5) am nächsten kommen, beschäftigen sich in erster Linie mit der automatischen Tabularisierung [10, 49, 112, 62, 88] von Constraint-Problemen, eine vollautomatische Regularisierung wurde hingegen in dieser Form noch nicht dokumentiert. Unter Regularisierung bzw. Tabularisierung wird in diesem Kontext das automatische Substituieren von einzelnen oder mehreren Constraints durch ein *Regular*- bzw. *Table*-Constraint verstanden. Die boolesche Skalarisierung beschreibt das automatische Substituieren einer Constraint-Menge und der involvierten Variablen durch boolesche Variablen und lineare *Scalar*-Constraints über diesen.

Die in dieser Arbeit vorgestellte boolesche Skalarisierung (Abschnitt 4.2.4 und Kapitel 5) beschreibt das Transformieren eines FD-CSPs in ein boolesches CSP ausschließlich mit *Scalar*-Constraints. Diese Art der Transformation ist am ehesten mit dem direkten Encoding verwandt, das in Abschnitt 3.2 vorgestellt wird. Beim direkten Encoding werden die ursprünglichen Constraints allerdings durch eine Klauselmenge repräsentiert, im Gegensatz dazu werden bei der booleschen Skalarisierung die ursprünglichen Constraints durch *Scalar*-Constraints über booleschen Variablen repräsentiert.

Es gilt der Unterschied zwischen Substitution und Substituierung zu beachten. Unter Substitution wird in der Mathematik das Ersetzen eines Terms durch einen anderen verstanden. Unter Substituierung wird in dieser Arbeit das Ersetzen von einzelnen oder mehreren Constraints durch ein oder mehrere andere Constraints verstanden. Es werden insbesondere die Begriffe Tabularisierung, Regularisierung und boolesche Skalarisierung verwendet, die eine Substituierung mit ausschließlich *Table-*, *Regular-* bzw. booleschen Scalar-Constraints beschreiben.

In den folgenden Abschnitten wird zunächst auf die bereits existierenden Transformationen von CSPs in binäre CSPs (Abschnitt 3.1) bzw. in SAT-Probleme (Abschnitt 3.2) eingegangen. Anschließend wird die Tabularisierung beschrieben und analog dazu erläutert, wie Regularisierung gegenwärtig in Ansätzen realisiert werden kann (Abschnitt 3.3) [1, 10, 151]. Abschließend und zusammenfassend wird eine Übersicht über existierende Verfahren zur Remodellierung von Constraint-Problemen gegeben und die weiteren Themen dieser Dissertation werden von den in diesem Kapitel aufgeführten Arbeiten abgegrenzt (Abschnitt 3.4).

3.1 Binäre Transformationen

Binäre Transformationen realisieren die Überführung von allgemeinen CSPs in binäre CSPs. Anfangs wurden die binären Transformationen hauptsächlich deswegen entwickelt, weil die binären CSPs bereits viel besser erforscht waren als allgemeine FD-CSPs. Die beiden vorherrschenden Techniken zum Lösen von binären CSPs (Backtracking kombiniert mit Vorwärtsprüfung (forward checking) und Backtracking kombiniert mit Kantenkonsistenz) existierten zunächst nur für binäre Constraints, wurden später aber auch für die Anwendung auf allgemeine FD-CSPs generalisiert [67, 102].

Die duale und die versteckte (Hidden-) Transformation sind zwei unterschiedliche Methoden, um CSPs in binäre CSPs umzuwandeln. Binäre CSPs $P = (X, D, C)$ umfassen nur Constraints, die maximal zwei Variablen enthalten ($|scope(c)| \leq 2 \mid \forall c \in C$). Die duale Transformation entwickelte sich aus der Forschergemeinschaft für relationale Datenbanken heraus und wurde in [48] von Dechter et al. in die Constraint-Community überführt. Die Hidden-Transformation ergab sich in erster Linie aus den Arbeiten des Philosophen Peirce [122]. Zusammen mit Rossi et al. konnte gezeigt werden, dass binäre Constraints dieselbe Ausdruckskraft haben wie nicht-binäre Constraints [137, 139].

Die duale Transformation

Ein dualer Graph zu einem Constraint-Netz zeichnet sich dadurch aus, dass die Knoten und Kanten im Gegensatz zur normalen Darstellung, auch Primal Graph genannt, vertauscht sind [48, 146]. Die n-ären Constraints des Ausgangs-CSPs sind daher im dualen CSP extensional in den Wertebereichen der umfassenden Variablen (Dual-Variablen genannt) enthalten. Die dualen Constraints hingegen repräsentieren, zwischen welchen Constraints des ursprünglichen CSPs gleiche Variablen (*shared variables*) existieren [137].

In [21] wird die duale Transformation eines CSPs $P = (X, D, C)$ in ein duales CSP $P^{dual} = (X^d, D^d, C^d)$ wie folgt beschrieben. Für jedes Constraint $c_i = (X_i, R_i) \in C$ des Ursprungs-CSPs wird eine duale Variable x_i^d erzeugt. Die entsprechende Domäne jeder so erzeugten Variable beinhaltet als Werte die Menge der gültigen Tupel der Relation R_i.

Für je zwei Constraints c_i und $c_j \in C$, die mindestens eine gemeinsame Variable enthalten, wird ein binäres, duales Constraint $c_{i,j}^d = (\{x_i^d, x_j^d\}, R_{i,j}^d) \in C^d$ erzeugt, das genau die beiden Variablen x_i^d und x_j^d enthält, die die ursprünglichen Constraints c_i und c_j repräsentieren. Die Relation $R_{i,j}^d$ definiert dabei alle gültigen Tupelpaare, die für die gemeinsam genutzten Variablen von c_i und c_j geeignet sind.

Die Hidden-Transformation

Bei der Hidden-Repräsentation eines CSPs werden Constraint-umfassende Variablen (Hidden-Variablen) und ursprüngliche Variablen kombiniert. Die ursprünglichen Constraints werden dabei in Hidden-Variablen transformiert, an die wiederum die ursprünglichen Variablen durch zusätzliche binäre Kompatibilitäts-Constraints, auch Hidden-Constraint genannt, gebunden werden.

Eine Beschreibung der Hidden-Transformation eines CSPs $P = (X, D, C)$ in ein Hidden-CSP $P^{hidden} = (X \cup X^h, D \cup D^h, C^h)$ ist in [21] wie folgt gegeben. Wie bei der dualen Transformation werden die dualen Variablen erzeugt, in diesem Verfahren allerdings Hidden-Variablen genannt X^h. Die Domänen der Hidden-Variablen ergeben sich analog zur dualen Transformation aus der Auflistung der zulässigen Tupel für das zugehörige Constraint. Im Unterschied zur dualen Transformation werden bei der Hidden-Transformation allerdings die ursprünglichen Variablen mit ihren Domänen weiter mit betrachtet.

Die sogenannten Hidden-Constraints C^h stellen binäre Constraints dar, die jeweils eine ursprüngliche Variable $x_i \in X$ und eine Hidden-Variable $x_j^h \in X^h$ beinhalten.

Jedes ursprüngliche Constraint c_j wird dabei durch eine Hidden-Variable $x_j^h \in X^h$ repräsentiert. Die ursprüngliche Variable $x_i \in X$ steht dann über ein Hidden-Constraint $c_{i,j}^h$ mit x_j^h in Relation, wenn $x_i \in scope(c_j)$ ist. Ein n-stelliges Constraint c mit $|scope(c)| = n$ führt somit zu n Hidden-Constraints. Es entsteht somit ein bipartiter Graph mit den beiden Knotenmengen X und X^h. Die Hidden-Constraints C^h garantieren dabei, dass jeder Variablen nur Werte zugewiesen werden, die die ursprünglichen Constraints, an denen diese Variable beteiligt ist, erfüllen.

Ein Vorteil der Hidden-Transformation gegenüber der dualen Transformation ist, dass bei diesem Vorgehen die ursprünglichen Variablen erhalten bleiben. Lösungen können direkt abgelesen werden, ohne dass eine Umrechnung anhand der Hidden-Variablen erfolgen muss. Zudem müssen weniger Constraints in der speicheraufwendigen, extensionalen Repräsentation in den umfassenden Variablen kodiert werden als bei der dualen Repräsentation. Vorteil der dualen Transformation ist demgegenüber, dass für diese Repräsentation weniger Variablen und häufig auch weniger Constraints benötigt werden.

Bacchus und van Beek [20] zeigten, dass die Repräsentation als duales CSP um Größenordnungen effizienter sein kann als die Repräsentation als FD-CSP, wenn die Anzahl der Constraints im Verhältnis zur Anzahl der Variablen gering und die Constraints restriktiv sind. Zeitgleich zeigten sie, dass die Hidden-Variable-Darstellung mittels eines modifizierten Forwardchecking-Algorithmus (FC+) in manchen Fällen exponentiell schneller gelöst werden kann als die FD-Darstellung mit dem Standard-Forwardchecking-Algorithmus.

Stergiou und Walsh veröffentlichten 1999 eine Kombination der Hidden- und der dualen Repräsentation, die Double-Repräsentation [146]. Sie vereint sowohl die Vor- als auch die Nachteile der beiden vorher genannten Repräsentationen, da beide vollständig enthalten sind. Es existieren neben den ursprünglichen und den dualen Variablen sowohl die dualen Constraints als auch die Hidden-Constraints.

Sowohl Stergiou und Walsh [146] als auch Bacchus [21] zeigten, dass Kantenkonsistenz auf die duale Repräsentation angewendet einen höheren Konsistenzgrad erreicht als lokale Konsistenz (GAC) auf der ursprünglichen n-ären Form, da bedingt durch die extensionalen Repräsentationen mehr inkonsistente Wertekombinationen aus den Domänen der Variablen herausgefiltert werden können. Auch wenn lokale Konsistenz für ein n-stelliges Constraint hergestellt wurde, sind anschließend mit den Wertebereichen der beteiligten Constraint-Variablen in der Regel Wertekombinationen möglich, die keine Lösung des Constraints darstellen. Bei einer extensionalen Constraint-Repräsentation ist das nicht möglich, da dabei ausschließlich gültige Kombinationen vorausgesetzt werden. So lässt sich in bestimmten Situationen in der dualen Repräsentation erkennen, dass ein CSP nicht lösbar ist, während dies mit lokaler Konsistenz auf dem ursprünglichen FD-CSP nicht nachweisbar ist. Kantenkonsistenz (zum Beispiel mit den PC-1 und PC-2 Algorithmen [44]) angewandt auf ein Hidden-CSP ist im Gegensatz dazu äquivalent zur lokalen Konsistenz im ursprünglichen FD-CSP [21].

Die Frage, ob die duale oder die Hidden-Transformation eines FD-CSPs in ein binäres CSP sinnvoll ist, muss in Abhängigkeit von der Problemstellung und von der Modellierung des Problems beantwortet werden. Dafür muss neben der Dauer für das Lösen des transformierten CSPs auch die Transformationsdauer berücksichtigt werden. Idealerweise kann der Modellierer ein CSP in der für ihn angenehmsten Art und Weise modellieren und der Solver löst es anschließend selbstständig und so effizient wie möglich.

Nach [20, 21] bietet die FD-Darstellung für die meisten Probleme die effizientere Repräsentation bezüglich der Anwendung von Lösungsverfahren. Für eine Klasse von stark einschränkenden Constraints kann die binäre Transformation jedoch wesentlich effizienter sein.

Die vollständige extensionale Darstellung des dualen CSPs ist vor allem dann lohnenswert, wenn die Constraints nur wenige erfüllbare Wertetupel aufweisen. Je größer die Menge der zu verwaltenden Wertetupel ist, umso schlechter ist das Laufzeitverhalten der Lösungsverfahren, da mehr Konsistenztests durchgeführt werden müssen, die bei dem ursprünglichem FD-CSP optimaler Weise redundant sind [20]. Nicht ganz so ausgeprägt ist dieses Verhalten bei den Hidden-CSPs, da bei diesen ausschließlich die n-ären Constraints in Hidden-Variablen transformiert werden und der Zugriff auf die ursprünglichen Variablen erhalten bleibt. Da globale Constraints in der Regel sehr viele gültige Wertetupel haben, bieten sich die vorgestellten Verfahren, mit wenigen Ausnahmen, nicht für die Transformation von globalen Constraints an.

In der Double-Repräsentation können die Vorteile beider Darstellungsformen genutzt werden. Weiterhin lassen sich bei Bedarf unterschiedliche hybride Repräsentationen erzeugen, die gegebenenfalls spezielle Constraints nur in der einen oder anderen Repräsentationsform enthalten. Damit sollen die Nachteile der extensionalen Darstellung für bestimmte Constraints umgangen werden [145, 146]. Dies erfordert jedoch zusätzlichen Anpassungsaufwand des Lösungsverfahrens an die Problemstellung. Das Problem bei der Transformation globaler Constraints kann damit jedoch in der Regel nicht umgangen werden.

3.2 Umwandlung in SAT-Probleme

Wie in Kapitel 2.2.1 bereits aufgeführt wurde, sind SAT-Probleme in der Lage jedes andere FD-Problem zu repräsentieren. Eine Möglichkeit der Remodellierung ist demzufolge die Überführung von FD-CSPs in boolesche CSPs. Dies erfordert eine Umwandlung sowohl aller nicht-booleschen Variablen in boolesche, als auch eine Umformung aller Constraints, angepasst an die neuen Variablen. Von Bedeutung ist dieser Ansatz hauptsächlich dadurch, dass für boolesche CSPs die sehr leistungsstarken SAT-Solver verwendet werden können.

In [56, 125, 157] sind mehrere Verfahren zur Umwandlung von FD-CSPs in boolesche CSPs erläutert. An dieser Stelle wird eine Auswahl davon exemplarisch vorgestellt. Die verschiedenen Encoding-Varianten werden in Kapitel 5, teilweise in abgewandelter Form, für die boolesche Skalarisierung verwendet. Sie stellen somit gleichzeitig konkurrierende Verfahren als auch die Grundlage für die boolesche Skalarisierung dar.

Das direkte Encoding

Nach [127] wird bei der direkten Encoding- (Verschlüsselungs-) Variante für jede mögliche Variablenzuweisung des ursprünglichen CSPs $P = (X, D, C)$ eine neue boolesche Variable $x^{de}_{i,j} \in X^{de}, i \in \{1, ..., |X|\}, j \in \{1, ..., |D_i|\}$ angelegt, die den Wert *Wahr* (1) oder *Falsch* (0) annehmen kann, je nachdem, ob die ursprüngliche Variable den entsprechenden Wert annimmt oder nicht ($x_i = d_j \leftrightarrow x^{de}_{i,j} = 1$). Für jede Zuweisung der Variablen X_i jedes Constraints $c_i = (X_i, R_i) \in C$ des ursprünglichen CSPs P, die keine gültige Belegung für c_i darstellt, wird eine Klausel über den neu erzeugten booleschen Variablen generiert, die diese Belegung ausschließt.

Zusätzlich werden für jede Variable $x \in X$ des ursprünglichen CSPs P sowohl sogenannte *addLeastOne*-Klauseln $\bigvee_{v \in D_i} x^{de}_{i,v}$ als auch *addMostOne*-Klauseln $(\neg x^{de}_{i,v_1} \vee \neg x^{de}_{i,v_2}), \forall v_1, v_2 \in D_i$ mit $i \neq j$ eingefügt, die gewährleisten, dass immer genau eine der booleschen Variablen, die Variablenzuweisungen der gleichen ursprünglichen Variablen repräsentieren, erfüllt ist.

Das mehrwertige direkte Encoding

Eine Variation des direkten Encodings ist das *mehrwertige direkte Encoding (multivalued direct encoding)* [143], welches genauso funktioniert wie das direkte Encoding mit der Ausnahme, dass die *addMostOne*-Klauseln weggelassen werden. Solch eine Codierung erlaubt mehrere gleichzeitige Zuweisungen für die gleiche ursprüngliche Variable. Eine konkrete Lösung kann allerdings wiederhergestellt werden, indem für jede ursprüngliche Variable eine der verbleibenden Zuweisungen ausgewählt wird.

Das logarithmische Encoding

Nach [56, 81, 157] basiert das logarithmische Encoding auf der Idee, jeder Variablen $x_i \in X$ des ursprünglichen CSPs $P = (X, D, C)$ genau $r = \lceil log_2 |D_i| \rceil$ viele binäre Variablen $x^{le}_{i,r}, ..., x^{le}_{i,1}$ zuzuordnen, wobei die Variablenzuweisungen für die Variablen $x^{le}_{i,r}, ..., x^{le}_{i,1}$ gerade den Index eines Domänenwertes der ursprünglichen Variable $x_i \in X$ in binärer Schreibweise darstellen.

Analog zum direkten Encoding wird auch beim logarithmischen Encoding für jede ungültige Belegung jedes ursprünglichen Constraints $c \in C$ eine Klausel erzeugt, die diese ausschließt. Da jede mögliche Belegung der Variablen $x^{le}_{i,r}, ..., x^{le}_{i,1}$ genau eine Zuweisung der ursprünglichen Variable x_i repräsentiert, sind bei diesem Verfahren keine *addLeastOne*- oder *addMostOne*-Klauseln notwendig. Bei nicht durch zwei teilbaren Domänengrößen muss allerdings verhindert werden, dass die binären Zahlen (Indizes), die keinem Domänenwert zugeordnet sind, angenommen werden. Deswegen wird jeder solche Index als eindimensionales, negatives Tupel betrachtet und in eine Klausel überführt.

Im Vergleich zum direkten Encoding benötigt das durch logarithmisches Encoding erzeugte CSP deutlich weniger Variablen (vorausgesetzt das ursprüngliche CSP beinhaltet nicht nur boolesche Variablen). Da beide Algorithmen für jedes Tupel, das ein Constraint verletzt, eine Klausel anlegen, das direkte Encoding allerdings zusätzlich noch die *addLeastOne*- und *addMostOne*-Klauseln erzeugt, werden beim logarithmischen Encoding weniger Klauseln (Constraints) benötigt. Das logarithmische Encoding hat allerdings den Nachteil, dass innerhalb einer Klausel zur Beschreibung eines negativen

Tupels eines Constraints $c = (X, T)$, deutlich mehr Literale enthalten sind als beim direkten Encoding ($|X|$ gegenüber $\sum_{i=1}^{|X|} \lceil log_2(|D_i|) \rceil$).

Es gilt also abzuwägen, ob das Einsparen von Variablen und der *addLeastOne*- und *addMostOne*-Klauseln oder das Einsparen von Literalen innerhalb der Klauseln vorteilhafter für das Lösen von SAT-Problemen ist. In der Theorie konnte Walsh in [157] zeigen, dass Unit-Propagation bei DPLL-SAT-Solvern ([43]) schwächer ist als beim logarithmischem Encoding. Auch in der Praxis konnte häufig festgestellt werden, dass das logarithmische Encoding eine schlechtere Performance bietet als das direkte Encoding. Eine nennenswerte Ausnahme ist dabei das Graph-Färbungsproblem, bei dem ein statisches Variablenordnungsschema verwendet wird [155]. Trotzdem bieten Verfahren, die auf dem direkten Encoding basieren, auch bei vielen Graph-Färbungsproblemen eine bessere Performance als solche, die auf dem logarithmischen Encoding basieren [57]. Weitere Anwendungen, für die gezeigt werden konnte, dass das direkte Encoding besser agiert, sind das Hamilton-Pfad-Problem [76] und Planungsprobleme [53].

Das Support-Encoding

Nach [59] transformiert das Support-Encoding die Variablen und Domänen genauso wie das direkte Encoding, stellt die Constraints des ursprünglichen CSPs allerdings nicht durch Klauseln dar, die nicht-erlaubte Tupel repräsentieren, sondern durch solche, die die erlaubten Tupel repräsentieren. Die angesprochenen Klauseln werden *Support-Klauseln* genannt. Es müssen dabei ebenfalls die aus dem direkten Encoding bekannten *addLeastOne*- und *addMostOne*-Klauseln erzeugt werden.

Das Support-Encoding ist nur für das Umwandeln von binären CSPs geeignet. Eine Überführung von FD-CSPs in binäre CSPs ist allerdings immer möglich, entsprechende Transformationen wurden in Abschnitt 3.1 vorgestellt.

Das minimale Support-Encoding

Das minimale Support-Encoding wurde in [18] definiert und transformiert ein FD-CSP in ein SAT-Problem. Im Wesentlichen entspricht das Vorgehen dabei dem des Support-Encodings, allerdings werden für jedes Constraint c mit Variablenmenge $scope(c) = \{x_1, x_2\}$ nur entweder für die Variable x_1 oder x_2 alle Support-Klauseln angegeben.

Anders als beim Support-Encoding erzwingt das minimale Support-Encoding keine lokale Konsistenz (arc-consistency) beim Anwenden von Unit-Propagation, dafür kann die Anzahl der Klauseln verringert werden.

Im Vergleich zum direkten und zum logarithmischen Encoding erzeugt das (minimale) Support-Encoding weniger Klauseln, wenn die zu transformierenden ursprünglichen Constraints einzeln betrachtet mehr Werte ausschließen als zulassen. Beispielsweise hat das Ungleichheits-Constraint $x \neq y$ für zwei Variablen mit Domänengröße $n > 1$ genau $n * (n - 1)$ zulässige und nur n unzulässige Tupel, wohingegen das Gleichheits-Constraint

$x = y$ sich genau umgekehrt verhält. Für das Ungleichheits-Constraint bietet sich das direkte Encoding somit eher an und für das Gleichheits-Constraint das Support-Encoding. Es besteht auch die Möglichkeit, verschiedene Encoding-Varianten zu kombinieren, um die verschiedenen Vorteile (aber auch Nachteile) der Verfahren gleichzeitig nutzen zu können.

Das Regular-Encoding

Eine weitere Encoding-Variante zum Umwandeln von binären CSPs in boolesche CSPs ist das Regular-Encoding [13]. Die Grundidee des Regular-Encodings ist es, jede Variable $x_i \in X$ des ursprünglichen CSPs mit Domäne $D_i = \{0, ..., m\}$ durch einen Vektor boolescher Variablen $\overrightarrow{U_i} = (x^{re}_{i,1}, ..., x^{re}_{i,m})^T$ zu ersetzen, wobei für den Vektor $\overrightarrow{U_i}$ die Einschränkung gilt, dass eine Variable $x^{re}_{i,j}$ nur dann den Wert 1 annehmen darf, wenn alle Variablen $x^{re}_{i,k}$, mit $k < j$ auch den Wert 1 annehmen. Jede solche Variable $x^{re}_{i,j}$ wird als Regular-Variable bezeichnet.

Jeder Vektor $\overrightarrow{U_i}$ hat demzufolge die Form $(1, 1, ..., 1, *, *, *, ..., *, 0, 0)^T$, wobei 1 für Variablen steht, denen der Wert 1 zugeordnet wurde, $*$ für Variablen, denen noch kein Wert zugeordnet wurde und 0 für Variablen, denen der Wert 0 zugeordnet wurde. Die Anzahl der 1en in $\overrightarrow{U_i}$ soll dabei gleich dem Wert $d_i \in D_i$ entsprechen, der der Variablen x_i zugeordnet ist.

Zwei wesentliche Vorteile des Regular-Enocdings sind nach [22] zum einen, dass Intervalle gut repräsentiert und zum anderen, dass die Grenzen der Intervalle effektiv propagiert werden können. Die Domäne einer ursprünglichen Variable $x_i \in X$ kann mit lediglich zwei Bedingungen $x^{re}_{i,l-1} = 1$ und $x^{re}_{i,u} = 0$ auf ein beliebiges Intervall $[l, u]$ begrenzt werden. Auch wenn das Regular-Encoding ideal für die Darstellung von Intervallen geeignet ist, erlaubt es nicht nur das Erstellen von Grenzenkonsistenz, sondern auch von lokaler Konsistenz.

Ein Nachteil dieses Verfahrens ist, dass, wie beim direkten und auch beim Support-Encoding, die Anzahl der Regular-Variablen linear mit der Domängröße jeder einzelnen Variable wächst. Bei großen Domänen trägt die hohe Anzahl an Variablen erheblich zu einer erhöhten Lösungszeit der SAT-Solver bei [22].

Das polynomiale-watchdog Encoding

Das Encoding von linearen pseudo-booleschen CSPs (siehe Definition 2.13) in SAT-Probleme wurde in [23] vorgestellt. Es basiert auf einer Zerlegung und Repräsentation der Koeffizienten der linearen Constraints durch Regular-Variablen (siehe dazu den vorherigen Abschnitt zum Regular Encoding). Es werden dabei sogenannte *polynomial watchdogs* (polynomiale Wachhunde) verwendet. Ein *polynomial watchdog* ist eine Formel, die einer booleschen Variablen den Wert 1 zuordnet, wenn ein gegebenes Constraint c nicht erfüllt ist.

Insbesondere wurde in [23] auch gezeigt, dass jedes pseudo-boolesche Constraint mit Ganzzahlgewichten in eine KNF-Formel mit polynomial vielen Variablen und Klauseln überführt werden kann, die mittels Unit-Propagation lokale Konsistenz erreichen. Diese Überführung stellt die Grundlage für viele lineare pseudo-boolesche Solver dar.

Das Buch „Bridging Constraint Satisfaction and Boolean Satisfiability " gibt in Kapitel 4 [126] eine Übersicht über die hier vorgestellten Encoding-Varianten und verweist auf weitere.

In der Grundform sind alle vorgestellten Encoding-Varianten für die vollständige Umwandlung eines CSPs in ein SAT-Problem gedacht. Soll ein Substituieren von Teil-CSPs durch boolesche Elemente ermöglicht werden, so muss in allen Varianten die ursprüngliche Variablenmenge beibehalten werden und eine Verknüpfung der ursprünglichen Variablen mit den neu erzeugten booleschen Variablen erstellt werden. Diese Verknüpfung wird durch zusätzliche Constraints geschaffen. Beim direkten Encoding kann dies zum Beispiel mit den Constraints $(x_i = d_j) \leftrightarrow (x^{de}_{i,j} = 1), \forall i \in \{1, ..., n\}, j \in \{1, ..., |D_i|\}$ realisiert werden.

Der Hauptnutzen dieser Art der Transformation ist die vollständige Umwandlung eines CSPs in ein boolesches CSP, um dieses anschließend mit einem SAT-Solver zu lösen. SAT-Solver können Probleme in der Regel sehr schnell lösen, allerdings verschiebt sich bei diesem Vorgehen der Zeitaufwand vom Lösen des CSPs in die Transformation. Der entscheidende Nachteil aller in diesem Abschnitt aufgeführten Umwandlungen gegenüber der in dieser Arbeit entwickelten Regularisierung und der booleschen Skalarisierung ist, dass keine direkten Transformationen für globale Constraints existieren, bei denen nicht alle gültigen oder ungültigen Tupel abgearbeitet werden müssen. Im Gegensatz zu den in dieser Arbeit entwickelten Substituierungen, sind die bisherigen Verfahren daher oftmals zu zeitaufwendig, als dass diese zu einer beschleunigten Lösungssuche bei realen Problemen mit globalen Constraints führen.

3.3 Tabularisierung und Regularisierung

Im Gegensatz zu den vorgestellten Transformationen in SAT-Probleme haben die Tabularisierung und die Regularisierung den Vorteil, dass sie punktuell und effektiv auf einzelne Teilbereiche von CSPs angewendet werden können und auch das Substituieren von Teil-CSPs zu einem signifikanten Geschwindigkeitsvorteil führen kann.

Sowohl bei der Tabularisierung als auch bei der Regularisierung werden entweder einzelne Constraints oder Teil-CSPs substituiert. Für ein gegebenes CSP $P = (X, D, C)$ ergibt sich ein Teil-CSP gerade aus einer Teilmenge C' der Constraints C mit den zugehörigen Variablen $X' \in X$ und Domänen $D' \in D$. Dabei muss die Menge der Variablen X' mindestens alle Variablen aus den Gültigkeitsbereichen der Constraints C' abdecken ($\forall c \in C', scope(c) \subseteq X'$).

Definition 3.1: *Teil-CSP.* Sei ein CSP $P = (X, D, C)$ gegeben, ein CSP $P' = (X', D', C')$ wird Teil-CSP von P genannt, wenn $X' \subseteq X, D' \subseteq D$ und $C' \subseteq C$ mit $scope(c) \subseteq X' \; \forall \, c \in C'$ ist und die Menge der Domänen D' alle Domänen D_i der zugehörigen Variablen $x_i \in X'$ enthält.

Sowohl für die Tabularisierung als auch für die Regularisierung ist es zunächst notwendig, substituierbare Teil-CSPs zu identifizieren, im Anschluss daran kann die eigentliche Substituierung, also das Umwandeln von Constraints oder Teil-CSPs in *Table*- bzw. *Regular*-Constraints erfolgen. Im Folgenden werden die beiden Bereiche vorgestellt und es wird auf entsprechende Literatur dazu verwiesen.

3.3.1 Das Finden von substituierbaren Teil-CSPs

Das Finden von substituierbaren Teil-CSPs wurde in der Vergangenheit den jeweiligen Constraint-Programmierern überlassen. Über Annotationen können zum Beispiel in MiniZinc Prädikate kenntlich gemacht werden, die in der Folge automatisch durch *Table*-Constraints ersetzt werden [49]. Erst in einem 2018 veröffentlichten Papier [10] wurden erstmals vier Möglichkeiten zum automatischen Erkennen von Teilproblemen für das Tabularisieren vorgestellt. Im Folgenden werden diese kurz vorgestellt.

Heuristiken für die Tabularisierung

Die Heuristiken aus [10] wurden für den Constraint-Modellierungsassistenten *SavileRow* [117] entwickelt und arbeiten auf dem jeweils zum modellierten CSP erstellten *abstrakten Syntaxbaum (abstract syntax tree, AST)*. *Savile Row* bietet dem Nutzer eine High-Level Sprache für die Spezifikation von Constraint Problemen und übersetzt diese automatisch in Eingabesprachen für Constraint-Solver wie z.B. Gecode, FlatZinc- oder SAT-Solver. Ein AST ist eine Datenstruktur, die bei der Interpretation und Compilation von Quellcode als Zwischenschritt entsteht. In einem AST ist ein gegebenes Programm baumartig und losgelöst von der konkreten Syntax gespeichert. Details zu Savile Row und seiner AST-Erzeugung können [117] und [118] entnommen werden.

Doppelte-Variablen

Die Doppelte-Variablen-Heuristik erkennt Constraints, die prädestiniert dafür sind, ein schwaches Konsistenzlevel zu haben. In den meisten Fällen kann ein Propagator nur dann lokale Konsistenz erreichen, wenn er keine doppelten Variablen enthält. Ein Vertreter dafür ist das *Global Cardinality*-Constraint, für das bekannt ist, dass das Erreichen lokaler Konsistenz beim Vorkommen doppelter Variablen NP-vollständig ist [33], wohingegen ein Polynomialzeit-Propagator existiert, wenn keine Variable mehrfach auftritt [130]. In der Folge können die so gefundenen Constraints, deren Propagatoren aufgrund des Vorkommens doppelter Variablen voraussichtlich nur ein schwaches Konsistenzniveau erreichen, tabularisiert werden.

Großer AST

Enthält ein AST eines CSPs viele Knoten, von denen allerdings nur wenige unterschiedliche Variablen darstellen, so ist davon auszugehen, dass das eigentliche Constraint nicht-kompakt repräsentiert wurde. Ein Beispiel für ein Constraint, das einen großen AST aufspannt, aber wenige Variablen enthält, ist das *Element*-Constraint, welches ausdrückt, dass der x-te Wert eines Arrays M gleich dem Wert y sein muss ($\mathit{element}(x, y, M) := M[x] = y$). Besonders bei großen Matrizen für M kann die Anzahl der Knoten im AST die Anzahl der Variablen im Constraint um ein Vielfaches übersteigen.

Die Große-AST-Heuristik identifiziert Constraints, die einen AST erzeugen, der mindestens n mal so groß ist wie die Anzahl verschiedener Variablen, die in diesem AST vorkommen.[1] Der Grundgedanke dieser Heuristik ist es, dass für Constraints, die einen großen AST erzeugen, möglicherweise ein *Table*-Constraint existiert, das die Bedingung kompakter ausdrückt. Es wird dabei vermutet, dass ein Constraint, welches durch einen kleineren AST dargestellt wird, in der Regel schneller propagiert, als ein Constraint, das durch einen größeren AST repräsentiert wird. Diese Vermutung trifft nicht immer zu, erzielte allerdings zumindest für die in [10] angegebenen Beispiele gute Ergebnisse.

Schwache Propagation

Die Schwache-Propagations-Heuristik soll Constraints mit schwachem Konsistenzlevel finden, die Constraints mit hohem Konsistenzlevel bei deren Propagation im Weg stehen. Seien zwei Constraints $c_1 = \mathit{AllDifferent}(\{x_1, x_2, x_3\})$ und $c_2 = (x_1 = 2x_4 + 3x_5)$, mit zwei Propagatoren, die lokale Konsistenz für das *AllDifferent*-Constraint und Grenzenkonsistenz für das *Summen*-Constraint erzeugen, gegeben. Das Tabularisieren des *Summen*-Constraints erhöht das Konsistenzlevel, so dass möglicherweise mehr Werte aus den Domänen von x_1, x_2 oder x_3 entfernt werden können und dadurch der *AllDifferent*-Propagator ebenfalls mehr Werte aus den Domänen der jeweils anderen Variablen ausschließen kann.

Gleiche Variablemengen

Haben zwei oder mehr Constraints $C' \subseteq C$ den gleichen *scope*, so kann diese Menge C' von Constraints, also die Konjunktion aller Constraints in C', in der Regel nur ein niedriges Konsistenzlevel herbeiführen. Dies gilt in der Regel selbst dann, wenn die einzelnen Constraints $c_i \in C'$ alle ein hohes Konsistenzlevel, wie zum Beispiel lokale Konsistenz, durch Propagation erreichen würden. Ein sehr populäres Beispiel dafür wurde bereits in Abbildung 2.1 dargestellt. Die drei Ungleichheit-Constraints über booleschen Variablen verfügen alle über Propagatoren, die lokale Konsistenz herstellen, trotzdem können diese nicht (ohne Suche) bestimmen, dass das CSP keine Lösung hat. Ein globales Constraint (z.B. das *AllDifferent*-, *Regular* oder *Table*-Constraint) würde diese Inkonsistenz ohne Suche erkennen können.

---

[1] In [10], wurde n mit Wert 5 belegt.

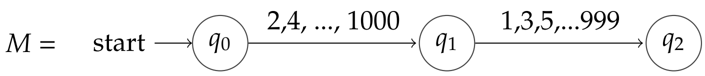


Abbildung 3.1: Ein sehr kompakter Automat (DFA) zur Darstellung von 250.000 Lösungen.

Die Gleiche-Variablemengen-Heuristik identifiziert solche Mengen von Constraints C' und kombiniert jede solche Menge C' von Constraints zu einem *Table*-Constraint, welches exakt die gleichen gültigen Belegungen erlaubt, wie es die ursprünglichen Constraints in C' tun. Das *Table*-Constraint erzeugt dabei allerdings lokale Konsistenz, wodurch in vielen Fällen der Lösungsprozess beschleunigt werden kann.

Heuristiken für die Regularisierung

Für das Finden substituierbarer Teilprobleme für die Regularisierung von Constraints und Teil-CSPs existieren im Gegensatz zur Tabularisierung keine expliziten Algorithmen. Prinzipiell können allerdings die gleichen Heuristiken wie bei der Tabularisierung verwendet werden. Zu beachten ist allerdings, dass es mindestens einer angepassten Parametrisierung bedarf.

Die Tatsache, dass der Aufwand der Propagation eines Constraints $Regular(X, M)$ in erster Linie von der Anzahl der Übergänge in M abhängt, kann als Grundlage für Heuristiken zum Erkennen von Teil-CSPs, die für die Regularisierung geeignet sind, dienen. Eine solche Heuristik müsste dann abschätzen oder berechnen, ob ein levelbasierter DFA M, der als Substituierung für ein oder mehrere ursprüngliche Constraints in Frage kommt, besonders wenige Übergänge besitzt oder nicht.

Mittels levelbasierten DFAs können teilweise sehr viele Lösungen sehr kompakt dargestellt werden. Der Automat M in Abbildung 3.1 stellt zum Beispiel alle Kombinationen von zwei Zahlen dar, bei denen die erste Zahl gerade ist und zwischen 2 und 1000 liegt und die zweite Zahl ungerade ist und zwischen 1 und 999 liegt. Es gibt dementsprechend $500 * 500 = 250.000$ verschiedene Wertepaare, die diese Eigenschaft erfüllen und alle werden durch den Automaten M beschrieben. Ein *Table*-Constraint hingegen müsste alle 250.000 Tupel explizit auflisten. Dieses Beispiel soll demonstrieren, dass obwohl das *Table*- und das *Regular*-Constaint die gleiche Mächtigkeit aufweisen, sie unterschiedlich gut für verschiedene Situationen anwendbar sind. Aus diesem Grund müssen weitere Heuristiken zum Finden von Teil-CSPs erforscht werden, die besonders für die Regularisierung geeignet sind.

3.3.2 Das Substituieren von Constraints durch *Table*- bzw. *Regular*-Constraints

Die eigentliche Tabularisierung, also das Ersetzen eines oder mehrerer ausgewählter Constraints $C' \subseteq C$ eines CSPs $P = (X, D, C)$, lässt sich durch das Formulieren und Lösen eines Teil-CSPs $P' = (X', D', C')$ mit $X' \subseteq X, D' \subseteq D, C' \subseteq C$ realisieren. Zu beachten ist, dass sowohl die Tabularisierung als auch die Regularisierung nur dann sinnvoll sind, wenn dadurch der gesamte Lösungsprozess inklusive der Transformation schneller als im ursprünglichen CSP P ausgeführt werden kann. Damit die Zeit für die Transformation nicht zu viel Zeit in Anspruch nimmt, sollte das zu ersetzende Teil-CSP möglichst schnell lösbar sein.

Ob ein CSP $P = (X, D, C)$ schnell lösbar ist, hängt in erster Linie von seiner Größe ($size(P)$), den Constraints C und der Anzahl der Überlagerungen der Constraints ab. Welche Constraints bei welchen Überlagerungen als komplex oder weniger komplex anzunehmen sind, ist im Vorfeld, ohne das CSP zu lösen, nicht zuverlässig vorhersagbar. Ein Maß für die Größe eines CSPs kann allerdings die Größe seines Suchraumes darstellen. Die Größe des Suchraumes ergibt sich dabei aus dem Produkt der Domänengrößen aller Variablen. Analog lässt sich die Größe eines einzelnen Constraints bestimmen.

Substituieren von Constraints durch *Table*-Constraints

Zunächst werden die Lösungen des zu substituierenden Teil-CSPs erstellt und in Tupel überführt. Ist die Anzahl der Lösungen des Teil-CSPs, also die Anzahl der Tupel, die für das *Table*-Constraint erzeugt werden sollen, zu groß, so ist eine anschließende Tabularisierung auf Grund von Ressourcenengpässen nicht immer möglich. In [10] wird als Obergrenze 10.000 Tupel angegeben. Diese Grenze kann allerdings abhängig vom jeweiligen Teil-CSP, der jeweiligen Tupelgröße und der verwendeten Hardware stark differieren. In [40] wird anhand des Still-Life-Problems (das Finden einer stabilen Population in dem Spiel Game-of-Life) gezeigt, dass auch *Table*-Constraints mit sehr vielen Tupeln (ca. 76 Millionen über 30 Variablen) zu einer Performanceverbesserung führen können.

Substituieren von Constraints durch *Regular*-Constraints

Das Substituieren von Constraints oder Teil-CSPs durch das *Regular*-Constraint kann ähnlich wie das Tabularisieren durch Lösen von Teil-CSPs und anschließendes Ersetzen dieser durch ein resultierendes *Regular*-Constaint erfolgen. Ein Ansatz dafür wird unter anderem in dieser Arbeit in Kapitel 5 gegeben. In der Literatur vertreten ist bisher allerdings eher die Herangehensweise, globale Constraints direkt durch *Regular*-Constaints zu ersetzen. Algorithmen für das Transformieren von *Scalar*- oder *Count*-Constraints sind zum Beispiel in [151] und [1] gegeben.

In [1] wird eine Vielzahl weiterer Transformationen von globalen Constraints zu deterministischen endlichen Automaten (DFAs) aufgelistet. Bisher mangelt es allerdings an Heuristiken, die erzeugten DFAs weiter untereinander und mit den Automaten aus der Lösung von Teil-CSPs zu kombinieren.

3.4 Zusammenfassende Betrachtungen

In diesem Abschnitt wird ein zusammenfassendes Bild über den aktuellen Stand der Forschung gegeben. Bisher blieben in diesem Kapitel Methoden zum Erkennen redundanter [39, 41, 140] und symmetriebrechender Constraints [60, 84, 86, 158] unerwähnt. Beides sind relevante Technologien, die ebenfalls durch Remodellierung von CSPs versuchen, eine Verbesserung der Lösungsgeschwindigkeit zu erzielen. Im Gegensatz zu den anderen vorgestellten Transformationen konkurrieren diese Verfahren allerdings nicht mit der Regularisierung. Sie sind viel mehr, genauso wie die Parallelisierung, als unabhängig zur Regularisierung existierende Optimierungen zu betrachten. Aus diesem Grund wird in dieser Arbeit nicht weiter auf die beiden Verfahren eingegangen.

In Abbildung 3.2 ist ein Ausschnitt des dreistufigen Ablaufs (1. Modellieren, 2. Remodellieren und Konfigurieren, 3. Lösen) zum Lösen eines Constraint-Problems mittels Remodellierung dargestellt. Die gestrichelten Container stellen dabei Teile dar, für die bisher keine oder nur unzureichende Vorgehen existieren. Am Anfang jedes Constraint-Programmes steht die Modellierung des Problems. Theoretisch ist die Modellierung des CSPs direkt als spezielles (*Regular*-, *Table*-, binäres oder boolesches) CSP möglich, allerdings erschwert das die Modellierung teilweise so gravierend, dass diese nicht mehr praktikabel ist. In der Regel besteht der Wunsch, ein Constraint-Problem so frei und dadurch anwendungsnah wie möglich modellieren zu können.

Wie in den Abschnitten 3.1 und 3.2 gezeigt wurde, existieren Verfahren zur Umwandlung von beliebigen CSPs in CSPs, die von spezialisierten Solvern gelöst werden können (binäre und boolesche CSPs). Diese haben allerdings den Nachteil, dass die effektiven, spezialisierten Solver in der Regel nur eingesetzt werden können, wenn das gesamte Problem transformiert wird. Eine Fokussierung auf besonders markante Ausschnitte sowie die Transformation und das effektive Lösen dieser Ausschnitte ist dabei nicht direkt (das heißt ohne eine Umformung des gesamten CSPs) möglich. Wie gezeigt wurde, müssen in der Regel alle positiven oder negativen Tupel aller Constraints für solch eine Umwandlung abgearbeitet werden. Durch die zunehmende Nutzung großer Constraints (z.B. der globalen Constraints) übersteigt der Aufwand einer solchen Umwandlung von CSPs oftmals den Aufwand der Lösung des ursprünglichen CSPs. Ist eine solche Transformation allerdings für ein CSP möglich und schnell durchführbar, so kann im Anschluss daran oft auf sehr leistungsstarke, spezialisierte Solver, wie zum Beispiel SAT-Solver oder binäre Solver, zugegriffen werden.

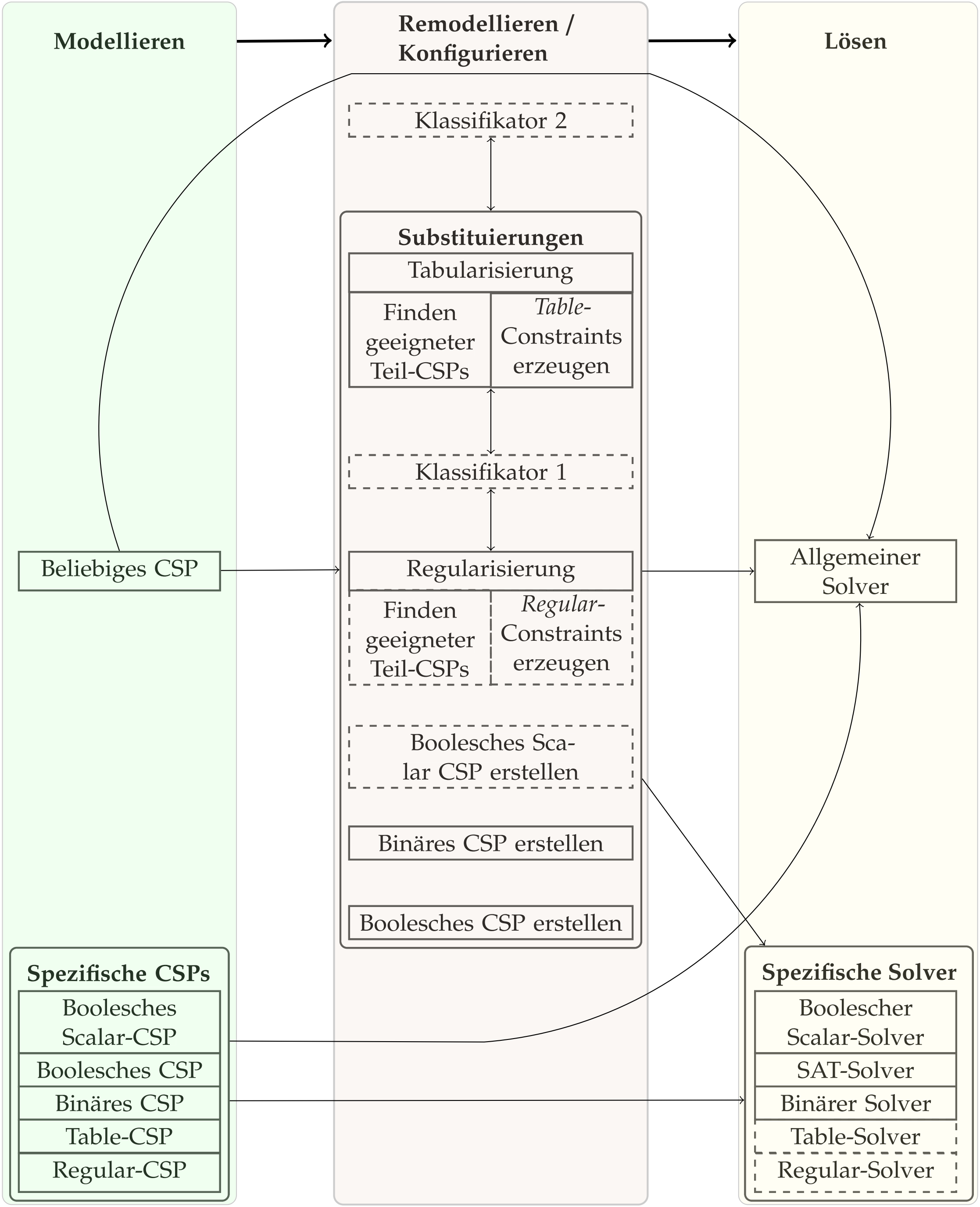


Abbildung 3.2: Der dreistufige Ablauf beim Lösen eines Constraint-Problems unter Verwendung von Remodellierungsansätzen.

Wie in Kapitel 5 gezeigt wird, ist die Transformation ausgewählter globaler Constraints in boolesche *Scalar*-Constraints wesentlich schneller möglich als eine Umwandlung dieser in SAT-Probleme mit den zuvor genannten Methoden. Die boolesche Skalarisierung wurde bisher in der Literatur kaum betrachtet, statt dessen geht entsprechende Literatur in der Regel davon aus, dass das CSP bereits als boolesches *Scalar*-CSP vorliegt und zum Beispiel durch Überführung in ein SAT-Problem mit einem SAT-Solver gelöst wird (siehe dazu z.B. [23]).

Die Tabularisierung hat, im Gegensatz zu den zuvor genannten Verfahren, den Vorteil, dass sie auch auf Teilbereiche eines CSPs anwendbar ist. Es können also gezielt „langsame“ Teile eines CSPs durch effektivere *Table*-Constraints substituiert werden. Erste Heuristiken zum Erkennen geeigneter Teilbereiche wurden entwickelt (siehe Abschnitt 3.3.1), wodurch eine (voll-) automatische Tabularisierung in Zukunft ermöglicht werden kann.

Bei der Regularisierung werden im Gegensatz zur Tabularisierung andere Datenstrukturen und Algorithmen verwendet, wodurch es unter anderem nicht explizit notwendig ist, alle Tupel eines Constraints bei der Transformation abzuarbeiten. Die kompakte Darstellung der im *Regular*-Constraint verwendeten Automaten erlaubt somit auch das Transformieren sehr großer (oftmals auch globaler) Constraints. Analog zur Tabularisierung gilt es auch bei der Regularisierung zunächst Heuristiken zu erforschen, die geeignete Teilprobleme erkennen. Eine direkte Übernahme der Heuristiken aus der Tabularisierung ist an dieser Stelle zu unpräzise, um die gewünschten Ergebnisse zu erzielen. Zusätzlich zum Erkennen von substituierbaren Teilproblemen müssen auch Verfahren erarbeitet werden, die verschiedene Constraints (insbesondere auch globale Constraints) in *Regular*-Constraints transformieren, diese zusammenfügen und komprimieren.

Als weitere zukünftige Forschungsziele ergeben sich zwei Klassifikatoren. Zum einen betrifft das die Entwicklung einer Heuristik, die abschätzt, wann (bei gleichbleibendem Solver) die Tabularisierung und wann die Regularisierung besser für ein CSP geeignet ist (Klassifikator 1 in der Abbildung). Zum anderen behandelt das die Entwicklung einer Heuristik, die abschätzt, welche aus der Fülle an möglichen Substituierungen am geeignetsten für ein CSP ist, wenn der zugrundeliegende Solver ebenfalls austauschbar ist (Klassifikator 2 in der Abbildung).

Zusätzlich sollten weitere Ansätze zur Transformation von CSPs verfolgt und mehr spezialisierte Solver, wie z.B. Regular- oder Table-Solver, geschaffen werden. Auch das automatische Konfigurieren (z.B. das Auswählen der Suchstrategie) eines Solvers anhand der verwendeten Constraints erscheint in Verbindung mit der Möglichkeit, die Anzahl an *Regular*- oder *Table*-Constraints innerhalb eines CSPs zu erhöhen, sehr vielversprechend.

Abgrenzung dieser Arbeit zu verwandten Arbeiten

In diesem Kapitel wurde ein Überblick über die aktuelle Forschung zum Thema Remodellierung von CSPs gegeben. Gleichzeitig wurden noch existierende Lücken in diesem Forschungsbereich aufgezeigt. In der Folge ergaben sich die Forschungsfragen, wie sie bereits in Abschnitt 1.2 formuliert wurden. Abgrenzend von den zuvor beschriebenen Arbeiten sind in dieser Arbeit die folgenden neuen Inhalte entstanden:

FD-vollständige Constraints (Kapitel 4):

- Es wird sowohl ein allgemeines Konzept für Constraints, die alle anderen FD-Constraints substituieren können, als auch für CSPs, die alle anderen FD-CSPs substituieren können, entwickelt.
- Die Vorteile, welche durch Substituierungen (mittels FD-vollständigen Constraints) für ein CSP entstehen können, werden ausgearbeitet.

Verfahren zur Substituierung von Constraints und Teil-CSPs (Kapitel 5):

- Es wird ein universelles Verfahren entwickelt, das jedes beliebige FD-Constraint in ein *Regular*-Constraint bzw. jedes FD-CSP in ein *Regular*-CSP überführen kann.
- Es werden zwei universelle Verfahren zur Umwandlung von FD-CSPs in boolesche *Scalar*-CSPs entwickelt.
- Für viele globale Constraints werden direkte Transformationen in *Regular*-Constraints aus der Literatur zusammengetragen, weiterentwickelt und neue Transformationen entwickelt, die eine Substituierung ohne Aufzählung aller positiven oder negativen Belegungen eines Constraints erlauben.
- Für viele globale Constraints werden direkte Transformationen in boolesche *Scalar*-Constraints entwickelt, welche eine Substituierung ohne Aufzählung aller positiven oder negativen Belegungen eines Constraints erlauben.
- Es werden Verfahren entwickelt, um beliebige logische Meta-Constraints in *Regular* und boolesche *Scalar*-Constraints zu überführen.
- Es werden weitere Substituierungen aus den bis dahin erarbeiteten abgeleitet, um weitere Transformationen zu schaffen, die für spezielle CSPs zu Beschleunigungen im Lösungsprozess führen. Übergeordnetes Ziel ist es dabei für alle CSPs Transformation zu entwerfen, die den Lösungsprozess des CSPs beschleunigen und welche im Vorfeld des Lösens eindeutig und schnell ermittelt werden können.

Evaluation der neuen Substituierungen (Kapitel 6):

- Die speziellen Substituierungen für globale Constraints werden evaluiert und es wird beurteilt, inwiefern diese für die Substituierung globaler Constraints eingesetzt werden sollten.

- Die neu entwickelten Substituierungsverfahren werden auf reale Anwendungsbeispiele angewendet und evaluiert.
- Es wird eine Heuristik entwickelt, die versucht eine bestmögliche Vereinigung von *Regular*-Constraints zu erzeugen, so dass das resultierende CSP mit minimalem Zeitaufwand gelöst werden kann.
- Ein prototypischer Klassifikator wird entwickelt, der abschätzt, wann eine Regularisierung oder eine Tabularisierung am geeignetsten für ein CSP ist.
- Die boolesche Skalarisierung wird dahingehend untersucht, ob die Anzahl der verwendeten booleschen Variablen oder *Scalar*-Constraints einen Einfluss auf die Lösungsgeschwindigkeit hat.

4 Finite Domain-vollständige Constraints

In diesem Kapitel wird ein allgemeines Konzept für Constraints, die in der Lage sind alle anderen FD-Constraints zu substituieren, entwickelt. Wir bezeichnen diese Constraints in der Folge als *Finite Domain-vollständige Constraints*.

Der Suchraum von FD-CSPs ist auf der einen Seite endlich, auf der anderen Seite kann er allerdings auch so groß sein, dass die Lösungsdauer einen akzeptablen Zeitaufwand oder die Ressourcen des Rechners übersteigt. Aus diesem Grund besteht ein hohes Interesse daran, den Lösungsprozess von FD-CSPs zu beschleunigen. Eine Möglichkeit der Beschleunigung ist das Umwandeln einzelner Constraints, Mengen von Constraints oder ganzer Constraint-Probleme in Constraints oder Constraint-Probleme einer anderen Art. Eine Übersicht über verbreitete Umwandlungsverfahren wurde in Kapitel 3 gegeben.

In der Praxis ergibt sich damit die Möglichkeit CSPs in äquivalente CSPs umzuwandeln, die nur ein einziges Constraint oder aber nur einen Typ von Constraints (z.B. nur *Table*- oder *Regular*-Constraints, oder nur boolesche Klauseln) enthalten. Inspiriert vom Konzept der NP-vollständigen Probleme wurden im Zusammenhang mit dieser Arbeit die Konzepte der Finite Domain-vollständigen Constraints (fd-complete constraints) und der Finite Domain-vollständigen CSPs (fd-complete CSPs) entwickelt und in [99] veröffentlicht.

Das charakteristische Merkmal FD-vollständiger Constraints ist es, dass sie aufgrund ihrer Struktur in der Lage sind, jedes andere FD-Constraint zu repräsentieren. Aus diesem Grund sind sie prädestiniert dafür als Substituierung für andere Constraints zu fungieren. Ein Vorteil gegenüber anderen Constraints ist, dass nicht getestet werden muss, ob eine Substituierung durchgeführt werden kann[1], sondern lediglich, ob diese Substituierung eine Zeitersparnis mit sich führt oder nicht.

Definition 4.1: ***Finite Domain-vollständige Constraints.*** Ein Finite Domain-vollständiges Constraint (fd-complete Constraint) ist ein Finite Domain-Constraint, welches aufgrund seiner Struktur jedes andere Finite Domain-Constraint $c \in C$ eines CSPs $P = (X, D, C)$ repräsentieren kann, ohne dass Anpassungen von Domänen $D_i \in D$, Variablen $x_i \in X$ oder anderer Constraints $c_j \in C, c_j \neq c$ in P notwendig sind [99].

---

[1] In der Theorie kann eine Substituierung eines FD-Constraints durch ein FD-vollständiges Constraint immer durchgeführt werden, in der Praxis kann es aber Hardware- oder Zeitbegrenzungen geben.

In diesem Kapitel werden zunächst zwei Beispiele für FD-vollständige Constraints (das *Table*- und das *Regular*-Constraint) gegeben (Abschnitt 4.1), bevor auf die Vorteile von FD-vollständigen Constraints eingegangen wird (Abschnitt 4.2).

4.1 Beispiele für FD-vollständige Constraints

In diesem Abschnitt werden sowohl das *Table*-Constraint als auch das *Regular*-Constraint als Vertreter für FD-vollständige Constraints vorgestellt und es wird erläutert, wie diese verwendet werden können, um jedes beliebige FD-Constraint zu repräsentieren.

4.1.1 Das *Table*-Constraint

Das Constraint $\mathit{Table}(\{x_1, ..., x_n\}, T)$ garantiert, wie in Abschnitt 2.5 beschrieben, dass die Variablen $x_1, ..., x_n$ nur Werte $d_1, ..., d_n$ annehmen können, für die das Tupel $(d_1, ..., d_n)$ ein Element der Tupelliste T ist.

Da FD-Constraints aufgrund der endlichen Domänen nur endlich viele gültige Variablenbelegungen repräsentieren können, kann jedes FD-Constraint durch das *Table*-Constraint repräsentiert werden. In der Praxis existieren allerdings Einschränkungen auf Grund von Speicherlimitierungen und begrenzter Rechenzeit. Ein FD-Constraint c^{fd} kann seine endlich vielen zulässigen Variablenbelegungen implizit oder explizit repräsentieren. Sind die zulässigen Variablenbelegungen bereits explizit dargestellt, so können diese auch in das *Table*-Constraint überführt werden. Sind die zulässigen Variablenbelegungen allerdings kompakt implizit dargestellt (wie es häufig bei globalen Constraints der Fall ist), so ist es möglich, dass eine explizite Auflistung der zulässigen Variablenbelegungen die vorhandenen Speicherkapazitäten des Computers oder die tolerierbare Rechenzeit übersteigt. Aus diesem Grund ist ein Solver, der alle Constraints zunächst in *Table*-Constraints überführt, auf den ersten Blick keine gute Alternative.

Neben der Substituierung aller Constraints durch einzelne *Table*-Constraints besteht auch die Möglichkeit, Mengen von Constraints (Teil-CSPs) durch ein einzelnes *Table*-Constraint zu ersetzen. In der Praxis kann dies oft zu einer Beschleunigung des Lösungsvorganges führen. Für die Substituierung einer Menge von Constraints C' bzw. eines Teil-CSPs $P' = (X', D', C')$ (siehe Definition 3.1) durch ein einzelnes *Table*-Constraint ergeben sich zwei Möglichkeiten:

- Das Erzeugen und anschließende Kombinieren der Tupellisten aller Constraints $c \in C'$ zu einer Tupelliste und darauffolgende Erstellen des zugehörigen *Table*-Constraints.

- Das Lösen des Teil-CSPs $P' = (X', D', C')$ und das anschließende Erstellen eines *Table*-Constraints aus der Auflistung der resultierenden Lösungen.

Variante 1

Bei der ersten Variante werden zunächst zu allen Constraints $c_i = (X_i, R_i) \in C'$ mit $X_i = \{x_1, ..., x_n\}$ die gültigen Tupellisten T_i erstellt. Bei expliziter Darstellung der durch R_i definierten Tupel in c können die Tupellisten T_i gegebenenfalls direkt entnommen werden. Sollten die Tupel allerdings implizit in c_i gegeben sein, so müssen sie zunächst aufgelistet werden. Eine Möglichkeit der Tupelberechnung ist die Behandlung des Constraints c_i als CSP P^{c_i} mit $X^{c_i} = scope(c_i), D^{c_i}, C^{c_i} = \{c\}$, wobei D^{c_i} genau die Domänen der Variablen X^{c_i} umfasst. Das Lösen von P^{c_i} ermittelt alle gültigen Variablenbelegungen $\phi_j \in X^{c_i} \rightarrow D^u$ mit $D^u = \bigcup_{\forall(D_i \in D^{c_i})} D_i$. Jede Variablenbelegung ϕ_j mit $\phi_j(x_1) = d_1, ..., \phi_j(x_n) = d_n$ kann in ein Tupel $t_j = (d_1, ..., d_n)$ überführt werden. Sei T_i gleich die Auflistung all dieser Tupel von Constraint c_i.

Anschließend müssen die so erzeugten Tupellisten T_i miteinander verschmolzen werden, um die Tupelliste T für das neue, C'-umfassende *Table*-Constraint zu erzeugen. Dies kann u.A. mittels *Merge*- oder *Join*-Methoden, wie sie aus dem Bereich der Datenbanken bekannt sind (z.B. der Sort-Merge Join-Algorithmus [164]), realisiert werden. Der *merge*-Vorgang spiegelt dabei einen Lösungsvorgang für die zu substituierende Constraint-Menge C' wieder. Das Constraint $Table(X', T)$ kann dann als Substituierung für die Constraints in C' verwendet werden.

Ausgehend davon, dass die Constraints C' bereits gegeben sind, deren Propagatoren aber noch nicht erstellt wurden, setzt sich der Zeitaufwand $t(\textbf{sub\_v1})$, der für die **Sub**stituierung von C' in ein *Table*-Constraint mit der **V**ariante **1** benötigt wird, aus drei Komponenten zusammen:

- der Summe der Zeitaufwände $t(transform(c_i))$ für das Erstellen der einzelnen Tupellisten T_i[2],
- der Zeitaufwand $t(merge)$ des *merge*-Algorithmus und
- dem Zeitmehr- oder Minderaufwand $t(prop)$ für das Propagieren des *Table*-Constraints anstelle der ursprünglichen Constraints[3].

$$t(sub\_v1) = (\sum_{\forall c_i \in C'} t(transform(c_i)) + t(merge) + t(prop) \tag{4.1}$$

---

[2] Da die Auflistung der Tupel aus einem Constraint c_i unabhängig von allen anderen Constraints $c_j \in C'$, mit $i \neq j$ erfolgt, bietet es sich an, diesen Vorgang zu parallelisieren, um den Zeitaufwand zu reduzieren.

[3] Im Falle eines Minderaufwands hat $t(prop)$ ein negatives Vorzeichen.

Variante 2

Das Hauptproblem, welches sich bei der zweiten Variante ergibt, ist, dass das Teil-CSP P' von P bereits so groß sein kann, dass eine Auflistung der Lösungen von P' nur unwesentlich weniger Zeit in Anspruch nimmt als das Auflisten der Lösungen von P [4]. Der Aufwand für das Lösen von P' muss dabei zusätzlich zum Transformationsaufwand berücksichtigt werden. Eine Beschleunigung des Lösungsvorganges von P durch die Substituierung der Constraints in P' durch ein *Table*-Constraint c^t, das äquivalent zu den Constraints in P' ist, kann also nur dann erfolgen, wenn P' „deutlich" schneller gelöst werden kann als P ($t(solve_{P'}) \ll t(solve_P)$).

Der Zeitaufwand $t(\textbf{sub\_v2})$ für die **Sub**stituierung mit **V**ariante **2** setzt sich aus dem Erstellen $t(create_{P'})$ und Lösen $t(solve_{P'})$ des Teil-CSPs P' sowie dem benötigten Zeitmehr- oder Minderaufwand $t(prop)$ für das Propagieren des *Table*-Constraints anstelle der ursprünglichen Constraints zusammen.

$$t(sub\_v2) = t(create_{P'}) + t(solve_{P'}) + t(prop) \tag{4.2}$$

In seltenen Ausnahmefällen besteht bei beiden Varianten die Möglichkeit, dass das Erstellen der Propagatoren (bzw. der Datenstrukturen, die der Propagator initial benötigt) für die ursprünglichen Constraints $c_i \in C'$ deutlich zeitaufwendiger ist als das Erstellen des *Table*-Propagators. Dadurch kann es vorkommen, dass das Erzeugen des Substituierten CSPs schneller erfolgt als das Erzeugen des ursprünglichen CSPs. Durch die Substituierung von C' kann schon beim Erstellen des CSPs Zeit eingespart werden. Ob ein CSP schneller oder langsamer erstellt werden kann, hat keinen direkten Einfluss auf die Lösungsgeschwindigkeit, der folgende Lösungsvorgang kann dann jedoch mit einem zeitlichen Vorsprung oder Rückstand gegenüber dem ursprünglichen CSP gestartet werden.

Das Beispiel 4.1 verdeutlicht die beiden Varianten an Hand eines konkreten CSPs.

Beispiel 4.1: Möglichkeiten der Tabularisierung[5]**.** *Betrachtet wird das CSP 8, welches durch ein äquivalentes CSP P^t ersetzt werden soll, das nur* Table-*Constraints beinhaltet.*

Durch das explizite Darstellen der Tupel, die durch die Constraints in C repräsentiert werden, können äquivalente Table-*Constraints erzeugt werden. Für c_2 ergibt sich beispielsweise* $c_2^t = \text{Table}(\{x_1, x_2, x_3\}, \{\{1,1,2\}, \{1,2,3\}, \{1,3,4\}, \{1,4,5\}, \{2,1,3\}, \{2,2,4\}, \{2,3,5\}, \{3,1,4\}, \{3,2,5\}, \{4,1,5\}\})$. *Analog können c_1^t, c_3^t, c_4^t und c_5^t gebildet werden, die dann über 20, 98, 8960 bzw. 590 Tupel verfügen.*

---

[4] Ein ähnliches Problem tritt zwar bei der ersten Variante beim Mergen auch auf, allerdings haben Merge-Algorithmen in der Regel im schlechtesten Fall einen quadratischen Aufwand.

[5] Zum Lösen aller im Beispiel erzeugten CSPs wurde jeweils der Choco-Solver mit der in Abschnitt 1.4 angegebenen Suchstrategie, Software und Hardware verwendet.

CSP 8: $P = (X, D, C)$ mit:

$$
\begin{aligned}
&X = \{x_1, ..., x_7\} && \text{(7 Variablen)}\\
&D = \{D_1 = ... = D_7 = \{1, 2, 3, 4, 5\}\} && \text{(7 Domänen)}\\
&C = \{c_1 = (x_1 \neq x_2) \cup\\
&\qquad c_2 = (x_1 + x_2 = x_3) \cup\\
&\qquad c_3 = (x_1 * x_2 > x_3) \cup\\
&\qquad c_4 = Count(X, 1, 3) \cup\\
&\qquad c_5 = (x_4 + x_5 + x_6 + x_7 \geq 8)
\end{aligned}
$$

Sollen an dieser Stelle die Constraints des Teil-CSPs $P' = (X' = \{x_1, x_2, x_3\}, D' = \{D_1, D_2, D_3\}, C' = \{c_1, c_2, c_3\})$ *von P durch ein gemeinsames* Table-*Constraint ersetzt werden, statt durch die drei einzelnen* c_1^t, c_2^t *und* c_3^t, *so können die beiden zuvor vorgestellten Varianten genutzt werden.*

Variante 1

Zunächst werden die Tupellisten T^1, T^2 *und* T^3 *wie zuvor erzeugt. Anschließend werden diese mittels eines Merge-Algorithmus zu einer Tupelliste* T^m *vereinigt. Da* c_2 *die gesamte Variablemenge* X' *von* C' *abdeckt und die zugehörige Tupelmenge* T_2 *nur 10 Tupel enthält, kann auch die kombinierte Tupelmenge* T^m *maximal 10 gültige Tupel aufweisen. Aus den drei einzelnen Tupellisten* T^1, T^2 *und* T^3 *entsteht die verschmolzene Tupelliste* $T^m = \{\{2, 3, 5\}, \{3, 2, 5\}\}$ *mit lediglich zwei Tupeln. Das zugehörige* Table-*Constraint* $c_m^t = (\{x_1, x_2, x_3\}, T^m)$ *dient als äquivalente Substituierung von* c_1 *bis* c_3.

Variante 2

Bei dieser Variante müssen die Tupellisten für c_1 *bis* c_3 *nicht erzeugt und anschließend durch ein merge-Verfahren vereinigt werden. Statt dessen werden die Constraints* c_1 *bis* c_3 *in ein neues CSP* $P' = (X' = \{x_1, x_2, x_3\}, D' = \{D_1, D_2, D_3\}, C' = \{c_1, c_2, c_3\})$ *überführt, welches direkt gelöst werden kann. Aus der Liste der Lösungen ergibt sich dabei die Tupelliste* T^m *für das zu erzeugende* Table-*Constraint* $c_m^t = (\{x_1, x_2, x_3\}, T^m)$.

Neben der ursprünglichen Variante (CSP 8), der Variante, bei der alle Constraints in Table-*Constraints umgewandelt sind und den beiden vorgestellten Varianten 1 und 2 sind auch noch weitere Mischformen möglich. Zur eindeutigen Identifizierung wird folgendes Namensschema eingeführt:*

$$Substituierungsart(\{Gruppe1^{V_1}, Gruppe2^{V_2}, ...\})$$

Wobei die Substituierungsart beschreibt, ob eine Tabularisierung T oder Regularisierung R durchgeführt wird, die Gruppen angeben, welche Mengen von Constraints durch ein gemeinsames Constraint substituiert werden und die V-Werte aufführen, ob die jeweils zugehörige Menge von Constraints mit der ersten oder zweiten vorgestellten Variante substituiert wurde. Bei Constraint-Mengen, die nur ein Constraint enthalten, hat die gewählte Variante (1 oder 2) keinen Einfluss auf den Lösungsprozess, weswegen bei diesen der V-Wert weggelassen wird.

	Variante	$t(create_P)$	$t(sub)$	$t(solve(P^t))$	$t(total)$
1	CSP 8	0,1710	0,0000	0,0160	0,1870
2	$T(\{\{c_1\},\{c_2\},\{c_3\},\{c_4\},\{c_5\}\})$	0,1731	0,1460	0,0089	0,3280
3	$T(\{\{c_1,c_2,c_3\}^1,\{c_4\},\{c_5\}\})$	0,1778	0,1452	0,0068	0,3298
4	$T(\{\{c_1,c_2,c_3\}^2,\{c_4\},\{c_5\}\})$	0,1762	0,1449	0,0072	0,3283
5	$T(\{\{c_1,c_2,c_3\}^1\})$	0,1754	0,0252	0,0012	0,2018
6	$T(\{\{c_1,c_2,c_3\}^2\})$	0,1733	0,0186	0,0014	0,1934
7	$T(\{\{c_1,c_2,c_3\}^1,\{c_4\}\})$	0,1717	0,1385	0,0068	0,3171
8	$T(\{\{c_1,c_2,c_3\}^2,\{c_4\}\})$	0,1729	0,1418	0,0072	0,3220
9	$T(\{\{c_1,c_2,c_3\}^1,\{c_5\}\})$	0,1719	0,0471	0,0016	0,2206
10	$T(\{\{c_1,c_2,c_3\}^2,\{c_5\}\})$	0,1720	0,0420	0,0018	0,2158

Tabelle 4.1: Zeitaufwände für das Erstellen, Transformieren und Lösen des CSPs 8 in verschiedenen Variationen (Teil 1: Tabularisierung).

Das CSP, welches aus CSP 8 durch Tabularisieren jedes einzelnen Constraints entsteht, hat somit die Bezeichnung $T(\{\{c_1\},\{c_2\},\{c_3\},\{c_4\},\{c_5\}\})$. *Das CSP, welches sich durch Tabularisierung der Constraint-Menge* $\{c_1,c_2,c_3\}$ *durch ein* Table-*Constraint nach Variante 1 und einzelner Tabularisierung der Constraints* c_4 *und* c_5 *ergibt, hat die Bezeichnung* $T(\{\{c_1,c_2,c_3\}^1,\{c_4\},\{c_5\}\})$. *Das CSP, welches durch Tabularisierung der Constraint-Menge* $\{c_1,c_2,c_3\}$ *durch ein* Table-*Constraint nach Variante 2, einzelner Tabularisierung des Constraints* c_4 *und Beibehalten des ursprünglichen Constraints* c_5 *entsteht, hat die Bezeichnung* $T(\{\{c_1,c_2,c_3\}^2,\{c_4\}\})$.

In Tabelle 4.1 ist der durchschnittliche Zeitaufwand für das Erstellen $t(create_P)$, *Transformieren* $t(sub)$ *und Lösen* $t(solve_{P^t})$ *sowie deren akkumulierte Zeit für das CSP 8 und verschiedene Tabularisierungen davon bei 1000 Durchläufen dargestellt. Dabei umfasst* $t(create_P)$ *die benötigte Zeit für das Anlegen der Variablen und Constraints ohne deren Propagatoren. Die Zeit wäre in einem perfektem Testszenario für alle 10 verschiedenen Substituierungen von CSP 8 gleich. In diesem Fall ist sie ein Maß für die Ungenauigkeit beim Versuch, die exakte Zeit für die Ausführung eines Programmes zu ermitteln (Störfaktoren sind z.B. nebenläufige Prozesse, Prozessorwechsel etc.). Der Zeitmehraufwand für die Substituierung* $t(sub)$ *entspricht abhängig von der verwendeten Variante* $t(sub\_v1)$ *(siehe Gleichung 4.1) bzw.* $t(sub\_v2)$ *(siehe Gleichung 4.2). Bei der ursprünglichen Variante (Zeile 1) ist* $t(sub)$ *null, da keine Substituierung durchgeführt wurde. In* $t(solve(P^t))$ *wird die Zeit berücksichtigt, die benötigt wird, um das transformierte CSP* P^t *zu lösen. Bei der ursprünglichen Variante ist statt dessen die Zeit angegeben, die erforderlich ist, um das ursprüngliche CSP 8 zu lösen.*

Beim Betrachten der Tabelle fällt auf, dass die meiste Zeit nicht für das Lösen, sondern für das Erstellen des CSPs ohne Propagatoren ($t(create_P)$*) benötigt wird, welche für alle Probleme annähernd gleich ist. Diese Komponente würde bei komplexeren Problemen allerdings einen deutlich geringeren bis hin zu vernachlässigenden prozentualen Anteil ausmachen. Die benötigte Zeit für die Substituierung (*$t(sub)$*) sowie die eigentliche Lösungszeit (*$t(solve(P^t))$*) differieren* *sehr stark in Abhängigkeit von der jeweiligen Modellierungsvariante. Es können die folgenden Beobachtungen festgestellt werden:*

- *Die zwei zuvor vorgestellten Varianten für die Substituierung von Teil-CSPs sind annähernd gleich schnell (Vergleich Zeile 3 mit 4, 5 mit 6 etc. in Tabelle 4.1).*
- *Das* Count-*Constraint zu transformieren ist sehr zeitaufwendig (Vergleich von $t(sub)$ der Zeilen 5 und 6 mit den Zeilen 7 und 8).*
- *Das* Count-*Constraint zu tabluarisieren verlangsamt die Lösungsgeschwindigkeit (Vergleich der $t(solve(P^t))$ Werte der Zeilen 5 und 6 mit den Zeilen 7 und 8).*
- *Auch das* Sum-*Constraint zu tabluarisieren verlangsamt die Lösungsgeschwindigkeit (Vergleich der $t(solve(P^t))$ Werte der Zeilen 5 und 6 mit den Zeilen 9 und 10).*
- *Alle Varianten mit Tabluarisierungen können schneller gelöst werden als das ursprüngliche CSP, betrachtet auf die reine Lösungszeit $t(solve(P^t))$ (immer um mindestens 40% reduzierte Lösungszeit).*
- *Im Falle von $T(\{\{c_1, c_2, c_3\}^1\})$ und $T(\{\{c_1, c_2, c_3\}^2\})$ konnte die reine Lösungszeit mittels Tabularisierung um mehr als 90% reduziert werden (siehe Zeilen 5 und 6).*

Aus diesen Beobachtungen lassen sich die folgenden Schlussfolgerungen ziehen:

- *Tabularisierungen sind zeitaufwendig können aber anschließend zu einer Verringerung der Lösungszeit führen.*
- *Die Tabularisierung bestimmter Constraints (hier des* Count-*Constraints und des* Sum-*Constraints) kann zu einer Verlangsamung der Lösungszeit $t(solve(P^t))$ führen.*

Wann eine Tabularisierung sinnvoll ist und welche der beiden Varianten oder gar eine Mischform beider schneller ist, lässt sich zu diesem Zeitpunkt nicht sicher vorhersagen und muss erst noch genauer untersucht werden. Mögliche Einflussfaktoren könnten allerdings die Größe des Suchraumes des Teil-CSPs und die Anzahl der repräsentierten Tupel in den einzelnen Constraints sein.

4.1.2 Das *Regular*-Constraint

Das *Regular*-Constraint erhält, wie in Abschnitt 2.5 beschrieben, eine geordnete Menge von Variablen $X = \{x_1, ..., x_n\}$ und einen deterministischen endlichen Automaten (bzw. im Kontext dieser Arbeit einen levelbasierten DFA) M als Eingabe und erlaubt nur Variablenbelegungen $\phi \in X' \rightarrow \Sigma$ mit $\phi(x_1) = d_1, ..., \phi(x_n) = d_n$, die von M akzeptierte Wörter $d_1d_2...d_n$ repräsentieren.

Da die Anzahl der gültigen Belegungen eines FD-Constraints endlich ist, können diese durch eine reguläre Sprache ausgedrückt werden (zum Beispiel durch Angabe aller Elemente der Sprache). Jede reguläre Sprache kann wiederum durch einen deterministischen endlichen Automaten ausgedrückt werden (Satz von Myhill-Nerode [77]).

Das Auflisten aller gültigen Belegungen eines Constraints und das anschließende Erstellen eines deterministischen endlichen Automaten (DFA) bzw. eines levelbasierten DFAs, der diese Belegungen widerspiegelt, ist eine Möglichkeit Constraints in *Regular*-Constraints zu überführen. Dieser Ansatz ist analog zum Vorgehen bei der Substituierung mittels *Table*-Constraints und erzeugt die gleichen Probleme. Das Auflisten der gültigen Belegungen (Tupel) eines Constraints kann bei impliziter, kompakter Darstellung im ursprünglichen Constraint c sehr zeit- und ressourcenaufwendig sein. Ein weiterer zu berücksichtigender Aufwand ist der, der durch das Minimieren des erzeugten Automaten entsteht. Eine Minimierung sollte immer vor der Übergabe an den Propagator erfolgen, da ein nicht-minimaler levelbasierter DFA die Propagation erheblich verlangsamen kann.

Bei der Regularisierung eines Teil-CSPs $P' = (X', D', C')$ kann analog zu den beiden Varianten der Tabularisierung vorgegangen werden.

Variante 1: Umwandeln der Relationen aller Constraints $c_1, ..., c_m \in C'$ in Automaten $M_1, ..., M_m$, Bilden eines Schnittmengenautomaten $M' = M_1 \cap ... \cap M_m$ (z.B. wie in [78] beschrieben) und anschließendes Erstellen des Constraints $Regular(X', M')$.

Variante 2: Das Lösen des Teil-CSPs $P' = (X', D', C')$ und das anschließende Erstellen eines Automaten M' aus den Lösungen von P' gefolgt vom Erstellen des Constraints $Regular(X', M')$.

Der Zeitaufwand, der durch die Transformation zustande kommt, kann ebenfalls durch die bereits aus der Tabularisierung bekannten Gleichungen 4.1 für Variante 1 und 4.2 für Variante 2 beschrieben werden.

Das folgende Beispiel greift das CSP 8 aus dem vorherigen Abschnitt wieder auf und zeigt seine Remodellierung mit *Regular*-Constraints.

Beispiel 4.2: Möglichkeiten der Regularisierung[6]**.** *Betrachtet wird das bereits bekannte CSP 8, welches durch ein äquivalentes CSP P^r ersetzt werden soll, das nur* Regular-*Constraints enthält.*

Bei jedem Constraint $c = (X, R) \in C$ entspricht die explizite Tupeldarstellung $T = \{t_1 ..., t_k\}$ der Relation R der Liste der zu akzeptierenden Wörter eines Automaten M. Dieser kann durch Aufzählen der Wörter und anschließendes Minimieren erzeugt werden. Mit diesem Vorgehen können äquivalente Regular-*Constraints für c_1 bis c_5 erzeugt werden. Für $c_2 = (x_1 + x_2 = x_3)$ ergibt sich beispielsweise $c_2^r = \text{Regular}(\{x_1, x_2, x_3\}, M_2)$, mit einem Automaten M_2, wie er in Abbildung 4.1 dargestellt ist. Der dargestellte Automat ist bereits minimal und repräsentiert*

---

[6] Zum Lösen aller im Beispiel erzeugten CSPs wurde jeweils der Choco-Solver mit der in Abschnitt 1.4 angegebenen Suchstrategie, Software und Hardware verwendet.

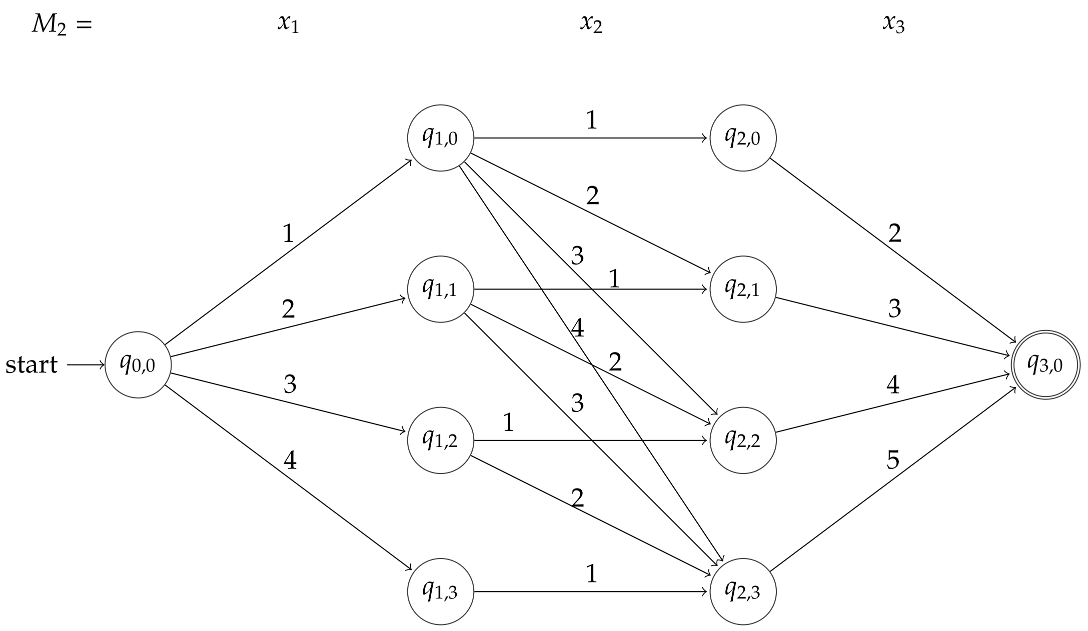


Abbildung 4.1: Der Automat M_2, der in c_2^r verwendet werden kann, um c_2 zu ersetzen.

genau das Constraint c_2 bzw. die Tupel von c_2^t. Dafür werden 10 Knoten und 18 Übergänge benötigt.

Analog können c_1^r, c_3^r, c_4^r und c_5^r gebildet werden, die dann über minimale Automaten mit 7 Knoten und 25 Übergängen, 12 Knoten und 44 Übergängen, 20 Knoten und 79 Übergängen bzw. 17 Knoten und 70 Übergängen verfügen.

Soll an dieser Stelle das Teil-CSP $P' = (X' = \{x_1, x_2, x_3\}, D' = \{D_1, D_2, D_3\}, C' = \{c_1, c_2, c_3\})$ von P als Ganzes durch ein einzelnes Regular-*Constraint ersetzt werden, so können analog zur Tabularisierung die beiden zuvor vorgestellten Varianten genutzt werden.*

Variante 1

Zunächst werden die Automaten M^1, M^2 und M^3 wie zuvor erzeugt. Anschließend werden diese mittels Automaten-Schnittbildung zu M' kombiniert und minimiert. In Abbildung 4.2 ist M' dargestellt. Das Constraint $c_m^r = \text{Regular}(\{x_1, x_2, x_3\}, M')$ dient als äquivalente Substituierung von c_1, c_2 und c_3.

Variante 2

Bei dieser Variante müssen die Automaten für c_1 bis c_3 nicht erzeugt und anschließend vereinigt werden. Statt dessen werden die Constraints c_1 bis c_3 in ein neues CSP $P' = (X' = \{x_1, x_2, x_3\}, D' = \{D_1, D_2, D_3\}, C' = \{c_1, c_2, c_3\})$ überführt, welches anschließend gelöst wird. Die Liste der Lösungen des Teil-CSPs P' kann unter Berücksichtigung der Variablenreihen-

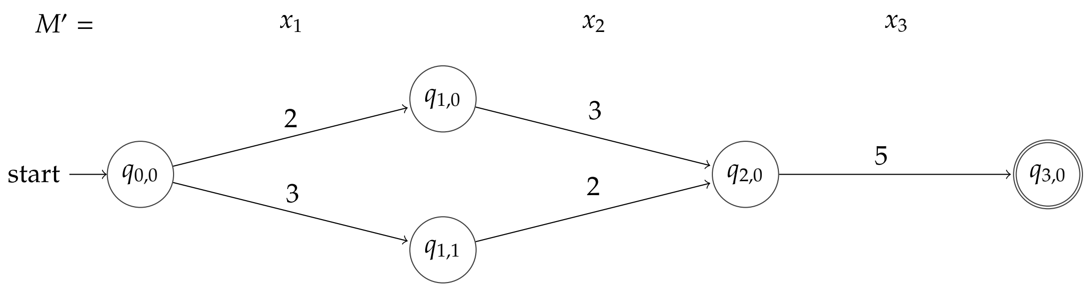


Abbildung 4.2: Der Automat M', der in c_m^r verwendet werden kann, um c_1, c_2 und c_3 zu ersetzen.

	Variante	$t(create_P)$	$t(sub)$	$t(solve(P^t))$	$t(total)$
1	Original	0,1710	0,0000	0,0160	0,1870
2	$R(\{\{c_1\},\{c_2\},\{c_3\},\{c_4\},\{c_5\}\})$	0,1743	0,4066	0,0039	0,5848
3	$R(\{\{c_1,c_2,c_3\}^1,\{c_4\},\{c_5\}\})$	0,1756	0,3901	0,0029	0,5686
4	$R(\{\{c_1,c_2,c_3\}^2,\{c_4\},\{c_5\}\})$	0,1719	0,3783	0,0028	0,5530
5	$R(\{\{c_1,c_2,c_3\}^1\})$	0,1755	0,0356	0,0014	0,2125
6	$R(\{\{c_1,c_2,c_3\}^2\})$	0,1776	0,0343	0,0016	0,2135
7	$R(\{\{c_1,c_2,c_3\}^1,\{c_4\}\})$	0,1754	0,3758	0,0023	0,5535
8	$R(\{\{c_1,c_2,c_3\}^2,\{c_4\}\})$	0,1763	0,3681	0,0023	0,5467
9	$R(\{\{c_1,c_2,c_3\}^1,\{c_5\}\})$	0,1749	0,0714	0,0031	0,2493
10	$R(\{\{c_1,c_2,c_3\}^2,\{c_5\}\})$	0,1751	0,0682	0,0031	0,2463

Tabelle 4.2: Zeitaufwände für das Erstellen, Transformieren und Lösen des CSPs 8 in verschiedenen Variationen (Teil 2: Regularisierung).

folge x_1, x_2, x_3 in eine Liste von Wörtern umgewandelt werden, die von M' akzeptiert werden sollen. Aus dieser Wortliste wird der Automat M' erzeugt, der anschließend minimiert wird. Der minimierte Automat entspricht dem in Abbildung 4.2 dargestellten. Das Constraint $c_m^r = \text{Regular}(\{x_1, x_2, x_3\}, M')$ *dient als äquivalente Substituierung von c_1 bis c_3.*

In Tabelle 4.2 sind die durchschnittlichen Zeitaufwände für das Erstellen $t(create_P)$, Transformieren $t(sub)$ und Lösen $t(solve(P^t))$ sowie die akkumulierte Zeit für verschiedene Variationen von CSP 8 bei 1000 Durchläufen dargestellt, wobei im Gegensatz zu Tabelle 4.1 an dieser Stelle Regularisierung verwendet wurde. Abgesehen vom Austausch der Tabularisierung durch die Regularisierung sind die Problemvariationen die gleichen wie in Beispiel 4.1.

Es können die folgenden Beobachtungen gemacht werden:

- *Beide Varianten der Regularisierung, für das Substituieren von Teil-CSPs, sind in allen Fällen annähernd gleich schnell (z.B. Vergleich der Zeilen 3 und 4 oder 5 und 6).*

- *Alle Regularisierungen können, bezogen auf die reine Lösungszeit $t(solve(P^t))$, schneller gelöst werden als das ursprüngliche CSP (Vergleich Zeile 1 mit allen anderen).*
- *Die Transformation des* Count-*Constraints ist sehr zeitaufwendig (siehe $t(sub)$ für Zeile 7 und 8 im Vergleich zu Zeile 5 und 6).*
- *Die Regularisierung des* Count-*Constraints hat einen negativen Einfluss auf die reine Lösungszeit $t(solve(P^t))$ (Vergleich von Zeile 5 und 6 mit Zeile 7 und 8).*
- *Auch die Regularisierung des* Sum-*Constraints hat einen negativen Einfluss auf die reine Lösungszeit $t(solve(P^t))$ (Vergleich von Zeile 5 und 6 mit Zeile 9 und 10).*
- *Die Regularisierung ist für dieses Beispiel, betrachtet auf die Gesamtzeit $t(total)$, immer langsamer als die Tabularisierung.*
- *Bei der Regularisierung konnte die reine Lösungszeit $t(solve(P^T))$ immer mindestens um 75% reduziert werden (bei der Tabularisierung waren es nur um mindestens 40%).*
- *Im Falle von $R(\{\{c_1, c_2, c_3\}^1\})$ und $R(\{\{c_1, c_2, c_3\}^2\})$ konnte die reine Lösungszeit mittels Regularisierung um ca. 90% reduziert werden.*

Es können grundsätzlich ähnliche Schlussfolgerungen wie bei der Tabularisierung getroffen werden. Die aufgeführten Zeitaufwände könnten vermuten lassen, dass das Umwandeln in Regular-*Constraints immer schlechter ist als das Umwandeln in* Table-*Constraints. Dies wird im Laufe der Arbeit allerdings widerlegt werden. Ein Anzeichen dafür, dass das Umwandeln in* Regular-*Constraints ein hohes Potential liefert, gibt ein Blick auf die Transformation des* Count-*Constraints. Werden die beiden Varianten $R(\{\{c_1, c_2, c_3\}^1, \{c_4\}\})$ (siehe Tabelle 4.2 Zeile 7) und $T(\{\{c_1, c_2, c_3\}^1, \{c_4\}\})$ (siehe Tabelle 4.1 Zeile 7) miteinander verglichen, so ist zwar die benötigte Transformationszeit und damit verbunden die Gesamtzeit bei der Regularisierung deutlich größer als bei der Tabularisierung, allerdings hat sich die reine Lösungszeit bei der Regularisierung im Vergleich zur Tabularisierung mehr als halbiert. Dies lässt die Vermutung zu, dass das* Regular-*Constraint bei größeren Problemen, bei denen das Lösen mehr Zeit in Anspruch nimmt als das Transformieren, zu einer schnelleren Gesamtauswertung führen kann.*

Das Beispiel 4.2 veranschaulicht, dass die Regularisierung zu einer Beschleunigung des eigentlichen Lösungsverfahren führen kann, dafür aber eine erhöhte Transformationszeit berücksichtigt werden muss. Im Gegensatz zur Tabularisierung lässt die Regularisierung allerdings noch eine weitere Möglichkeit der Automatenerzeugung zu, die die Transformationszeit erheblich verringern kann. Es besteht die Möglichkeit, Meta-Informationen von Constraints wie deren Struktur, Eigenschaften oder Semantik auszunutzen, um einen minimalen Automaten daraus zu erzeugen, ohne vorher deren Tupel explizit auflisten zu müssen. Diese Möglichkeit kann zu einer erheblichen Zeitersparnis führen und ist einer der entscheidenden Vorteile der Regularisierung gegenüber der Tabularisierung. Auf die verschiedenen Varianten der Transformation von Constraints in *Regular*-Constraints wird in Kapitel 5 gesondert eingegangen.

4.2 Der Nutzen von FD-vollständigen Constraints

Die beiden Beispiele 4.1 und 4.2 aus den vorherigen Abschnitten zeigen, wie FD-vollständige Constraints andere Constraints ersetzen können. Dabei wurde erkennbar, dass verschiedene Constraints unterschiedlich gut für die Substituierung anderer Constraints geeignet sein können.

So ist das Verwenden einer Datenstruktur, die nur 10 Tupel speichern muss (c_2^t aus Beispiel 4.1), kompakter als das Speichern eines Graphen mit 10 Knoten und 18 Übergängen (c_2^r aus Beispiel 4.2). Im Gegenzug erscheint die Substituierung von c_4 durch ein *Regular*-Constraint wesentlich kompakter (20 Knoten und 79 Übergänge) als durch ein *Table*-Constraint, bei dem alle 8960 gültigen Belegungen als eigenständige Tupel angelegt werden müssen.

Die folgenden Erläuterungen basieren auf den im Zusammenhang mit dieser Arbeit entstandenen Veröffentlichungen [95, 100] und wurden für diese Arbeit erweitert.

4.2.1 Erreichen eines höheren Konsistenzniveaus

Durch das Substituieren von Constraints durch FD-vollständige Constraints kann ein höheres Konsistenzniveau erreicht werden. Dies kann sowohl beim Substituieren eines einzelnen Constraints als auch beim Substituieren einer Menge von Constraints durch ein FD-vollständiges Constraint erfolgen.

Die Substituierung einzelner Constraints

Constraints mit einem geringeren Konsistenzniveau wie z.B. das *Sum*-Constraint, welches in den meisten Implementierungen [3, 42, 129] nur Grenzenkonsistenz erreicht, können durch Constraints mit einem höheren Konsistenzniveau, wie z.B. dem *Table*- oder dem *Regular*-Constraint (lokale Konsistenz), ersetzt werden. Es wird davon ausgegangen, dass sich bei höherem Konsistenzniveau aber ansonsten gleichbleibender Problembeschreibung die Anzahl der Knoten im Suchbaum verkleinert oder gleich bleibt. Eine Heuristik, die diese Idee nutzt, ist die in Abschnitt 3.3.1 vorgestellte schwache Propagations-Heuristik aus [10].

Auch bei dem zuvor bereits angesprochenen *Global Cardinality*-Constraint mit Vorkommen doppelter Variablen kann das Konsistenzniveau durch Substituierung erhöht werden. Da das Erreichen lokaler Konsistenz eines solchen Constraints NP-vollständig ist [33], wird in diesem Fall auf ein schwächeres Konsistenzniveau zurückgegriffen. Eine Substituierung eines *Global Cardinality*-Constraints mit Vorkommen doppelter Variablen durch ein FD-vollständiges Constraint, wie dem *Table*- oder dem *Regular*-Constraint, kann ohne Vorkommen doppelter Variablen erfolgen und somit lokale Konsistenz erreichen.

Anmerkung 1. Auch bei gleichbleibendem Konsistenzniveau ist es möglich, dass eine Ersetzung eines Constraints durch ein anderes zu einer Verbesserung (oder Verschlechterung) der Lösungsgeschwindigkeit führt. Dies erklärt sich mit den unterschiedlichen Algorithmen und Datenstrukturen, die in den verschiedenen Propagatoren verwendet werden. Die richtige Vorhersage, welches Constraint in welcher Situation am schnellsten zu einer Lösung führt, ohne vorherige Berechnung aller Möglichkeiten, stellt dabei ein eigenes Forschungsproblem dar.

Die Substituierung von Constraint-Mengen

Bei der Substituierung einer Menge von Constraints C durch ein FD-vollständiges Constraint c^{fd} kann zusätzlich das Konsistenzniveau der Constraints von C als Ganzes erhöht werden. Das folgende CSP 9 gibt ein Beispiel für die Erhöhung des Konsistenzniveaus der Constraints c_1 bis c_3. Es wird dabei davon ausgegangen, dass die Propagatoren für c_1 bis c_3 lokale Konsistenz erzeugen.

CSP 9: $P = (X, D, C)$ mit:
$X = \{x_1, x_2, x_3\}$
$D = \{D_1 = \{0, 1\}, D_2 = \{0, 1\}, D_3 = \{0, 1\}\}$
$C = \{c_1 = (x_1 \neq x_2), c_2 = (x_1 \neq x_3), c_3 = (x_2 \neq x_3)$

Obwohl alle Constraints in C initial lokal konsistent sind, verfügt das CSP P über keine Lösung. Die Constraints in C sind zusammen betrachtet also nicht-konsistent. Werden die Constraints C durch ein FD-vollständiges Constraint c^{fd} mit einem Propagator, der lokale Konsistenz herstellt, ersetzt, so kann durch einmaliges Propagieren festgestellt werden, dass keine Lösung für P existiert. Somit kann durch die Substituierung von c_1 bis c_3 mit dem Constraint c^{fd} das Konsistenzniveau des ganzen CSPs angehoben werden.

Lemma 4.1. *Besitzt ein CSP P mit Constraints, deren Propagatoren ausschließlich lokale Konsistenz erzeugen, keine* shared variables *(geteilten Variablen), so ist das CSP ohne Vorkommen von fehlerhaften Zuständen lösbar (sofern eine Lösung existiert). Fehlerhafte Zustände sind dabei Knoten im Suchbaum von P, in denen mindestens eine Domäne leer ist.*

Definition 4.2: *Shared Variables.* Die Schnittmenge der Variablen zweier Constraints $c_1 = (X_1, R_1)$ und $c_2 = (X_2, R_2)$ wird als Menge der shared variables bezeichnet.

$$sharedVariables(c_1 = (X_1, R_1), c_2 = (X_2, R_2)) = X_1 \cap X_2 \tag{4.3}$$

Beweis 4.1. *Es wird zunächst gezeigt, dass ein CSP P mit nur einem Constraint c, dessen Propagator lokale Konsistenz erzeugt, in keinen fehlerhaften Zustand gelangt. Anschließend wird erläutert, dass ein CSP mit mehreren Constraints aber ohne* shared variables *ebenfalls in keinen fehlerhaften Zustand gelangen kann.*

Widerspruchsbeweis:

1. *Angenommen q_i ist ein fehlerhafter Zustand, der durch die Zuordnung der Variablen x mit dem Wert d erreicht wurde. Da c das einzige Constraint in P ist, kann die Zuordnung nur dieses Constraint verletzen. Da der Propagator von c allerdings in jedem Schritt, also auch in allen vorherigen Zuständen $q_0, ..., q_{i-1}$, lokale Konsistenz hergestellt hat, kann ein solcher Wert d für x per Definition von lokaler Konsistenz nicht mehr in der Domäne von $D(x)$ enthalten gewesen sein.* ↯

2. *Jedes CSP $P = (X = \{x_1, ..., x_n\}, D = \{D_1, ..., D_n\}, C = \{c_1, ..., c_m\})$ mit $m \geq 1$ vielen Constraints, das keine* shared variables *zwischen den einzelnen Constraints besitzt, kann in m viele CSPs $P_i = (scope(c_i), D^i, \{c_i\}), \forall i \in \{1, ..., m\}$ aufgeteilt werden, wobei D^idie Menge der Domänen der Variablen in $scope(c_i)$ ist. Die CSPs können nach 1. separat und ohne Backtracking gelöst werden. Das Kreuzprodukt der Lösungen der einzelnen CSPs stellt die Lösung des ursprünglichen CSPs dar.* □

Eine Hypothese, die sich nach Lemma 4.1 aufstellen lässt, ist die, dass CSPs mit weniger Überschneidungen (weniger *shared variables* über allen Constraints) im Durchschnitt weniger Backtracking benötigen. Dies lässt sich an den verschiedenen Variationen von CSP 8 beim Lösen mit dem CHOCO Solver bei Standardeinstellungen beobachten. Ohne Substituierung ist die Anzahl der *shared variables* gleich 21 und die Anzahl der fehlerhaften Zustände bei der Suche nach allen Lösungen gleich 1. In den Substituierungen, in denen die Constraints c_1, c_2 und c_3 mit einem gemeinsamen *Table*- oder *Regular*-Constraint substituiert wurden, ist die Anzahl der *shared variables* auf 10 und die Anzahl der fehlerhaften Zustände auf 0 gesunken. In den Tabellen 4.1 und 4.2 ist erkennbar, dass die angesprochenen Substituierungen, welche über lediglich 10 *shared variables* verfügen (jeweils Zeilen 3 bis 10), eine geringere Lösungszeit $t(solve(P^t))$ aufweisen als die mit 21 *shared variables* (jeweils die Zeilen 1 und 2).

4.2.2 Verringerung der Anzahl der Propagationsaufrufe

Verfügt eine Menge von Constraints C mit $|C| > 1$ über *shared variables*, so besteht die Möglichkeit, dass die zu den Constraints gehörenden Propagatoren zyklisch aufgerufen werden, was einen verlangsamenden Einfluss auf den Lösungsprozess hat.

Durch Substituierung einer Constraint-Menge C mit *shared variables* durch ein einzelnes Constraint c reduziert sich die Anzahl der Constraints, damit die Anzahl der Propagatoren, die sich überlagern, und somit auch die Anzahl der Propagatoren, die sich zyklisch aufrufen können.

Betrachtet wird an dieser Stelle ein Teil-CSP $P' = (X, D, C)$, mit $X = \{x_1, ..., x_n\}, D = \{D_1, ..., D_n\}$ und $C = \{c_1, ..., c_m\}$, wobei alle Constraints in C alle Variablen in X abdecken ($scope(c_i) = X, \forall c_i \in C$). Seien des Weiteren $P = \{p_1, ..., p_m\}$ die zu $c_1, ..., c_m$ zugehörigen Propagatoren. Durch Einflüsse von außen (z.B. die Suche oder das Ausführen von

Propagatoren, die zu Constraints gehören, die *sharedVariables* mit dem Teil-CSP haben) wird ein Wert aus einer Domäne D_i entfernt.

Infolge dieser Domänenänderung müssen alle Propagatoren in P aufgerufen werden. Sollte keiner der Propagatoren einen Wert ausschließen können, so mussten zuvor trotzdem alle m Propagatoren einmal aufgerufen werden.

Im schlechtesten verbleibenden Fall entfernt der letzte aufgerufene Propagator einen einzelnen Wert aus einer Domäne D_i. Ohne Beschränkung der Allgemeinheit sei $p \in P$ dieser letzte Propagator. Da alle Propagatoren $p' \in P, p' \neq p$ die Variable x_i, deren Domäne geändert wurde, enthalten, müssen diese Propagatoren nun erneut aufgerufen werden. Wird dabei keine Domäne weiter eingeschränkt, so wurden insgesamt $m + (m-1)$ viele Propagatoren aufgerufen (p wurde einmal aufgerufen und die anderen $(m-1)$ Propagatoren zweimal).

Im wiederum verbleibenden schlechtesten Fall wird erneut ein einzelner Wert aus einer Domäne $D_{i'}$ vom letzten aufgerufenen Propagator entfernt. Dieser Prozess lässt sich fortsetzen, und hat zur Folge, dass im schlechtesten Fall $m + (m-1) * k$ viele Propagatorenaufrufe benötigt werden um k Werte aus den Domänen D zu entfernen. Daraus ergibt sich, dass bis zu $k = (\sum_{D_i \in D} |D_i| - 1) + 1$ viele Domänenentfernungen notwendig sein können, um festzustellen, dass eine Domäne leer wird. Bei dieser Konstellation werden dafür $m + (m-1) * ((\sum_{D_i \in D} |D_i| - 1) + 1)$ viele Propagatorenaufrufe benötigt.

Werden die Propagatoren $p \in P$ allerdings durch einen einzelnen zu P äquivalenten Propagator p' ersetzt, der lokale Konsistenz erzeugt, dann wird lediglich ein Aufruf dieses Propagators benötigt um eine Inkonsistenz oder alle aus den Domönen D zu entfernenden Werte zu ermitteln. In Beispiel 4.3 wird ein zyklisches Propagieren veranschaulicht, das durch Substituierung mit einem FD-vollständigen Constraint vermieden werden kann.

Beispiel 4.3: Das Überlagern von Propagatoren. *Gegeben sei das CSP 10 mit drei Propagatoren* p_1, p_2 *und* p_3*, die jeweils für die einzelnen Constraints* c_1, c_2 *und* c_3 *lokale Konsistenz erzeugen. Das Constraint* c_3 *beschreibt die ganzzahlige Division ohne Rest. Die Variablenbelegung* ϕ *mit* $\phi(x_1) = 1, \phi(x_2) = 2$ *und* $\phi(x_3) = 3$ *ist eine gültige Belegung für* c_3*.*

CSP 10: $P = (X, D, C)$ mit:
$X = \{x_1, x_2, x_3\}$ (3 Variablen)
$D = \{D_1 = \{1,2,3\}, D_2 = \{2,3,4\}, D_3 = \{3,4,5,6,7\}\}$ (3 Domänen)
$C = \{c_1 = (x_1 < x_2), c_2 = (x_2 < x_3), c_3 = (x_3 \; div \; x_2 = x_1)$ (3 Constraints)

Das CSP 10 ist bereits lokal konsistent, soll an dieser Stelle allerdings ein Teil-CSP eines größeren, nicht weiter beschriebenen CSPs P^{big} *darstellen. Der Lösungsvorgang von* P^{big} *könnte zum Beispiel die Werteeliminierung* $1 \notin D_1$ *zur Folge haben, die den Propagationsvorgang von CSP 10 startet. In Abbildung 4.3 ist eine mögliche Propagationsfolge dargestellt. In jedem Block sind die Variablen mit ihren Domänen dargestellt sowie darunter eine Liste der Propagatoren, die*

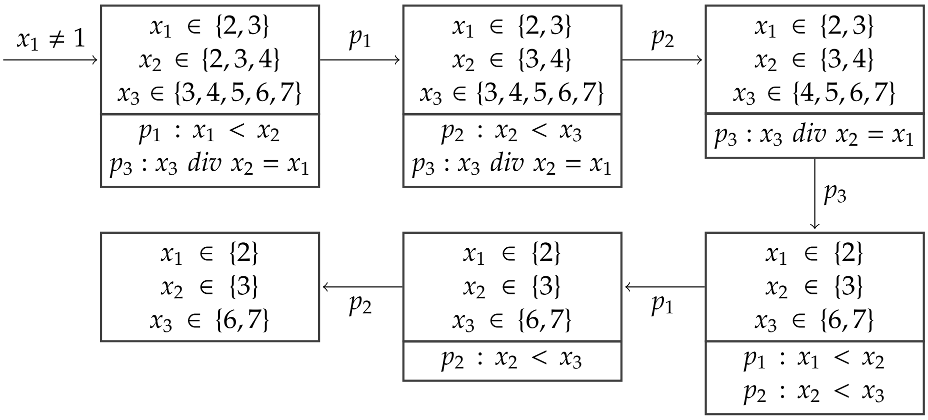


Abbildung 4.3: Eine mögliche Propagationsfolge für CSP 10.

noch zu propagieren sind. Als Propagationsreihenfolge wurde dabei die Indexreihenfolge der zugehörigen Constraints gewählt.

Durch die Eliminierung des Wertes 1 aus der Domäne von x_1 *müssen alle Propagatoren aufgerufen werden, deren Constraints diese Variable enthalten (*p_1 *und* p_3*). Der Propagator* p_1 *wird demzufolge als Erstes aufgerufen und eliminiert den Wert 2 aus der Domäne* D_2*. Daher müssen anschließend die Propagatoren* p_2 *und* p_3 *noch ausgeführt werden. Dieses Vorgehen setzt sich fort, bis der Fixpunkt* $x_1 \in \{2\}, x_2 \in \{3\}, x_3 \in \{6,7\}$ *erreicht wurde. Um diesen zu erreichen, musste zwei mal der Propagator* p_1*, zweimal der Propagator* p_2 *und einmal der Propagator* p_3 *aufgerufen werden.*

Würden die Constraints c_1 *bis* c_3*, statt diese einzeln auszuwerten, durch ein gemeinsames Constraint* c *substituiert werden, welches über einen Propagator verfügt, der lokale Konsistenz herstellt, so würde eine einzige Propagation genügen. In Abbildung 4.4 sind die Tupel eines* Table-*Constraints* c^t *und der Automat eines* Regular-*Constraints* c^r *dargestellt, die als Substituierung für die Constraints* c_1 *bis* c_3 *möglich sind. Im Vergleich zu* c_1 *bis* c_3 *müssen die Propagatoren von* c^t *und* c^r *nur einmal aufgerufen werden, um zu den gleichen Domänenausschlüssen zu gelangen, wie sie in Abbildung 4.3 dargestellt sind.*

Das Beispiel zeigt, dass die Propagationsdauer sich überlagernder Constraints C (in diesem Fall c_1, c_2 und c_3) mittels Substituierung mit einem gemeinsamen Constraint c^{fd} reduziert werden kann. Diese Reduktion kann selbst dann auftreten, wenn die Propagationsdauer $t_{prop}(c^{\text{fd}})$ von c^{fd} größer ist als die Summe der Propagationsdauern $\sum_{c \in C} t_{prop}(c)$ der einzelnen Constraints in C. In dem konkreten Beispiel ist das genau dann der Fall, wenn die Propagationsdauer $t_{prop}(c^{\text{fd}})$ von c^{fd} geringer ist als die Summe

$$T = \{\{1,2,3\},\ \{1,3,4\},\ \{1,3,5\},\ \{1,4,5\},\ \{1,4,6\},\\ \{1,4,7\},\ \{2,3,6\},\ \{2,3,7\}\}$$

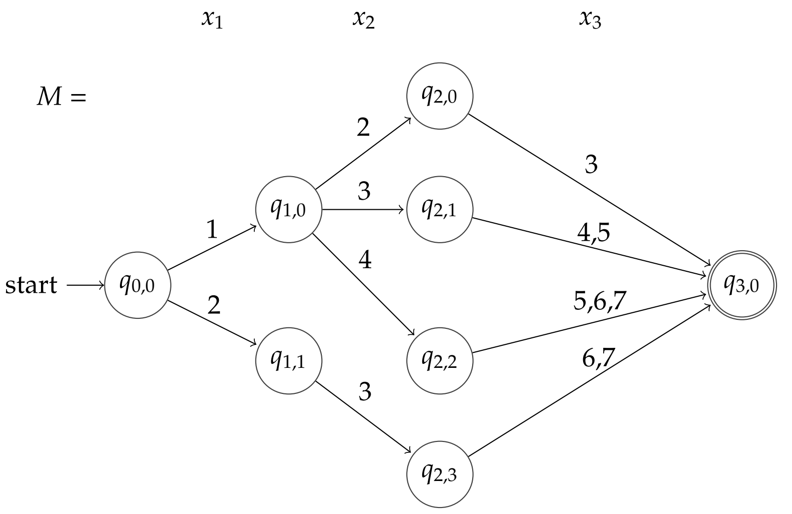


Abbildung 4.4: Die Tupelliste eines *Table*-Constraints und der Automat eines *Regular*-Constraints, die als Substituierung für die Constraints von CSP 10 dienen können.

der Propagationsdauern für das doppelte Ausführen von p_1 und p_2 plus das einmalige Ausführen von p_3 [7].

Des Weiteren kann bei CSP 8 beobachtet werden, dass sich die Anzahl der Propagationsaufrufe durch das Substituieren der Constraints c_1 bis c_3 verringert. Werden in der ursprünglichen Variante von CSP 8 noch 94 Propagationsaufrufe benötigt um alle Lösungen zu finden, so sind es bei den Varianten, in denen die Constraints c_1, c_2 und c_3 durch ein gemeinsames FD-vollständiges Constraint substituiert wurden (jeweils die Zeilen 3-10 in Tabelle 4.1 und 4.2), lediglich 50 Propagationsaufrufe.

Kriterien für eine präzise und schnelle Vorhersage der Propagationsdauer sind gegenwärtig nicht ausreichend bekannt. Zum ohnehin nur schwer vorhersagbaren zyklischen Verhalten der Propagatoren kommt erschwerend hinzu, dass die Propagationsdauer des gleichen Propagators für verschiedene Domänenänderungen sehr unterschiedlich ausfallen kann.

---

[7] Beachtet werden muss, dass sich die Ausführungszeit eines Propagators stark von Aufruf zu Aufruf unterscheiden kann, abhängig davon wie die am jeweiligen Constraint beteiligten Variablen gegenwärtig belegt sind.

4.2.3 Beseitigung von verlangsamender Redundanz

Sich überlagernde Constraints können redundante Informationen enthalten. In [64] wird gezeigt, dass sich die Lösungsgeschwindigkeit von CSPs unter bestimmten Umständen durch das Einfügen redundanter Constraints beschleunigen lässt. In vielen Fällen hat eine Überlagerung von Constraints allerdings eine Verlangsamung zur Folge, da gleiche Informationen mehrfach ausgewertet werden müssen. Da viele verschiedene Arten von Constraints existieren, die über sehr unterschiedliche Strukturen mit unterschiedlichen Eigenschaften verfügen, ist es in der Regel schwer diese verlangsamende Redundanz sichtbar zu machen und zu beseitigen. Das Beispiel 4.4 veranschaulicht eine solche Redundanz in zwei Constraints.

Beispiel 4.4: Die Redundanz verschiedener Constraints. *Das CSP 11 beinhaltet zwei Constraints c_1 und c_2 unterschiedlichen Typs (ein* Summen-*Constraint und ein* AllDifferent-*Constraint). Die beiden Constraints haben auf Grund ihrer Art einen völlig unterschiedlichen Aufbau, beschreiben allerdings eine ähnliche Tupelmenge. In Abbildung 4.5 sind die beiden Tupelmengen und Automaten M_1 und M_2 von äquivalenten* Regular-*Constraints für c_1 und c_2 dargestellt. Es ist zu erkennen, dass, obwohl die beiden Constraints c_1 und c_2 unterschiedlichen Typs sind, sie bis auf eine Belegung ϕ mit $\phi(x_1) = 2, \phi(x_2) = 2$ und $\phi(x_3) = 2$ die gleiche Lösungsmenge beschreiben und die Repräsentation der beiden Automaten sich lediglich in einem Übergang (von $q_{1,1}$ mit Wert 2 zu $q_{2,1}$) unterscheiden. Die restlichen Tupel bzw. Automatenbestandteile sind redundant, müssten also folglich bei der Propagation beider Constraints doppelt ausgewertet werden.*

CSP 11: $P = (X, D, C)$ mit:

$$X = \{x_1, x_2, x_3\} \quad \text{(3 Variablen)}$$
$$D = \{D_1 = D_2 = D_3 = \{1, 2, 3\} \quad \text{(3 Domänen)}$$
$$C = \{c_1 = (x_1 + x_2 + x_3 = 6), c_2 = \mathit{AllDifferent}(\{x_1, x_2, x_3\}) \quad \text{(2 Constraints)}$$

Bei der Substituierung von c_1 und c_2 durch ein FD-vollständiges Constraint würde dieser doppelte Propagationsaufwand wegfallen und der Lösungsvorgang des Constraint-Problems kann somit beschleunigt werden.

Anmerkung 2. Auch wenn eine Menge von Constraints C viel Redundanz aufweist, muss das nicht bedeuten, dass sich eine Substituierung durch ein einzelnes Constraint beschleunigend auswirkt. Verfügen die einzelnen Constraints in C über sehr effektive Propagatoren, so kann das Entfernen von Redundanz durch das Verwenden eines einzelnen Constraints c' trotzdem zu einer Verlangsamung führen, falls der Propagator von c' deutlich ineffektiver ist als die ursprünglichen Propagatoren.

4.2.4 Spezialisierte Solver

Eine Menge von Constraints, die den gleichen Propagationsalgorithmus (Propagator) P verwenden, bezeichnen wir im Folgenden als *Constraints des gleichen Typs T_P*. Beispiels-

$c_1 = (x_1 + x_2 + x_3 = 6)$
$T_1 = \{\{1,2,3\}, \{1,3,2\}, \{2,1,3\}, \{2,2,2\}, \{2,3,1\}, \{3,1,2\}, \{3,2,1\}\}$

$c_2 = \mathit{AllDifferent}(\{x_1, x_2, x_3\})$
$T_2 = \{\{1,2,3\}, \{1,3,2\}, \{2,1,3\}, \{2,3,1\}, \{3,1,2\}, \{3,2,1\}\}$

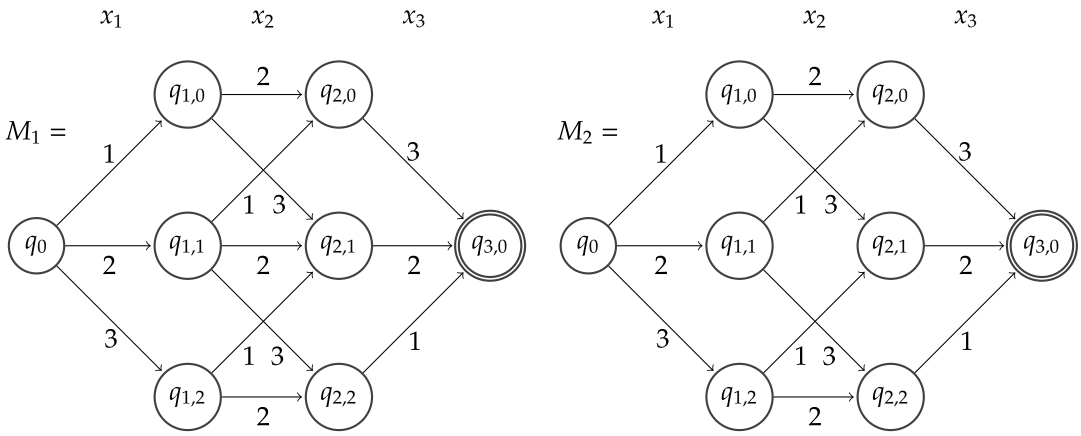


Abbildung 4.5: Die beiden Constraints c_1 und c_2 sowie deren Tupel und Automatenrepräsentation.

weise verfügen alle *Count*-Constraints über den gleichen Propagator. Gleiches gilt jeweils für alle *Table*-, *Regular*-, *Scalar*- oder *Sum*-Constraints. Darüber hinaus besteht aber auch die Möglichkeit, dass verschiedene Constraints den gleichen Propagationsalgorithmus nutzen. Beim *Sum*-, beim *Scalar*- und bei verschiedenen arithmetischen Constraints kann dieser Fall beispielsweise eintreten. Des Weiteren kann ein einzelnes Constraint verschiedene Propagatoren $P_1, ..., P_n$ besitzen. Ist dies der Fall, so ist das Constraint von allen Typen $T_{P_1}, ..., T_{P_n}$. Ein FD-CSP, welches nur über Constraints verfügt, die mindestens einen gleichen FD-vollständigen Typ besitzen, wird in dieser Arbeit als FD-vollständiges CSP bezeichnet.

Definition 4.3: ***Finite Domain-vollständige CSPs.*** Ein Typ T_P wird Finite Domain-vollständig genannt, wenn es für jedes beliebige FD-Constraint c immer eine zu c äquivalente Constraint-Menge C gibt, so dass jedes $c_i \in C$ vom Typ T_P ist.
Ein Constraint c und eine Constraint-Menge C gelten dabei als äquivalent, wenn es eine bijektive Funktion f gibt, die jeder gültigen Variablenbelegung für $X = \mathit{Scope}(c)$ eine gültige Variablenbelegung für $X' = \bigcup_{\forall c_i \in C} \mathit{Scope}(c_i)$ zuweist und umgekehrt.
Ein CSP $P = (X, D, C)$ ist Finite Domain-vollständig, wenn alle Constraints $c \in C$ mindestens einen gemeinsamen FD-vollständigen Typ T_P besitzen (nach [99]).

Zwei Vertreter für FD-vollständige CSPs sind CSPs, bei denen alle Constraints durch *Table*-Constrains (Table-CSP) oder durch *Regular*-Constraints (Regular-CSP) repräsentiert werden. Nach Definition 4.1 kann ein FD-vollständiges Constraint jedes andere FD-Constraint repräsentieren ohne deren Variablen oder Domänen anzupassen. Diese

Eigenschaft wird für FD-vollständige CSPs nicht gefordert. Die Constraints eines FD-vollständigen CSPs müssen selbst nicht FD-vollständig sein. Ein Beispiel dafür ist das boolesche *Scalar*-Constraint (Vergleich Abschnitt 2.7). Dieses fordert, dass das Skalarprodukt aus einem Vektor boolescher Variablen und einem Vektor von konstanten Koeffizienten in Relation zu einem konstanten Wert steht. Es ist offensichtlich, dass ein beliebiges FD-Constraint nicht ohne Anpassung der Variablen und Domänen durch ein Constraint, welches nur boolesche Variablen erlaubt, repräsentiert werden kann. Wie in Kapitel 5 gezeigt wird, ist ein CSP, welches nur boolesche Variablen und *Scalar*-Constraints über diesen erlaubt, hingegen ein FD-vollständiges CSP. Da per Definition alle FD-Constraints in FD-vollständige Constraints überführbar sind, bietet es sich trotzdem an, den Typ eines FD-vollständigen Constraints als Typ für die Constraints eines FD-vollständigen CSPs zu verwenden.

Dass ein Typ T_P FD-vollständig ist, kann unter anderem dadurch gezeigt werden, dass eine allgemeingültige Bildungsvorschrift angegeben wird, die angibt, wie ein beliebiges FD-CSP P in ein FD-vollständiges CSP P^{FD} überführt wird.

Es wird an dieser Stelle zwischen zwei Arten von FD-vollständigen CSPs unterschieden. Auf der einen Seite sind es die direkt konvertierten FD-vollständigen CSPs vom Typ T_P, die erreicht werden können, indem lediglich die Constraints eines beliebigen CSPs in Constraints vom Typ T_P überführt werden. Auf der anderen Seite stehen die indirekt konvertierten FD-vollständigen CSPs vom Typ T_P, bei denen zusätzlich zu den Constraints auch die Variablen und Domänen angepasst werden müssen.

Definition 4.4: ***Direkt konvertierte FD-vollständige CSPs.*** Ein FD-vollständiges CSP P^{fd}, das mittels Substituierung der Constraints $c \in C$ eines gegebenen CSPs $P = (X, D, C)$ durch Constraints vom Typ T_P entstanden ist, ohne dass Variablen oder Domänen angepasst werden müssen, wird als direkt konvertiertes FD-vollständiges CSP bezeichnet (nach [99]).

Definition 4.5: ***Indirekt konvertierte FD-vollständige CSPs.*** Ein FD-vollständiges CSP $P^{fd} = (X', D', C')$, das mittels Substituierung der Constraints $c \in C$ eines gegebenen CSPs $P = (X, D, C)$ durch Constraints $c' \in C'$ vom Typ T_P unter Umwandlung der Variablen X und Domänen D in neue Variablen X' und Domänen D' mit $X \neq X' \vee D \neq D'$ überführt wird, wird als indirekt konvertiertes FD-vollständiges CSP bezeichnet (nach [99]).

Die zuvor erwähnten *Table*-CSPs und *Regular*-CSPs sind Vertreter der direkt konvertierten FD-vollständigen CSPs, da sie durch Konvertierung der ursprünglichen Constraints in *Table*- bzw. *Regular*-Constraints direkt erzeugt werden können. Vertreter der indirekt konvertierten FD-vollständigen CSPs sind unter anderem CSPs mit lediglich binären Constraints und SAT-Probleme, die z.B. durch binäre Transformationen (siehe Abschnitt 3.1) oder Encoding (siehe Abschnitt 3.2) erzeugt werden können. Ein weiterer Vertreter der indirekt konvertierten FD-vollständigen CSPs sind die linearen pseudo-booleschen CSPs, also solche, die nur boolesche Variablen und lineare Constraints über diesen enthalten. In Kapitel 5 werden verschiedene allgemeingültige als auch spezielle Bildungsvorschriften für das Umwandeln von FD-CSPs in lineare pseudo-boolesche CSPs aufgeführt. Durch Angabe der allgemeingültigen Bildungsvorschriften wird folglich auch gezeigt, dass boolesche *Scalar*-CSPs FD-vollständige CSPs sind.

In dieser Arbeit wird der Begriff boolesches *Scalar*-CSP als Synonym für lineare pseudo-boolesche CSPs verwendet.

Definition 4.6: *Boolesche* Scalar-*CSPs*. Ein boolesches *Scalar*-CSP ist ein CSP $P = (X^B, D^B, C^{Scalar})$, das nur boolesche Variablen mit Domänen $D_1 = ... = D_n = \{0, 1\}$ beinhaltet und bei dem jedes Constraint $c \in C^{Scalar}$ ein *Scalar*-Constraint der Form: $Scalar(X', \vec{c}, \Re, r)$ mit $X' \subseteq X^B, \vec{c} \in \mathbb{Z}^{|X'|}, \Re \in \{<, \leq, =\geq, >, \neq\}$ und $r \in \mathbb{Z}$ darstellt [99].

Als Folge der Umwandlung allgemeiner CSPs in speziellere können zum Lösen dieser transformierten CSPs spezialisierte Solver, wie z.B. der SAT-Solver, verwendet werden. Ziel ist es, auch für andere Typen von Constraints, wie zum Beispiel das *Table*- oder *Regular*-Constraint, solch spezialisierte Solver zu erzeugen, um eine Performancesteigerung zu erreichen.

Liegen zum Beispiel alle Constraints $C = \{c_1, ..., c_m\}$ als *Table*-Constraints vor, kann also auf alle Tupellisten $T = \{T_1, ..., T_m\}$ der Constraints zugegriffen werden, so kann ein *Table*-Solver mittels Merge- oder Join-Methoden alle Tupellisten zu einer Tupelliste kombinieren und somit alle gültigen Tupel des ursprünglichen CSPs auflisten. Die Herausforderung dabei ist es, die richtige Reihenfolge der Merge- bzw. Join-Operationen festzulegen, um die Vereinigung schnellstmöglich durchführen zu können.

Ein ähnlicher Ansatz kann bei *Regular*-CSPs verfolgt werden. Liegen alle Constraints $C = \{c_1, ..., c_m\}$ eines CSPs als *Regular*-Constraints vor, so können die zugrundeliegenden Automaten $M = \{M_1, ..., M_m\}$ mittels Automatenschnittmengenbildung in einen alle Constraints umfassenden Automaten überführt werden. In einem solchen Schnittmengenautomaten repräsentiert jeder Pfad vom Startknoten zu einem finalen Knoten eine Lösung des ursprünglichen CSPs. An dieser Stelle besteht die Herausforderung darin, die richtige Reihenfolge der Schnittmengenoperationen, aber auch die richtige Reihenfolge der Variablen festzulegen, um schnellstmöglich einen kleinstmöglichen levelbasierten DFA zu erzeugen.

In diesem Kapitel wurden *Regular*-Constraints und *Table*-Constraints als Vertreter für FD-vollständige Constraints und Table-CSPs, Regular-CSPs sowie boolesche *Scalar*-CSPs als Vertreter für FD-vollständige CSPs eingeführt. Anhand eines Beispiels zur Tabularisierung und Regularisierung wurde jeweils verdeutlicht, dass Transformationen von einzelnen Constraints, Teil-CSPs oder ganzen CSPs zu einer Beschleunigung des Lösungsverfahrens von CSPs führen können.

Des Weiteren wurden Gründe für eine mögliche Beschleunigung des Lösungsvorganges von CSPs durch Substituierung erarbeitet. Das Hauptproblem, welches sich bei den vorgestellten Substituierungen abzeichnet, ist der hohe Zeitaufwand, der für die Transformationen der CSPs benötigt wird. Diesen gilt es in der Folge zu verringern.

5 Die Substituierung von Constraints

In Kapitel 4 konnte gezeigt werden, dass das Transformieren von einzelnen Constraints, Teil-CSPs oder ganzen CSPs in FD-vollständige Constraints oder FD-vollständige CSPs zu einer Beschleunigung des Lösungsvorgangs eines CSPs führen kann. Auf der anderen Seite erhöht sich durch die Transformationen der Aufwand zum Erstellen des zu lösenden CSPs teilweise signifikant, was den Effekt der Verringerung der Lösungszeit teilweise zunichte macht oder sogar zu einer Verlangsamung führen kann. Um diesem Problem zu begegnen, werden im Folgendem weitere Algorithmen entwickelt, die eine schnellere Transformation von beliebigen und speziellen Constraints in *Regular*- und boolesche *Scalar*-Constraints erlauben.

Zunächst werden in Abschnitt 5.1 allgemeingültige Verfahren für das Umwandeln von FD-Constraints in *Regular*- bzw. boolesche *Scalar*-Constraints eingeführt, bevor in Abschnitt 5.2 auf die Substituierung ausgewählter globaler Constraints eingegangen wird. Folgend wird in Abschnitt 5.3 auf die Substituierung boolescher Meta-Constraints eingegangen. Dies sind Constraints, die wiederum selbst Constraints als Eingaben erhalten. Abschließend werden im Abschnitt 5.4 Schlussfolgerungen, die sich aus der Regularisierbarkeit und booleschen Skalarisierbarkeit von FD-Constraints und -CSPs ergeben, gezogen.

5.1 Die Substituierung von kleinen Constraint-Mengen

Ausgehend von einem CSP $P = (X, D, C)$ bietet sich für die Umwandlung einer Constraint-Menge $C' \subseteq C$ in ein *Table*- (c^t) oder *Regular*-Constraint (c^r) ein zweigeteiltes Vorgehen an:

1. Das Lösen des Teil-CSPs $P' = (X', D', C')$, mit $X' = \bigcup_{\forall c \in C'} scope(c)$ und D' enthält die zu X' gehörenden Domänen.
2. Das Generieren von spezifischen Constraints aus den Lösungen von P'.

Das Lösen des Teil-CSPs P' kann unabhängig von der Substituierungsart (Tabularisierung, Regularisierung, boolesche Skalarisierung) mit den gängigen Verfahren der Constraint-Programmierung realisiert werden. Insbesondere können dabei weiterführende Techniken wie Parallelisierung oder erneute Remodellierung genutzt werden. In der Regel sollte es sich bei P' allerdings um ein wesentlich kleineres und somit schneller zu lösendes CSP als P handeln. Ist das Teil-CSP P' zu groß bzw. zu komplex gewählt, so dass das Finden aller Lösungen dieses CSPs nicht wesentlich schneller erfolgt als das Finden aller Lösungen von P, so kann durch die Transformation auch keine Verringerung der Lösungszeit erzielt werden. Es gilt zu beachten, dass die Transformationszeit für das Umwandeln von P zu P' zusätzlich zur Lösungszeit von P' berücksichtigt werden muss.

Anmerkung 3. Da die Lösungsgeschwindigkeit eines CSPs nur schwer vorhersagbar ist, kann als Approximation die Suchraumgröße, die sehr schnell durch das Multiplizieren der Domänengrößen berechnet wird, herangezogen werden. Die verwendete Hard- und Software kann im Einzelfall sehr stark differieren, was einen erheblichen Einfluss auf die Laufzeit von Berechnungen auf den verschiedenen Endgeräten hat. Daher empfiehlt es sich, für jeden Rechner eine spezifische Maximalgröße anzugeben, die ein zu substituierendes CSP haben kann, damit dieses noch in adäquater Zeit von der jeweiligen Hardware gelöst werden kann.

Für die Tabularisierung ergibt sich, dass die Lösungen ϕ_j mit $\phi_j(x_1) = d_1, ..., \phi_j(x_n) = d_n$ von P' direkt in Tupel $t_j = (d_1, ..., d_n)$ überführt werden können. Sei T die Liste aller so erzeugter Tupel. Das Constraint $\mathit{Table}(X', T)$ mit $X' = \{x_1, ..., x_n\}$ dient dann als Substituierung für die Constraints C' in P.

Das Erstellen eines adäquaten *Regular*- oder booleschen *Scalar*-Constraints aus den Lösungen S von P' gestaltet sich als wesentlich komplexer als das Erzeugen eines *Table*-Constraints und wird in den folgenden Abschnitten detailliert erläutert.

5.1.1 Die Regularisierung von CSPs

In diesem Abschnitt wird die Regularisierung von FD-CSPs vorgestellt. Das beschriebene Vorgehen ist dabei auf alle FD-CSPs und somit auch auf alle FD-Constraints anwendbar.

Die im Folgenden vorgestellten Algorithmen zur Erzeugung eines Automaten (levelbasierten DFAs) aus den k Lösungen $\phi_1, ..., \phi_k \in (X \to D)$ eines Teil-CSPs $P' = (X, D, C)$ wurden in Zusammenhang mit dieser Arbeit entworfen und in [98] veröffentlicht. Es wird dabei davon ausgegangen, dass die Variablen $X = \{x_1, ..., x_n\}$ in der festen Reihenfolge $x_1, ..., x_n$ vorliegen.

Die Matrix $S \in D'^{n \times k}$ enthält alle Werte aller Lösungen von P'. Jeder Eintrag $S_{i,j}$ der Matrix ergibt sich dabei aus dem Wert, der der Variablen x_i in Lösung ϕ_j zugeordnet ist ($S_{i,j} = \phi_j(x_i), \forall i \in \{1, ..., n\}, j \in \{1, ..., k\}$). In Tabelle 5.1 ist die Lösungsmatrix S visualisiert. Jede Spalte in S repräsentiert folglich eine Lösung von P'.

Für die Erzeugung eines levelbasierten DFAs als Grundlage für das zu erzeugende *Regular*-Constraint werden die Mengen $T = \{T_1, ..., T_n\}$ der Präfixe mit Größe $i \in \{1, ..., n\}$ der Lösungssequenzen S_j benötigt. Jede Präfix-Menge T_i beinhaltet genau die i-stelligen

S	S_1	S_2	$\dots$	S_k
x_1	$S_{1,1}$	$S_{1,2}$	$\dots$	$S_{1,k}$
x_2	$S_{2,1}$	$S_{2,2}$	$\dots$	$S_{2,k}$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
x_n	$S_{n,1}$	$S_{n,2}$	$\dots$	$S_{n,k}$

Tabelle 5.1: Die Lösungsmatrix S des Teil-CSPs P'.

Tupel, die als partielle Belegung für $\{x_1, ..., x_i\}$ fungieren können, so dass mindestens eine partielle Belegung für $\{x_{i+1}, ..., x_n\}$ existiert, welche P' erfüllt (siehe Gleichung 5.1).

$$T_i = \bigcup_{l=1}^{k} \{(S_{1,l},\ S_{2,l},\ \dots,\ S_{i,l}) \mid \forall i \in \{1, \dots, n\}\} \tag{5.1}$$

Dementsprechend enthält T_1 alle einstelligen Tupel $(S_{1,v})$ mit $v \in D_1$ und $\phi(x_1) = v$ ist Teil mindestens einer gültigen Belegung für das zu substituierende CSP P'. Die Menge T_2 enthält alle verschiedenen zweistelligen Tupel (a, b) mit $a \in \{S_{1,1}, ..., S_{1,k}\}$ und $b \in \{S_{2,1}, ..., S_{2,k}\}$, für die eine gültige Variablenbelegung $\phi \in X \to \mathbb{N}^n$ mit $\phi(x_1) = a$ und $\phi(x_2) = b$ existiert.

Die Menge T_n entspricht der Menge der Lösungen von P' in Form von Tupeln (hier auch als Sequenzen bezeichnet). Es ergibt sich automatisch, dass jede Menge T_{i+1} mit $i \in \{1, ..., n-1\}$ immer mindestens so viele Tupel enthalten muss wie die Menge T_i $(|T_1| \leq |T_2| \leq \dots \leq |T_n| = k)$.

Im Folgenden werden die Mengen T_1 bis T_n genutzt um einen levelbasierten DFA $M = (Q, \Sigma, \delta, q_0, q_f)$ zu erzeugen. Für jedes Element t jeder Menge T_i mit $i \in \{1, ..., n-1\}$ kann ein Zustand q_t erzeugt werden, der das Präfix t repräsentiert. Zusätzlich wird ein Startzustand q_0 und ein finaler akzeptierender Zustand q_f erzeugt. Die so entstandenen Zustände unterliegen einer Partitionierung in $n+1$ viele Level $Q_0, ..., Q_n$ mit $Q = \bigcup_{i=0}^{n} Q_i$ und $Q_i \cap Q_j = \emptyset, \forall i, j \in \{0, ..., n\}, i \neq j$, wobei Q_0 nur den Zustand q_0, Q_n nur den Zustand q_f und alle anderen Zustandsmengen $Q_i, \forall i \in \{1, ..., n-1\}$ genau die Zustände enthalten, die die i-elementigen Tupel T_i repräsentieren. Somit ergibt sich die Menge der Zustände Q wie folgt:

$$\begin{aligned} Q = (Q_0 = \{q_0\}) \cup (Q_1 = \{q_{1,t} \mid \forall t \in \{1, ..., |T_1|\}\}) \cup ... \\ ... \cup\ (Q_{n-1} = \{q_{n-1,t} \mid \forall t \in \{1, ..., |T_{n-1}|\}\}) \cup (Q_n = \{q_f\}) \end{aligned} \tag{5.2}$$

Das Alphabet Σ des Automaten ist die Vereinigung aller Domänenwerte D der Variablen X des Teil-CSPs P':

$$\Sigma = \bigcup_{D_i \in D} D_i \tag{5.3}$$

Abschließend muss für die Definition eines Automaten die Übergangsfunktion δ definiert werden. Im Folgenden wird dafür die Syntax $\delta(q_1, w) \rightarrow q_2$ verwendet, um auszudrücken, dass von einem Zustand q_1 mit der Eingabe w in den Zustand q_2 übergegangen wird. Es ergeben sich für die Erzeugung des levelbasierten DFAs drei Fälle für die Übergangsfunktion δ.

1. Der Ausgangszustand ist der Startzustand q_0. Dann folgt, dass für jedes 1-Tupel $t_r = (v) \in T_1$ der folgende Übergang erlaubt ist:

   $$\delta(q_0, v) = q_{1,r} \tag{5.4}$$

2. Der Ausgangszustand ist ein innerer Zustand $q_{i,l} \in Q_i$, für $i \in \{1, ..., n-2\}$, der das Tupel $t_l = (v_1, ..., v_i)$ repräsentiert und der Zielzustand ist nicht der finale Zustand q_f. Dann folgt, dass für jedes $(i+1)$-Tupel $t_r = (v_1, ..., v_i, v_{i+1}) \in T_{i+1}$ der folgende Übergang erlaubt ist:

   $$\delta(q_{i,l}, v_{i+1}) = q_{i+1,r} \tag{5.5}$$

3. Der Ausgangszustand ist $q_{n-1,l} \in Q_{n-1}$, der das Tupel $t_l = (v_1, ..., v_{n-1})$ repräsentiert und der Zielzustand ist der finale Zustand q_f. Dann folgt, dass für jedes n-Tupel $t_r = (v_1, ..., v_n) \in T_n$ der folgende Übergang erlaubt ist:

   $$\delta(q_{n-1,l}, v_n) = q_f \tag{5.6}$$

Der entstandene levelbasierte DFA $M = (Q, \Sigma, \delta, q_0, q_f)$ akzeptiert genau die Lösungen von P' und das Constraint *Regular*(X, M) kann als Substituierung für die Constraints C im ursprünglichen CSP verwendet werden.

Anmerkung 4. Der so erzeugte levelbasierte DFA ist in der Regel nicht-minimal, er kann vor der Verwendung im Propagator allerdings noch minimiert werden, um so den Rechenaufwand für die vielfach auszuführende Propagation zu minimieren. An dieser Stelle kann auf die im Vergleich zu allgemeinen DFAs wesentlich effektivere Minimierung der levelbasierten DFAs zurückgegriffen werden, bei der nur Zustände $q_j \in Q_i$ mit Zuständen des gleichen Levels $q_k \in Q_i$ für $j \neq k, i \in \{1, ..., n\}$ verglichen werden müssen und nicht mit allen anderen Zuständen.

In Algorithmus 2 ist dargestellt, wie der zuvor beschriebene Automat konkret erzeugt werden kann. Zunächst werden die eingegebenen Tupel (ein zweidimensionales Array, das die Matrix S repräsentiert) mittels der in Gleichung 5.7 angegebenen lexikographischen Ordnung sortiert (Zeile 1), und ein Startzustand Q_0 und ein finaler Zustand Q_f erzeugt (Zeilen 2-4). Des Weiteren werden zwei Arrays Qs und *values* angelegt, die die Zustände entlang des aktuellen Pfades vom Startzustand hin zum finalen Zustand (exklusive Start- und Endzustand) und die Werte der Übergänge entlang dieses Pfades (inklusive Start- und Endzustand) repräsentieren (Zeilen 5 und 6). Das *values*-Array wird dabei mit Werten $\perp$ initialisiert, die nicht im Alphabet Σ des DFAs liegen.

$$\begin{aligned} t_1 = (t_{1,1}, \ldots, t_{1,n}) \leq (t_{2,1}, \ldots, t_{2,n}) = t_2 \leftrightarrow\ & (t_{1,1} < t_{2,1}) \\ & \vee ((t_{1,1} = t_{2,1}) \wedge (t_{1,2} < t_{2,2})) \\ & \vee \ldots \\ & \vee ((t_{1,1} = t_{2,1}) \wedge \ldots \wedge (t_{1,n-1} = t_{2,n-1}) \wedge (t_{1,n} < t_{2,n})) \\ & \vee ((t_{1,1} = t_{2,1}) \wedge \ldots \wedge (t_{1,n-1} = t_{2,n-1}) \wedge (t_{1,n} = t_{2,n})) \end{aligned} \tag{5.7}$$

Für jedes Tupel $t \in tuples$ wird anschließend der Pfad entlang der Knoten Qs solange verfolgt, bis der aktuelle Tupelwert $t[i]$ von dem zuvor abgespeicherten Wert $values[i]$ abweicht (Zeilen 7-9). Aufgrund der vorherigen Sortierung der Eingabetupel (siehe Gleichung 5.7) hat der Pfad vom Startzustand Q_0 bis Q_{i-1}, also der Pfad, der für das vorherige Tupel und das aktuelle gleich ist, eine maximale Länge. Der levelbasierte DFA *dfa* enthält somit schon den Teil des Lösungspfades von Q_0 zu Q_{i-1} mit den Werten $t[0], \ldots, t[i-1]$. Es muss lediglich der Pfad von Q_{i-1} bis Q_f mit den Werten $t[i], \ldots, t[nbVars-1]$ ergänzt werden. Dafür wird ein neuer Zustand erstellt, an der entsprechenden Stelle $Qs[i]$ dem Pfad hinzugefügt und ein Übergang vom alten Zustand Q_{old} zum neuen Zustand $Qs[i]$ mit dem Wert $t[i]$ erzeugt (Zeilen 10 und 11). Anschließend wird die Werteänderung in das *values*-Array aufgenommen (Zeile 12) und die weiteren Array-Felder $values[i+1], \ldots, values[nbVars-1]$ werden mittels der *removeValues*-Methode auf den $\perp$-Wert gesetzt (Zeile 13). Für den letzten Wert eines jeden Tupels $t[nbVars-1]$ wird ein Übergang vom jeweils letzten Zustand des aktuellen Pfades (abgelegt in Q_{old}) zum finalen Zustand Q_f erzeugt (Zeile 15).

Abschließend wird der levelbasierte DFA aus dem Startzustand mit allen von diesem erreichbaren Zuständen erzeugt (Zeile 16) und für eine optimale Nutzung im späteren Propagator minimiert (Zeile 17). Die Minimierung ist dabei optional, führt aber zu einer Erhöhung der Propagationsgeschwindigkeit. Der erzeugte DFA wird zurückgegeben und kann in einem *Regular*-Constraint als Substituierung für die Constraint-Menge mit den Lösungen *tuples* verwendet werden.

Ein ausführliches Beispiel, das die Wirkungsweise, das Potential und die Probleme dieses Vorgehens erläutert, wurde bereits in Abschnitt 4.1.2 mit Beispiel 4.2 gegeben.

Algorithmus 2: Ein Algorithmus zum Erzeugen eines levelbasierten DFAs

```
Input: int[][]: tuples
       int: nbVars
Output: DFA: dfa
tuples = sortTuples(tuples)
State $Q_0$ = new State
State $Q_{old}$ = $Q_0$
State $Q_f$ = new State
State[] Qs = new State[nbVars − 1]
int[] values = {⊥, ⊥, ..., ⊥}
forall (int[] t : tuples) do
    forall (int i = 0; i < nbVars − 1; i + +) do
        if (values[i] ≠ t[i]) then
            Qs[i] = new State
            $Q_{old}$.addEdge(Qs[i], t[i])
            values[i] = t[i]
            removeValues(values, i + 1)
        $Q_{old}$ = Qs[i]
    $Q_{old}$.addEdge($Q_f$, t[nbVars − 1])
DFA dfa = new DFA($Q_0$)
dfa.minimize()
return dfa
```

5.1.2 Die boolesche Skalarisierung von CSPs

In diesem Abschnitt werden zwei neue Verfahren zur booleschen Skalarisierung von FD-CSPs eingeführt. Das erste Verfahren ist dabei bezüglich der Repräsentation der Constraints an die direkte Encoding-Variante und das zweite Verfahren an die Support Encoding-Variante angelehnt (Vergleich Abschnitt 3.2). In beiden Fällen wird analog zum direkten Encoding für jede ursprüngliche Variable $x_i \in X$ eines CSPs oder Teil-CSPs und für jeden Wert der zugehörigen Domäne $d_j \in D_i$ eine boolesche Variable $x^B_{i,j} \in X^B$ angelegt, die widerspiegelt, ob die ursprüngliche Variable x_i den j-ten Wert der Domäne D_i annimmt oder nicht ($x_i = d_j \leftrightarrow x^B_{i,j} = 1$).

Da die konfliktbasierte boolesche Skalarisierung für jedes ungültige Tupel des zu substituierenden Constraints ein neues *Scalar*-Constraint über booleschen Variablen anlegt, ist dieses Verfahren besonders gut für die Substituierung von Constraints mit anteilig sehr vielen Lösungen geeignet. Die Support-basierte boolesche Skalarisierung erzeugt hingegen für jedes gültige Tupel des zu substituierenden Constraints ein neues

Scalar-Constraint über booleschen Variablen. Daher ist dieses Verfahren besonders gut für Constraints mit anteilig sehr wenigen Lösungen geeignet. In der Praxis können die beiden Verfahren beim Substituieren eines CSPs in Kombination eingesetzt werden. Mittels Stichproben kann dabei ermittelt werden, ob ein Constraint über viele oder wenige gültige Tupel verfügt und das entsprechende boolesche Skalarisierungsverfahren ausgewählt werden.

Die konfliktbasierte boolesche Skalarisierung

An dieser Stelle wird ein CSP bzw. Teil-CSP $P = (X, D, C)$ mit $X = \{x_1, ..., x_n\}$, $D = \{D_1, ..., D_n\}$ und $C = \{c_1, ..., c_m\}$ als Ausgangspunkt für die neu entwickelte konfliktbasierte boolesche Skalarisierung betrachtet.

Die aus dem direkten Encoding bekannten *addLeastOne* $\bigvee_{v \in D_i} x^{de}_{i,v}$ und *addMostOne*-Klauseln $(\neg x^{de}_{i,v_1} \vee \neg x^{de}_{i,v_2}), \forall v_1, v_2 \in D_i$ mit $i \neq j$ wurden eingeführt, um für die Variablen $X^B_i = \{x^B_{i,j} \mid \forall j \in \{1, ..., |D_i|\}, i \in \{1, ..., n\}\}$ sicherzustellen, dass exakt eine von ihnen den Wert 1 annimmt. Anstelle einer Klauselmenge müssen bei der booleschen Skalarisierung *Scalar*-Constraints verwendet werden, um diesen Sachverhalt auszudrücken. Die Constraints $C^{oneOff} = \{Scalar(X^{de}_i, (1, ..., 1)^T, =, 1) \mid \forall i \in \{1, ..., n\}\}$ gewährleisten genau das gewünschte Verhalten, wobei ein *oneOff*-Constraint jeweils die Einschränkungen sowohl einer *addLeastOne*-Klausel als auch einer *addMostOne*-Klausel zusammen repräsentiert. Folgend werden analog zum direkten Encoding die ungültigen Belegungen des (Teil-)CSPs P in Constraints überführt. Im Gegensatz zum direkten Encoding haben diese allerdings nicht die Form einer Klauselmenge, sondern die einer Menge von *Scalar*-Constraints über booleschen Variablen.

Definition 5.1: ***Die konfliktbasierte boolesche Skalarisierung.*** Sei ein CSP $P = (X, D, C)$ gegeben, seine konfliktbasierte boolesche Skalarisierung sei definiert durch $P^{kbs} = (X^B, D^B, C^S)$ mit:

- $X^B = \{x^B_{i,j} \mid i \in \{1, ..., |X|\}, j \in \{1, ..., |D_i|\}\}$ ist eine Menge von booleschen Variablen, wobei jede Variable $x^B_{i,j}$ widerspiegelt, ob die ursprüngliche Variable x_i den j-ten Wert der Domäne D_i annimmt oder nicht $(x_i = d_j \leftrightarrow x^B_{i,j} = 1)$.

- $D^B = \{D^B_{i,j} = \{0, 1\} \mid i \in \{1, ..., |X|\}, j \in \{1, ..., |D_i|\}\}$ ist die Menge der booleschen Domänen der neu erzeugten Variablen.

- $C^S = C^{oneOff} \cup C^{constraint}$ ist die Menge der *Scalar*-Constraints über den neu erzeugten booleschen Variablen X^B.

  Damit die Variablen $X^B_i = \{x^B_{i,j} \mid \forall j \in \{1, ..., |D_i|\}\}$ die möglichen Variablenzuweisungen für x_i repräsentieren können, muss gewährleistet werden, dass exakt eine Variable in X_i^B den Wert 1 annimmt. Dafür werden die *oneOff*-Constraints $C^{oneOff} = \{Scalar(X_i^B, (1, ..., 1)^T, =, 1) \mid \forall i \in \{1, ..., n\}\}$ angelegt.

Für jedes ungültige Tupel $t_i = (v_1, ..., v_n) \notin T$ jedes Constraints $c_j = (\{x_1, ..., x_n\}, T) \in C$ des ursprünglichen CSPs P wird ein neues Constraint $c_{i,j}^s = Scalar(\{x_{1,idx(v_1)}^B, x_{2,idx(v_2)}^B, ..., x_{n,idx(v_n)}^B\}, (1, ..., 1)^T, \neq, n)$ erstellt, das dieses Tupel ausschließt. Die Funktion $idx(v_i)$ ermittelt dabei den Index von v_i in der zugehörigen Domäne D_i. Die Menge $C^{constraint}$ entspricht der Menge aller solcher Constraints $c_{i,j}^s$ für alle ungültigen Tupel der ursprünglichen Constraints.

Durch die angegebene Umwandlung lassen sich alle FD-CSPs in boolesche *Scalar*-CSPs überführen, allerdings entstehen dabei sehr viele Constraints. Allein für jedes ungültige Tupel $t_i \notin T_j$ jedes Constraints $c_j = (X_j, T_j)$ des ursprünglichen CSPs P wird ein neues Constraint $c_{i,j}^s \in C^{constraint}$ angelegt. Zusätzlich wird für jede Variable $x_i \in X$ des ursprünglichen CSPs P ein Constraint $c_i \in C^{oneOff}$ angelegt, das repräsentiert, dass der Variablen x_i genau ein Wert ihrer Domäne zugeordnet wird.

Abhängig von der Anzahl ungültiger Tupel jedes ursprünglichen Constraints kann die Menge der zu erzeugenden booleschen *Scalar*-Constraints gegebenenfalls reduziert werden. Wird nicht für jedes ungültige Tupel jedes ursprünglichen Constraints $c \in C$ ein boolesches *Scalar*-Constraint erzeugt, sondern werden mehrere ursprüngliche Constraints zu einem Teil-CSP P' zusammengefasst und wird anschließend unter Berücksichtigung einer festen Variablenreihenfolge für jedes ungültige Tupel von P' ein boolesches *Scalar*-Constraint erzeugt, so verändert sich die Anzahl der booleschen *Scalar*-Constraints nach oben oder unten, je nachdem, ob P' mehr oder weniger ungültige Tupel besitzt als die einzelnen Constraints akkumuliert.

Im Falle, dass nicht das gesamte CSP transformiert werden soll, müssen die ursprünglichen Variablen $x_i \in X$, analog zum direkten Encoding, mittels zusätzlicher Constraints $C^{binding} = \{x_i = d_j \leftrightarrow x_{i,j}^B = 1 \mid \forall i \in \{1, ..., n\}, j \in \{1, ..., |D_i|\}\}$ an die neuen booleschen Variablen $x_{i,j}^B \in X^B$ gebunden werden.

Im folgenden Beispiel 5.1 wird gezeigt, wie ein FD-CSP in ein boolesches *Scalar*-CSP umgewandelt werden kann.

Beispiel 5.1: Die konfliktbasierte boolesche Skalarisierung[1]. *Gegeben sei das CSP 8 aus den Beispielen 4.1 und 4.2, welches in ein boolesches* Scalar-*CSP umgewandelt werden soll. Die Variablen und Domänen des ursprünglichen CSPs werden in boolesche Variablen umgewandelt, wobei jede neue Variable* $x_{i,j}^B$ *repräsentiert, ob die ursprüngliche Variable* $x_i \in X$ *den* j*-ten Wert aus der Domäne* D_i *annimmt oder nicht. Es ergeben sich somit die neuen Variablen* $X^B = \{x_{i,j}^B \mid \forall i \in \{1, ..., 7\}, j \in \{1, ..., 5\}\}$ *mit Domänen* $D_{i,j} = \{0, 1\}, \forall i \in \{1, ..., 7\}, j \in \{1, ..., 5\}$.

---

[1] Zum Lösen aller im Beispiel erzeugten CSPs wurde jeweils der CHOCO-Solver mit der in Abschnitt 1.4 angegebenen Suchstrategie, Software und Hardware verwendet.

Für jede Variable $x_i \in X$ *muss ein* Scalar-*Constraint angelegt werden, das garantiert, dass nur eine Variablenbelegung für* x_i *gleichzeitig möglich ist, also nur eine Variable* $x_{i,1}, ..., x_{i,5}$ *den Wert 1 annimmt für* $i \in \{1, ..., 7\}$*. Daraus ergeben sich die OneOff-Constraints* $C^{oneOff} = \{c_i = \text{Scalar}(\{x_{i,1}, ..., x_{i,5}\}, (1,1,1,1,1)^T, =, 1) \mid \forall i \in \{1, ..., 7\}\}$.

Für die neu erzeugten Variablen müssen anschließend neue Scalar-*Constraints angelegt werden, welche die Constraints* c_1 *bis* c_5 *adäquat ersetzen. Dafür müssen für jedes Constraint* $c_i, \forall i \in \{1, ..., 5\}$ *die ungültigen Tupel* $\overline{T_i}$ *ermittelt werden, die keine Lösung von* c_i *darstellen. Für* c_1 *ergeben sich dabei die Tupel* $\overline{T_1} = \{(1,1), (2,2), (3,3), (4,4), (5,5)\}$*. Für die weiteren Constraints* c_2 *bis* c_5 *ergeben sich negative Tupellisten der Größen:* $|\overline{T_2}| = 115$, $|\overline{T_3}| = 27$, $|\overline{T_4}| = 69165$ *und* $|\overline{T_5}| = 35$.

Die negativen Tupellisten werden anschließend nach dem zuvor beschriebenen Verfahren in Scalar-*Constraints umgewandelt. Für* c_1 *ergeben sich die folgenden fünf* Scalar-*Constraints:* $\text{Scalar}(\{x_{1,1}, x_{2,1}\}, (1,1)^T, \neq, 2)$, $\text{Scalar}(\{x_{1,2}, x_{2,2}\}, (1,1)^T, \neq, 2)$, $\text{Scalar}(\{x_{1,3}, x_{2,3}\}, (1,1)^T, \neq, 2)$, $\text{Scalar}(\{x_{1,4}, x_{2,4}\}, (1,1)^T, \neq, 2)$ *und* $\text{Scalar}(\{x_{1,5}, x_{2,5}\}, (1,1)^T, \neq, 2)$.

Die anderen negativen Tupellisten können analog umgewandelt werden, sodass ein boolesches Scalar-*CSP mit 35 Variablen und 69354 Constraints* ($5 + 115 + 27 + 69165 + 35$ *für die Repräsentation der ursprünglichen Constraints* c_1 *bis* c_5 *und 7 OneOff-Constraints) entsteht. Analog zu den Beispielen 4.1 und 4.2 wurden auch in diesem Fall verschiedene Varianten des Problems erzeugt.*

Die in Abschnitt 4.1.1 festgelegte Namensgebung wurde beibehalten, die Substituierungsart wurde lediglich um B für boolesche Skalarisierung erweitert. Statt der Substituierung von Teil-CSPs mit den beiden in Abschnitt 4.1.1 vorgestellten Varianten 1 und 2 ist bei der booleschen Skalarisierung der V-Wert entweder K für konflikbasiert oder S für Support-basiert. Es ergibt sich somit zum Beispiel die Bezeichnung $B(\{\{c_1\}^K, \{c_2\}^K, \{c_3\}^K, \{c_4\}^K, \{c_5\}^K\})$ *für die konfliktbasierte boolesche Skalarisierung von CSP 8, wobei jedes Constraint separat transformiert wurde. Die Bezeichnung* $B(\{\{c_1, c_2, c_3\}^K, \{c_4\}^K\})$ *wiederum gibt an, dass die drei Constraints* c_1*,* c_2 *und* c_3 *als Teil-CSP* $P' = (\{x_1, x_2, x_3\}, \{D_1, D_2, D_3\}, \{c_1, c_2, c_3\})$*, mittels konfliktbasierter boolescher Skalarisierung substituiert, das Constraint* c_4 *separat in boolesche* Scalar-*Constraints überführt, die neu entstandenen booleschen Variablen mit den ursprünglichen Variablen verknüpft und das Constraint* c_5 *unverändert beibehalten wurde.*

In Tabelle 5.2 sind die durchschnittlichen Zeitaufwände für das Erstellen $t(create_P)$*, Transformieren* $t(sub)$ *und Lösen* $t(solve(P^t))$*, die akkumulierte Zeit* $t(total)$ *sowie die Anzahl der benötigten Variablen und Constraints für die verschiedenen Varianten von CSP 8 bei 1000 Durchläufen dargestellt.*

Für das ursprüngliche Problem werden sechs statt zu erwartende fünf Constraints benötigt. Das liegt daran, dass der verwendete Choco-Solver das Constraint c_3 *in zwei Constraints* $c_{3.1} = (x_{new} = (x_1 * x_2))$ *und* $c_{3.2} = (x_{new} > x_3)$ *aufteilt. Aus diesem Grund werden auch acht anstatt zu erwartender sieben Variablen benötigt, da die Variable* x_{new} *zum CSP hinzugefügt wird.*

	Variante	$t(create_P)$	$t(sub)$	$t(solve(P^t))$	$t(total)$	#Vars	#Cons
1	CSP 8	0,1710	0,0000	0,0160	0,1870	8	6
2	$B(\{\{c_1\}^K, \{c_2\}^K, \{c_3\}^K, \{c_4\}^K, \{c_5\}^K\})$	0,1703	0,7666	1,8173	2,7542	35	69354
3	$B(\{\{c_1, c_2, c_3\}^K, \{c_4\}^K, \{c_5\}^K\})$	0,1710	0,7821	2,1905	3,1436	35	69330
4	$B(\{\{c_1\}^K, \{c_2\}^K, \{c_3\}^K\})$	0,1704	0,0337	0,0105	0,2145	22	164
5	$B(\{\{c_1, c_2, c_3\}^K\})$	0,1709	0,0246	0,0115	0,2070	22	140
6	$B(\{\{c_1\}^K, \{c_2\}^K, \{c_3\}^K, \{c_4\}^K\})$	0,1765	0,7701	1,5981	2,5447	42	69348
7	$B(\{\{c_1, c_2, c_3\}^K, \{c_4\}^K\})$	0,1738	0,7309	1,7539	2,6587	42	69324
8	$B(\{\{c_1\}^K, \{c_2\}^K, \{c_3\}^K, \{c_5\}^K\})$	0,1699	0,0546	0,0109	0,2354	38	214
9	$B(\{\{c_1, c_2, c_3\}^K, \{c_5\}^K\})$	0,1711	0,0482	0,0115	0,2307	38	190

Tabelle 5.2: Zeitaufwände für verschiedene Varianten der konfliktbasierten booleschen Skalarisierung.

Aus Tabelle 5.2 lässt sich entnehmen, dass das ursprüngliche CSP (Zeile 1) am schnellsten gelöst werden kann (bezogen auf die Gesamtzeit $t(total)$). Auffallend ist, dass im Gegensatz zur Tabularisierung und Regularisierung die mittels konfliktbasierter boolescher Skalarisierung umgeformten Varianten die reine Lösungszeit $t(solve(P^t))$ nicht immer verringern. Es kann festgestellt werden, dass die Lösungsdauer genau dann stark erhöht ist, wenn die Anzahl der erzeugten Constraints sehr hoch ist (>60.000) (Zeilen 2, 3, 6 und 7). Anderseits ist allerdings auch zu erkennen, dass sich die reine Lösungszeit verringern kann (Zeilen 4,5,8 und 9). Das betrifft in diesem Fall alle Varianten, bei denen das Count-Constraint c_4 nicht transformiert wurde. Ein Indikator dafür, ob eine konfliktbasierte boolesche Skalarisierung zu einer Verringerung der Lösungszeit führt, scheint die Anzahl der Constraints des transformierten CSPs zu sein. Wenn die Anzahl der Constraints klein ist, verringert sich der Aufwand für das Lösen des booleschen Scalar-*CSPs in der Regel deutlich.*

Das Beispiel 5.1 verdeutlicht, dass sowohl die hohe Umwandlungszeit in als auch die Lösungsdauer von booleschen *Scalar*-CSPs zu einer erheblichen Verlangsamung des Lösungsvorgangs führen kann. In dem folgenden Abschnitt 5.2 wird allerdings noch gezeigt, dass sowohl schnellere Umwandlungen in boolesche *Scalar*-CSPs als auch Umwandlungen in boolesche *Scalar*-CSPs mit weniger Constraints möglich sind. Eine Verringerung der Constraint-Anzahl führt potentiell zu einer Verringerung der Lösungszeit des transformierten booleschen *Scalar*-CSPs.

Eine weitere Optimierung, die noch nicht berücksichtigt wurde, ist das Verwenden eines spezialisierten Solvers für boolesche *Scalar*-CSPs. Es wird davon ausgegangen, dass die Verwendung eines solchen Solvers zusätzlich zu einer signifikanten Verringerung der Lösungszeit führen kann. Es wurde an dieser Stelle auf den Einsatz eines solchen spezialisierten Solvers verzichtet, da diese in der Regel nur schnell entscheiden, ob ein Problem lösbar ist oder nicht, bzw. für ein Optimierungsproblem eine optimale Lösung suchen können. Der Picat-Solver [165] beispielsweise erstellt beim Suchen nach allen

Lösungen eines CSPs P mittels boolescher Variablen nach jedem Finden einer Lösung ein neues CSP P', welches die bisher gefundenen Lösungen ausschließt. Ein Vergleich mit den bisherigen Testläufen, die nach allen Lösungen suchen, wäre dadurch an dieser Stelle schwierig.

Die Support-basierte boolesche Skalarisierung

Es wird erneut von einem CSP bzw. Teil-CSP $P = (X, D, C)$ mit $X = \{x_1, ..., x_n\}$, $D = \{D_1, ..., D_n\}$ und $C = \{c_1, ..., c_m\}$ als Ausgangspunkt für die neu entwickelte Support-basierte boolesche Skalarisierung ausgegangen. Diese Substituierung verfolgt die gleiche Grundidee wie das Support-Encoding (siehe Abschnitt 3.2): für jede gültige Belegung eines Constraints (oder eines CSPs) wird ein neues Constraint angelegt. Darüber hinaus wird bei der Support-basierten booleschen Skalarisierung für jedes solches Constraint eine boolesche Variable angelegt, die widerspiegelt, ob die entsprechende gültige Belegung angenommen wird oder nicht. Ein Vorteil gegenüber dem Support-Encoding ist, dass die Support-basierte boolesche Skalarisierung auch zum Substituieren von nicht binären Constraints verwendet werden kann.

Bei der Support-basierten booleschen Skalarisierung wird für jedes gültige Tupel $t \in T$ jedes ursprünglichen Constraints $c = (X', T) \in C, X' \subseteq X$ ein boolesches *Scalar*-Constraint angelegt. Die Umformung der Variablen in boolesche Variablen erfolgt genauso wie bei dem direkten Encoding 5.1 bzw. wie bei der konfliktbasierten booleschen Skalarisierung. Die *oneOff*-Constraints werden ebenfalls genauso wie bei der konfliktbasierten booleschen Skalarisierung erstellt.

Für jedes Tupel $t_k \in T$ jedes ursprünglichen Constraints $c_l = (X', T) \in C$ wird bei dieser Transformation eine zusätzliche boolesche Variable $x^C_{l,k}$ eingefügt, der genau dann der Wert 0 zugeordnet ist, wenn die partielle Belegung der Variablen X' das Constraint c_l mit Tupel t_k erfüllt. Andernfalls wird der Variablen $x^C_{l,k}$ der Wert 1 zugeordnet.

Des Weiteren muss für jedes ursprüngliche Constraint $c_l \in C$ ein neues Constraint erzeugt werden, das garantiert, dass nur eine der $x^C_{l,k}$ booleschen Variablen den Wert 0 annimmt. Die beschriebenen Constraints müssen dabei als *Scalar*-Constraints über boolesche Variablen formuliert werden.

Definition 5.2: ***Die Support-basierte boolesche Skalarisierung.*** Sei ein CSP $P = (X, D, C)$ gegeben, seine Support-basierte boolesche Skalarisierung sei definiert durch $P^{SBS} = (X^B, D^B, C^S)$ mit:

- $X^B = \{x^B_{i,j} \mid i \in \{1, ..., |X|\}, j \in \{1, ..., |D_i|\}\}$
  $\cup\{x^C_{l,k} \mid l \in \{1, ..., |C|\}, k \in \{1, ..., |T_l|\}, c_l = (X_l, T_l)\}$
  ist eine Menge von booleschen Variablen, wobei jede Variable $x^B_{i,j}$ widerspiegelt, ob die ursprüngliche Variable $x_i \in X$ gleich den j-ten Wert der Domäne $D_i \in$

D annimmt oder nicht ($x_i = d_j \leftrightarrow x^B_{i,j} = 1$). Zusätzlich wird für jedes gültige Tupel $t_k = (v_1, ..., v_n) \in T_l$ jedes Constraints $c_l = (X_l, T_l) \in C$ mit $X_l = \{x_1, ..., x_n\}$ des ursprünglichen CSPs eine boolesche Variable angelegt, deren inverser Wert widerspiegelt, ob das entsprechende Tupel erfüllt ist oder nicht ($x^C_{l,k} = 0 \leftrightarrow x_1 = v_1 \wedge ... \wedge x_n = v_n$).

- $D^B = \{D_{i,j} = D_{l,k} = \{0,1\} \,|\, i \in \{1, ..., |X|\}, j \in \{1, ..., |D_i|\}, l \in \{1, ..., |C|\}, k \in \{1, ..., |T_l|\}, c_l = (X_l, T_l)\}$ ist die Menge der booleschen Domänen der neu erzeugten Variablen.
- $C^S = \{c^{tupel}_{l,k} | l \in \{1, ..., |C|\}, k \in \{1, ..., T_l\}, c_l = (X_l, T_l)\}$
  $\cup \{c^{oneNot}_l \,|\, l \in \{1, ..., |C|\}\}$
  $\cup \{c^{oneOff}_i \,|\, i \in \{1, ..., |X|\}\}$
  ist die Menge der Constraints über den neu erzeugten booleschen Variablen X^B. Für jedes gültige Tupel $t_k \in T_l$ jedes ursprünglichen Constraints $c_l = (X_l, T_l) \in C$, mit $X_l = \{x_1, ..., x_n\}$, kann ein Constraint $c_{l,k} \in C^{Scalar}$ in Form eines *Scalar*-Constraints erzeugt werden. Das Tupel $t_k = (v_1, ..., v_n) \in T_l$ repräsentierende *Scalar*-Constraint ist definiert durch $c^{tupel}_{l,k} = Scalar(\{x'_1, ..., x'_n, x^C_{l,k}\}, (1, ..., 1, n)^T, \geq, n)$ mit x'_i ist gleich der booleschen Variablen $x^B_{i,j} \in X^B$, die die Zuordnung $\phi(x_i) = d_j = v_i$ repräsentiert.

  Die *Scalar*-Constraints $c^{oneNot}_l = Scalar(\{x^C_{l,1}, ..., x^C_{l,|T_l|}\}, (1, ..., 1)^T, =, |T_l| - 1) \mid \forall l \in \{1, ..., |C|\}$ realisieren für das boolesche *Scalar*-CSP P^{SBS}, dass für jedes Constraint $c_l \in C$ des ursprünglichen CSPs P nur ein zulässiges Tupel $t_k \in T_l$ gleichzeitig erfüllt ist.

  Damit die Variablen $X^B_i = \{x^B_{i,j} \mid \forall j \in \{1, ..., |D_i|\}\}$ die möglichen Variablenzuweisungen für x_i repräsentieren können, muss gewährleistet werden, dass exakt eine Variable in X^B_i den Wert 1 annimmt. Dafür werden die *oneOff*-Constraints $C^{oneOff} = \{Scalar(X^B_i, (1, ..., 1)^T, =, 1) \mid \forall i \in \{1, ..., n\}\}$ angelegt.

Jedes Constraint $c^{tupel}_{l,k}$ mit $l \in \{1, ..., |C|\}, k \in \{1, ..., T_l\}$ und $c_l = (X_l, T_l)$ kann nur dann erfüllt sein, wenn entweder alle booleschen Variablen $x^B_{i,j}$ für $i \in \{1, ..., |X_l|\}$ mit $(x^B_{i,j} = 1) \leftrightarrow (x_i = d_j = v_i), d_j \in D_j, (v_1, .., v_{|X_l|}) = t_k \in T_l$ den Wert 1 zugeordnet bekommen oder die Tupelvariable $x^C_{l,k}$ den Wert 1 zugewiesen bekommt. Der erste Fall ist gleichbedeutend damit, dass die ursprünglichen Variablen $x_1, ..., x_{|X_l|} \in X_l$ die Werte $v_1, ..., v_{|X_l|}$ annehmen und das ursprüngliche Constraint c_l somit mit dem Tupel t_k erfüllt ist. Im zweiten Fall könnte das ursprüngliche Constraint c_l auf den ersten Blick trotzdem mit dem Tupel t_k erfüllt sein. Jedes Constraint kann aber gleichzeitig nur mit einem Tupel erfüllt sein. Von den dem Constraint c_l zugeordneten Variablen $x^C_{l,1}, ..., x^C_{l,|T_l|}$, bekommt aufgrund des c^{oneNot}_l-Constraints nur eine den Wert 0 zugeordnet. Aufgrund der Constraints $c^{tupel}_{l,1}, ..., c^{tupel}_{l,|T_l|}$ und des c^{oneNot}_l-Constraints ergibt sich somit die Äquivalenz, dass das ursprüngliche Constraint c_l genau dann mit Tupel t_k erfüllt ist, wenn die Variable $x^C_{l,k}$ den Wert 0 zugeordnet bekommt.

Durch die angegebene Umwandlung lassen sich alle FD-CSPs in boolesche *Scalar*-CSPs überführen, allerdings entstehen dabei sehr viele Variablen und Constraints. Allein für jedes gültige Tupel jedes Constraints des ursprünglichen CSPs wird eine neue Variable $x_{l,k}^{C}$ und ein neues Constraint $c_{l,t}^{tupel}$ angelegt. Zusätzlich wird für jedes Constraint $c_l \in C$ des ursprünglichen CSPs P ein Constraint $c_l \in C^{Scalar}$ angelegt, welches realisiert, dass nur ein Tupel pro Constraint erfüllt ist. Ebenso wird für jede Variable $x_i \in X$ des ursprünglichen CSPs ein Constraint $c_i \in C^{Scalar}$ angelegt, das realisiert, dass nur eine Wertezuordnung pro Variable gleichzeitig gültig ist.

Um die Anzahl der Variablen und Constraints zu reduzieren, empfiehlt es sich in manchen Fällen, nicht jedes gültige Tupel T jedes Constraints C einzeln umzuwandeln, sondern stattdessen nur jedes gültige Tupel T' eines Teil-CSPs P' von P umzuwandeln. Hat das Teil-CSP $P' = (X', D', C')$ weniger gültige Belegungen als gültige Belegungen in den einzelnen Constraints C' existieren, so reduziert sich die Anzahl der Variablen und Constraints im booleschen *Scalar*-CSP dementsprechend.

Sollen nur Teil-CSPs $P' = (X', D', C')$ oder einzelne Constraints C' in boolesche *Scalar*-CSPs umgewandelt werden, so müssen die ursprünglichen Variablen X erhalten bleiben, die Constraints C' entfernt, die neuen Variablen und Constraints hinzugefügt und die ursprünglichen Variablen mit den neuen booleschen Variablen verknüpft werden. Für die Verknüpfung der ursprünglichen Variablen mit den neuen booleschen Variablen werden die aus der konfliktbasierten booleschen Skalarisierung bereits bekannten Verknüpfungs-Constraints $C^{binding} = \{x_i = d_j \leftrightarrow x_{i,j}^{B} = 1 \mid \forall i \in \{1, ..., |X'|\}, j \in \{1, ..., |D_i|\}\}$ verwendet.

In Beispiel 5.2 wird an Hand von CSP 8 gezeigt, wie die Anzahl der Variablen und Constraints im booleschen *Scalar*-CSP mittels Umwandlung von Teil-CSPs verringert werden kann.

Beispiel 5.2: Die Support-basierte boolesche Skalarisierung[2]. *Gegeben sei das CSP 8 aus den Beispielen 4.1 und 4.2, welches mittels Support-basierter boolescher Skalarisierung transformiert werden soll. Die Variablen und Domänen des ursprünglichen CSPs werden in boolesche Variablen umgewandelt, wobei jede neue Variable* $x_{i,j}^{B}$ *repräsentiert, ob die ursprüngliche Variable* $x_i \in X$ *den j-ten Wert aus der Domäne* D_i *annimmt oder nicht. Es ergeben sich somit die neuen Variablen* $X^B = \{x_{i,j}^{B} \mid \forall i \in \{1, ..., 7\}, j \in \{1, ..., 5\}\}$ *mit Domänen* $D_{i,j} = \{0, 1\}, \forall i \in \{1, ..., 7\}, j \in \{1, ..., 5\}$. *Für die neu erzeugten Variablen müssen neue Constraints angelegt werden, die die alten adäquat ersetzen und vom Typ* Scalar-*Constraint sind.*

Um beispielsweise das Constraint $c_2 = (\{x_1, x_2, x_3\}, T_2)$ *zu ersetzen, muss für jedes gültige Tupel* $t_1, ..., t_{10} \in T_2$ *eine neue boolesche Variable* $x_{2,1}^{C}, ..., x_{2,10}^{C}$ *angelegt werden, die repräsentiert, ob die ursprünglichen Variablen* $\{x_1, x_2, x_3\}$ *die Belegung* t_i *annehmen* ($x_{2,i}^{C} = 0$) *oder nicht* ($x_{2,i}^{C} = 1$).

---

[2] Zum Lösen aller im Beispiel erzeugten CSPs wurde jeweils der Choco-Solver mit der in Abschnitt 1.4 angegebenen Suchstrategie, Software und Hardware verwendet.

Zu beachten ist die gegensätzliche Beschreibung: das Constraint c_j ist genau dann mit Tupel t_i erfüllt, wenn $x^C_{j,i}$ gleich 0 ist.

Für c_2 ergeben sich somit die zehn Scalar-*Constraints:*

$$\begin{aligned}
c_{2,1}^{tupel} &= \text{Scalar}(\{x^B_{1,1}, x^B_{2,1}, x^B_{3,2}, x^C_{2,1}\}, (1,1,1,3)^T, \geq, 3),\\
c_{2,2}^{tupel} &= \text{Scalar}(\{x^B_{1,1}, x^B_{2,2}, x^B_{3,3}, x^C_{2,2}\}, (1,1,1,3)^T, \geq, 3),\\
c_{2,3}^{tupel} &= \text{Scalar}(\{x^B_{1,1}, x^B_{2,3}, x^B_{3,4}, x^C_{2,3}\}, (1,1,1,3)^T, \geq, 3),\\
c_{2,4}^{tupel} &= \text{Scalar}(\{x^B_{1,1}, x^B_{2,4}, x^B_{3,5}, x^C_{2,4}\}, (1,1,1,3)^T, \geq, 3),\\
c_{2,5}^{tupel} &= \text{Scalar}(\{x^B_{1,2}, x^B_{2,1}, x^B_{3,3}, x^C_{2,5}\}, (1,1,1,3)^T, \geq, 3),\\
c_{2,6}^{tupel} &= \text{Scalar}(\{x^B_{1,2}, x^B_{2,2}, x^B_{3,4}, x^C_{2,6}\}, (1,1,1,3)^T, \geq, 3),\\
c_{2,7}^{tupel} &= \text{Scalar}(\{x^B_{1,2}, x^B_{2,3}, x^B_{3,5}, x^C_{2,7}\}, (1,1,1,3)^T, \geq, 3),\\
c_{2,8}^{tupel} &= \text{Scalar}(\{x^B_{1,3}, x^B_{2,1}, x^B_{3,4}, x^C_{2,8}\}, (1,1,1,3)^T, \geq, 3),\\
c_{2,9}^{tupel} &= \text{Scalar}(\{x^B_{1,3}, x^B_{2,2}, x^B_{3,5}, x^C_{2,9}\}, (1,1,1,3)^T, \geq, 3) \textit{ und}\\
c_{2,10}^{tupel} &= \text{Scalar}(\{x^B_{1,4}, x^B_{2,1}, x^B_{3,5}, x^C_{2,10}\}, (1,1,1,3)^T, \geq, 3).
\end{aligned}$$

Zusätzlich muss ein Constraint $c_2^{oneNot} = \text{Scalar}(\{x^C_{2,1}, \ldots, x^C_{2,10}\}, (1, \ldots, 1)^T, =, 9)$ *erzeugt werden, das garantiert, dass nur eine Lösung für c_2 gleichzeitig gültig ist. Zur Repräsentation von c_2 sind demzufolge 11* Scalar-*Constraints notwendig. Für die anderen Constraints aus C wird analog verfahren. Des Weiteren muss für jede Variable $x_i \in X$ ein* Scalar-*Constraint angelegt werden, das garantiert, dass nur eine Variablenbelegung für x_i gleichzeitig möglich ist, also nur eine Variable $x_{i,1}, \ldots, x_{i,5}$ den Wert 1 annimmt für $i \in \{1, \ldots, 7\}$. Daraus ergeben sich die* Scalar-*Constraints* $c_i^{oneOff} = \text{Scalar}(\{x_{i,1}, \ldots, x_{i,5}\}, (1,1,1,1,1)^T, =, 1), \forall i \in \{1, \ldots, 7\}$. *Insgesamt entstehen so für das gegebene Problem 9831 Variablen und 9808 Constraints*[3].

In Tabelle 5.3 sind die durchschnittlichen Zeitaufwände für das Erstellen $t(create_P)$, Transformieren $t(sub)$ und Lösen $t(solve(P^t))$ sowie die akkumulierte Zeit $t(total)$, die Anzahl der benötigten Variablen und Constraints für die aus Abschnitt 5.1.2 bekannten Variationen von CSP 8 bei 1000 Durchläufen mit der soeben vorgestellten Substituierung dargestellt.

Es ist zu erkennen, dass die Umwandlung von Constraints, die viele gültige Tupel haben (c_4), die Erzeugung sehr vieler Variablen und Constraints nach sich ziehen, die das System verlangsamen (Zeilen 2, 3, 6 und 7). Werden nur die Constraints umgewandelt, welche nur über wenige gültige Tupel verfügen, so bleibt die Variablen- und Constraint-Anzahl deutlich geringer (z.B. Zeilen 4 und 5). Des Weiteren ist erkennbar, dass die Umwandlung des Teil-CSPs $P' = (\{x_1, x_2, x_3\}, \{D_1, D_2, D_3\}, \{c_1, c_2, c_3\})$ im Ganzen sowohl die Anzahl der erzeugten

---

[3] Die tatsächliche Anzahl der Variablen (9831) und Constraints (9808) entspricht nicht der erwarteten Anzahl (9713 Variablen und 9690 Constraints). Die Abweichung entsteht dadurch, dass der Choco-Solver sehr große lineare Constraints, unter Erzeugung neuer Variablen, in kleinere Constraints aufteilt. Aus diesem Grund werden für das *Count*-Constraint 94 und für das *Sum*-Constraint 24 zusätzliche Variablen und Constraints benötigt.

	Variante	$t(create_P)$	$t(sub)$	$t(solve(P^t))$	$t(total)$	#Vars	#Cons
1	CSP 8	0,1710	0,0000	0,0160	0,1870	8	6
2	$B(\{\{c_1\}^S, \{c_2\}^S, \{c_3\}^S, \{c_4\}^S, \{c_5\}^S\})$	0,1723	0,4455	2,1171	2,7349	9831	9808
3	$B(\{\{c_1, c_2, c_3\}^S, \{c_4\}^S, \{c_5\}^S\})$	0,1714	0,4782	1,0276	1,6771	9705	9680
4	$B(\{\{c_1\}^S, \{c_2\}^S, \{c_3\}^S\})$	0,1708	0,0428	0,0116	0,2251	150	148
5	$B(\{\{c_1, c_2, c_3\}^S\})$	0,1762	0,0278	0,0020	0,2060	14	10
6	$B(\{\{c_1\}^S, \{c_2\}^S, \{c_3\}^S, \{c_4\}^S\})$	0,1772	0,4386	1,3586	1,9744	9224	9222
7	$B(\{\{c_1, c_2, c_3\}^S, \{c_4\}^S\})$	0,1774	0,4383	1,0506	1,6663	9098	9094
8	$B(\{\{c_1\}^S, \{c_2\}^S, \{c_3\}^S, \{c_5\}^S\})$	0,1705	0,1004	0,0557	0,3266	784	782
9	$B(\{\{c_1, c_2, c_3\}^S, \{c_5\}^S\})$	0,1711	0,0926	0,0274	0,2910	648	644

Tabelle 5.3: Zeitaufwände für verschiedene Varianten der Support-basierten booleschen Skalarisierung.

booleschen Variablen und Constraints als auch die Lösungsdauer ($t(solve(P^t))$) reduziert (Vergleich der Zeilen 2 mit 3, 4 mit 5, 6 mit 7 und 8 mit 9).

Beim Vergleich des Support-basierten Verfahrens mit dem konfliktbasierten booleschen Skalarisierungsverfahren lassen sich die folgenden Punkte feststellen:

- Die Anzahl der Constraints konnte bei dem Support-basierten Ansatz im Vergleich zum konfliktbasierten verringert werden, gleichzeitig hat sich allerdings die Anzahl der Variablen erhöht.
- Sowohl die konfliktbasierte als auch die Support-basierte boolesche Skalarisierung führen bei diesem Beispiel zu einer Erhöhung der Gesamtlösungsdauer $t(total)$.
- Bei beiden Verfahren ist die Variante, die die Constraints c_1, c_2 und c_3 als ein Teil-CSP substituiert und die Constraints c_4 und c_5 im Originalen behält, die schnellste boolesche *Scalar* Version.
- Sowohl die Umwandlung des *Count*-Constraints c_4 als auch die des *Sum*-Constraints c_5 verlangsamen den Lösungsprozess des CSPs.
- Im Fall von $B(\{\{c_1, c_2, c_3\}^S\})$ konnte mit der Support-basierten booleschen Skalarisierung die reine Lösungsdauer $t(solve(P^t))$ signifikant reduziert werden (um 87,5%, Zeile 5 gegenüber Zeile 1 in Tabelle 5.3).
- Im Beispiel ist in den meisten Fälle die Support-basierte boolesche Skalarisierung schneller als die konfliktbasierte. Dies ist zum Beispiel der Fall, wenn die Zeilen 2 bis 7 von Tabelle 5.2 mit den gleichen Zeilen in Tabelle 5.3 verglichen werden. Bei den Fällen $B(\{\{c_1\}^{(K|S)}, \{c_2\}^{(K|S)}, \{c_3\}^{(K|S)}, \{c_5\}^{(K|S)}\})$ (jeweils Zeile 8) und $B(\{\{c_1, c_2, c_3\}^{(K|S)}, \{c_5\}^{(K|S)}\})$ (jeweils Zeile 9) verhält es sich allerdings umgekehrt. Dies lässt sich damit erklären, dass die Anzahl der gültigen Tupel des *Sum*-Constraints

c_5 wesentlich größer ist, als die Anzahl der ungültigen Tupel (590 gültige gegenüber 35 ungültigen). Ist die Anzahl der gültigen Tupel im Verhältnis zur Anzahl der ungültigen Tupel hoch, so bietet sich das konfliktbasierte und umgekehrt das Support-basierte Verfahren an.

Auch wenn die Gesamtdauer des Lösungsprozesses für das gegebene Beispiel durch die beiden Varianten der booleschen Skalarisierung nicht reduziert werden konnte, konnte die reine Lösungsgeschwindigkeit, bei richtiger Auswahl der zu substituierenden Constraints, deutlich reduziert werden. Da, wie gezeigt wurde, sich die reine Lösungszeit zum Lösen eines CSPs durch boolesche Skalarisierung verringern kann, bietet sich dieses Verfahren besonders für CSPs an, bei denen die reine Lösungszeit einen erheblich größeren Teil der benötigten Gesamtzeit ausmacht, als die Zeit, die für das Erstellen des CSPs benötigt wird.

Für die boolesche Skalarisierung ergibt sich neben der Problematik der zeitaufwendigen Transformation zusätzlich das Problem, dass, wenn sehr viele Constraints, und im Fall der Support-basierten booleschen Skalarisierung zusätzlich sehr viele Variablen, erzeugt werden, diese die reine Zeit zum Lösen des CSPs deutlich erhöhen können. Es ist bei der booleschen Skalarisierung also sowohl darauf zu achten, dass die Transformation wenig Zeit in Anspruch nimmt, als auch, dass das Ergebnis der Substituierung möglichst kompakt ist.

5.2 Die Substituierung spezieller globaler Constraints

Bisher besteht bei allen in Abschnitt 5.1 vorgestellten Substituierungen das Problem, dass die Transformation der Constraints in entsprechende FD-vollständige Constraints zu zeitaufwendig ist, als dass eine signifikante Steigerung der Lösungsgeschwindigkeit eines CSPs erreicht werden kann. Bis zu diesem Punkt wurden allerdings auch nur sehr allgemeine Verfahren für die Transformation vorgestellt. Diese können zwar immer eingesetzt werden, führen aber bei allen Constraint-Arten zu hohen Transformationszeiten. Bei der booleschen-Skalarisierung ergibt sich zusätzlich das Problem, dass durch die bisher vorgestellten Substituierungen ein System mit sehr vielen Constraints und beim zweiten Verfahren auch sehr vielen Variablen entstehen kann, obwohl kompaktere Repräsentationen existieren.

Aufgrund der hohen Transformationszeiten sind die bisher vorgestellten Transformationen in der Regel ungeeignet für globale Constraints. Im Folgenden wird diskutiert, wie für ausgewählte globale Constraints Substituierungen realisiert werden können.

Aufgrund der unterschiedlichen Datenstrukturen und Algorithmen, die bei den verschiedenen globalen Constraints verwendet werden, empfiehlt es sich, individuell an die jeweiligen Constraints angepasste Substituierungen vorzunehmen. In den folgenden Abschnitten wird jeweils ein Constraint c benannt und eine Möglichkeit der Regularisierung und der booleschen Skalarisierung für dieses angegeben. Für alle nicht aufgeführten Constraints können die zuvor vorgestellten Verfahren zur Substituierung verwendet werden, sodass für jedes Constraint mindestens eine Möglichkeit besteht, dieses in ein *Table-*, *Regular-* oder boolesches *Scalar*-Constraint zu überführen. Da für die Tabularisierung, wie sie im Kontext dieser Arbeit verstanden wird, immer alle gültigen Tupel ermittelt und dem Propagator im Konstruktor übergeben werden müssen, ergibt sich an dieser Stelle für die Tabularisierung kein Vorteil gegenüber den bereits in Abschnitt 5.1 aufgeführten Verfahren. Bei der Regularisierung und der booleschen Skalarisierung wird nachfolgend allerdings auf den Schritt der vollständigen Tupelauflistung verzichtet. Im Folgenden wird daher auf die direkte Regularisierung und boolesche Skalarisierung von speziellen globalen Constraints eingegangen.

Im weitere Verlauf wird ein CSP $P = (X', D, C)$ mit jeweils einem zu substituierenden Constraint $c = (X, R) \in C$ mit $X = \{x_1, ..., x_n\} \subseteq X'$ betrachtet. Des Weiteren wird bei der booleschen Skalarisierung davon ausgegangen, dass die booleschen Variablen $X^B = \{x^B_{i,j} \mid i \in \{1, ..., n\}, j \in \{1, ..., |D_i|\}\}$, wie in den vorherigen Abschnitten beschrieben, für alle Variablen X von c erstellt werden und im Folgenden verwendet werden können. Dafür müssen die beiden Voraussetzungen 5.1 und 5.2 unabhängig von ihrer technischen Umsetzung gelten:

V 5.1: $x_i = d_j \leftrightarrow x^B_{i,j} = 1, \forall x_i \in X, x_{i,j} \in X^B, i \in \{1, ..., n\}, d_j \in D_i, j \in \{1, ..., |D_i|\}$

V 5.2: $\sum_{j=1}^{|D_i|} x^B_{i,j} = 1, \forall i \in \{1, ..., n\}$

Die Voraussetzung 5.1 gewährleistet die Äquivalenz der Wertzuordnung von Wert d_j zur ursprünglichen Variable x_i mit der Wertzuordnung von Wert 1 zur booleschen Variable $x^B_{i,j}$. Voraussetzung 5.2 gewährleistet hingegen, dass genau eine der booleschen Variablen $x^B_{i,j}$, die die ursprüngliche Variable x_i repräsentieren, den Wert 1 annehmen muss. Damit ist sichergestellt, dass die ursprüngliche Variable x_i nicht mehrere oder keinen Domänenwert zugeordnet bekommt.

Die in den folgenden Abschnitten geführten Beweise basieren darauf, dass die Voraussetzungen 5.1 und 5.2 erfüllt sind und unterteilen sich jeweils in zwei Teile. Der erste Teil beweist jeweils, dass aus der Erfüllung des gegebenen Constraints c auch die Erfüllung der angegebenen Substituierung $sub(c)$ folgt ($c \Rightarrow sub(c)$). Der zweite Teil zeigt hingegen jeweils, dass aus der Erfüllung der Substituierung von c auch die Erfüllung von c folgt ($sub(c) \Rightarrow c$).

5.2.1 Das *Count*-Constraint

Das Constraint $c = Count(\{x_1, ..., x_n\}, occ, v)$ mit $x_i, occ \in X, \forall i \in \{1, ..., n\}, v \in \mathbb{N}$ garantiert, dass die Anzahl der Variablen in $x_1, ..., x_n$, denen der Wert v zugeordnet wird, dem Wert der Variablen *occ* entspricht, das heißt der Wert v wird den Variablen aus $\{x_1, \ldots, x_n\}$ *occ*-mal zugeordnet.

Regularisierung

In der in Zusammenhang mit dieser Arbeit entstandenen Veröffentlichung [95] und auch im Global Constraint Catalog [1] wurde jeweils eine Möglichkeit für die Überführung eines *Count*-Constraints in einen DFA aufgeführt. Grundidee dabei ist es $m = max(D_{occc}) + 1$ viele Zustände $Q = \{q_0, ..., q_m\}$ zu erzeugen. Jeder Zustand $q_i \in Q$ kann genau dann erreicht werden, wenn der Wert v i-mal eingelesen wurde. Der Zustand q_0 ist der Startzustand und alle Zustände q_i mit $i \in D_{occ}$ sind akzeptierende Zustände. Wird der so erzeugte DFA mit einem DFA geschnitten, der alle Wörter der Länge n akzeptiert und anschließend minimiert, so entsteht ein minimaler levelbasierter DFA M^{Count}, der im Constraint $Regular(\{x_1, ..., x_n\}, M^{Count})$ als Regularisierung für das ursprüngliche *Count*-Constraint c verwendet werden kann.

Boolesche Skalarisierung

Es wird davon ausgegangen, dass alle ursprünglichen Domänen $D_1, ..., D_n$ der am *Count*-Constraint beteiligten Variablen $x_1, ..., x_n$ den Wert v enthalten. Anderenfalls könnten die Variablen, deren Domänen den Wert v nicht enthalten, aus dem *Count*-Constraint entfernt werden und mit einem reduzierten *Count*-Constraint fortgefahren werden.

Wenn die i-te Variable $x_i \in X$ des ursprünglichen CSPs den j-ten Wert v_j ihrer Domäne D_i annimmt, bedeutet das, dass in der zugehörigen booleschen Darstellung der Variable $x^B_{i,j} \in X^B$ der Wert 1 zugeordnet wird. Nimmt x_i den j-ten Wert nicht an, so nimmt $x^B_{i,j} \in X^B$ den Wert 0 an. Folglich müssen zur Repräsentation des Constraints $c = Count(\{x_1, ..., x_n\}, occ, v)$ in der booleschen Darstellung v_{occ} viele Variablen in $\{x^B_{i,j_i} \mid \forall i \in \{1, ..., n\}, v = d_{j_i} \in D_i\}$ den Wert 1 annehmen, wobei v_{occ} gerade der Wert ist, welcher der Variablen *occ* zugeordnet wird.

In Abhängigkeit von *occ* ergeben sich die zwei folgenden Varianten der booleschen Skalarisierung.

Fall 1: Die Variable *occ* hat nur einen Wert in der Domäne: $D_{occ} = \{d\}$.

Das Constraint c^{const}, welches die Summe $1 * x^B_{1,j_1} + 1 * x^B_{2,j_2} + ... + 1 * x^B_{n,j_n} = d$ ausdrückt, kann als Substituierung für das ursprüngliche *Count*-Constraint c verwendet werden.

$$c^{const} = Scalar(\{x^B_{1,j_1}, ..., x^B_{n,j_n}\}, (1, ..., 1)^T, =, d) \text{ mit } v = d_{j_i} \in D_i \; \forall i \in \{1, ..., n\}, j_i \in \{1, ..., |D_i|\} \tag{5.8}$$

Beweis 5.1. *Im Folgenden wird zunächst nachgewiesen, dass das* Scalar-*Constraint c^{const} erfüllt sein muss, wenn das* Count-*Constraint c erfüllt ist* (Count → *sub*(Count)). *Anschließend wird* *gezeigt, dass das* Count-*Constraint c auch erfüllt sein muss, wenn das* Scalar-*Constraint* c^{const} *erfüllt ist (sub*(Count) → Count).

Teil 1 *(c → sub(c)): Ohne Beschränkung der Allgemeinheit sei bei erfülltem* Count-*Constraint c den ersten d Variablen* $x_1, ..., x_d$ *der Wert v zugeordnet (anderenfalls können die Indizes entsprechend vergeben werden), wobei d gleich der Wert ist, welcher der Variablen occ zugeordnet ist. Aufgrund des geltenden Zusammenhangs* $x_i = v \leftrightarrow x^B_{i,j_i} = 1, \forall i \in \{1, ..., n\}, d_{j_i} = v, d_{j_i} \in D_i$, *folgt, dass den booleschen Variablen* $x^B_{1,j_1}, ..., x^B_{d,j_d}$ *der Wert 1 und den booleschen Variablen* $x^B_{d+1,j_{d+1}}, ..., x^B_{n,j_n}$ *der Wert 0 zugeordnet wird. Demzufolge ist die Summe der Variablen* $x^B_{i,j_i}, \forall i \in \{1, ..., n\}$ *gleich d und das* Scalar-*Constraint* c^{const} *ist erfüllt.* □

Teil 2 *(sub(c) → c): Ohne Beschränkung der Allgemeinheit sei bei erfülltem* Scalar-*Constraint* c^{const} *den ersten d Variablen* $x^B_{1,j_1}, ..., x^B_{d,j_d}, \forall d_{j_i} = v$ *der Wert 1 zugeordnet (anderenfalls können die Indizes entsprechend vergeben werden), wobei d gleich der Wert ist, welcher der Variablen occ zugeordnet ist. Aufgrund des geltenden Zusammenhangs* $x_i = v \leftrightarrow x^B_{i,j_i} = 1, \forall i \in \{1, ..., n\}, d_{j_i} = v, d_{j_i} \in D_i$ *folgt, dass den ursprünglichen Variablen* $x_1, ..., x_d$ *der Wert v und den ursprünglichen Variablen* $x_{d+1}, ..., x_n$ *ein Wert ungleich v zugeordnet wird. Demzufolge kommt der Wert v in den Variablen* $x_1, ..., x_n$ *genau d mal vor und das* Count-*Constraint c ist erfüllt.* □

Fall 2: Die Variable *occ* hat mehr als einen Wert in der Domäne: $D_{occ} = \{d_{min}, ..., d_{max}\}$.

Sei der Vektor boolescher Variablen $\overrightarrow{U_{occ}} = (x^{occ}_1, ..., x^{occ}_k)^T$ mit $k = max(D_{occ})$ die Regular-Darstellung der Variable *occ* (siehe Abschnitt 3.2 zum Regular-Encoding). Damit die Regular-Variableneigenschaft erfüllt ist, muss das Constraint $c^{order} = (x^{occ}_1 \geq ... \geq x^{occ}_k)$ zum System hinzugefügt werden. Des Weiteren müssen die Regular-Variablen x^{occ}_i mit den booleschen Variablen $x^B_{occ,j}$ verbunden werden. Dabei drücken die Variablen $x^B_{occ,j}$ die möglichen Domänenwerte für x_{occ} aus.

Es ergeben sich die Constraints $C^{binding}$ wie folgt:

- Für den Fall, dass der Wert 0 in der Domäne von *occ* ist, wird das Constraint $c^{binding}_0 = (x^B_{occ,0} + x^{occ}_1 = 1)$ zu $C^{binding}$ hinzugefügt.
- Für jeden Wert $i \in D(occ), 0 < i < k$ wird das Constraint $c^{binding}_i = (x^{occ}_i - x^{occ}_{i+1} - x_{occ,i} = 0)$ zu $C^{binding}$ hinzugefügt.
- Für $i = k$ muss das Constraint $c^{binding}_k = (x_{occ,k} = x^{occ}_k)$ zu $C^{binding}$ hinzugefügt werden.

Der Variablen $x_{occ,i}$ wird durch diese Constraints genau dann der Wert 1 zugeordnet, wenn x^{occ}_i der Wert 1 und x^{occ}_{i+1} der Wert 0 zugeordnet wird (Beweis mittels Wahrheitstabelle trivial).

Unter Hinzufügen der booleschen Variablen $x^{occ}_1, ..., x^{occ}_k$ und der Constraints c^{order} und $c^{binding}_i, \forall i \in D(occ)$ dient das *Scalar*-Constraint $c^{variable}$ in Gleichung 5.9, welches die

Summe $1 * x^B_{1,j_1} + ... + 1 * x^B_{n,j_n} - 1 * x^{occ}_1 ... - 1 * x^{occ}_k = 0$ ausdrückt, als Substituierung für das ursprüngliche *Count*-Constraint c.

$$c^{variable} = Scalar(\{x^B_{1,j_1}, ..., x^B_{n,j_n}, x^{occ}_1, ..., x^{occ}_k\}, (1^n, (-1)^k)^T, =, 0) \text{ mit } v = d_{j_i} \in D_i \; \forall i \in \{1, ..., n\} \tag{5.9}$$

Anmerkung 5. Sowohl die Constraints in $C^{binding}$ als auch c^{order} lassen sich als *Scalar*-Constraints darstellen, wodurch diese Art der Transformation des *Count*-Constraints für eine vollständige boolesche Skalarisierung geeignet ist.

Beweis 5.2. ***Teil 1*** ($c \longrightarrow sub(c)$)**:** *Ohne Beschränkung der Allgemeinheit sei d ein beliebiger, aber fester Wert aus der Domäne von occ. Nach Fall 1 wurden bei einem erfüllten* Count-*Constraint c d vielen Variablen den Wert v und folglich auch d vielen Variablen in $x^B_{i,j_i}, \forall i \in \{1, ..., n\}$ der Wert 1 zugeordnet. Dementsprechend ist die Summe der $1 * x^B_{1,j_1} + ... + 1 * x^B_{n,j_n}$ gleich dem Wert d.*

Da die Variable occ mit Wert d belegt wurde, folgt aufgrund von $(x_{occ} = d) \leftrightarrow (x^B_{occ,j} = 1), d_j = d \in D(occ)$, dass der booleschen Variable $x^B_{occ,j}$ für $d_j = d$ der Wert 1 und allen anderen Variablen $x^B_{occ,j}$ für $d_j \neq d$ der Wert 0 zugeordnet wird. Aufgrund der $C^{binding}$- und C^{order}-Constraints wird allen Variablen $x^{occ}_1, ..., x^{occ}_d$ der Wert 1 und allen Variablen $x^{occ}_{d+1}, ..., x^{occ}_k$ der Wert 0 zugeordnet.

Für $0 \in D(occ)$ folgt aufgrund von $c^{binding}_0$, dass wenn d gleich 0 ist, also $x^B_{occ,0}$ gleich 1 ist, x^{occ}_1 gleich 0 sein muss. Aufgrund des Constraints c^{order} müssen dann alle weiteren Variablen $x^{occ}_i, \forall i > 1$ ebenfalls den Wert 0 annehmen.

Ist d größer als 0 und kleiner als k, so folgt aus $x_{occ,d} = 1$ und dem Constraint $c^{binding}_d = (x^{occ}_d - x^{occ}_{d+1} - x_{occ,d})$, dass x^{occ}_d der Wert 1 und x^{occ}_{d+1} der Wert 0 zugeordnet werden muss. Aufgrund des c^{order}-Constraints ergibt sich daraus, dass alle Variablen $x^{occ}_i, \forall i < d$ den Wert 1 und alle Variablen $x^{occ}_j, \forall j > d$ den Wert 0 zugeordnet bekommen.

Ist d gleich k, so folgt aus $c_{occ,d} = 1$ und $c^{binding}_k$, dass auch x^{occ}_d der Wert 1 zugeordnet werden muss. Alle anderen Variablen d^{occ}_i müssen dann aufgrund des Constraints c^{order} ebenfalls den Wert 1 zugeordnet bekommen.

Aufgrund der soeben beschriebenen Fälle ergibt sich, dass die ersten d Variablen aus $x^{occ}_1, ..., x^{occ}_k$ den Wert 1 und die anderen den Wert 0 zugeordnet bekommen. Das hat zur Folge, dass die ersten n Variablen in $c^{variable}$ immer den Wert d aufsummieren und die folgenden k Variablen den Wert d wieder subtrahieren, so dass sich in Summe der Wert 0 ergibt. Damit folgt, dass das Scalar-*Constraint $c^{variable}$ immer erfüllt ist, wenn das* Count-*Constraint c erfüllt ist ($c \longrightarrow sub(c)$).*

□

Teil 2 ($sub(c) \longrightarrow c$)**:** *Ist das* Scalar-*Constraint $c^{variable}$ erfüllt, so ist d vielen Variablen in $x^B_{i,j_i}, \forall i \in \{1, ..., n\}$ und d vielen Variablen in $x^{occ}_1, ..., x^{occ}_k$ der Wert 1 zugeordnet. Der Wert $d \in \{0, 1, ..., k\}, k = max(D_{occ})$ ist dabei beliebig aber fest. Genauer müssen aufgrund des*

Constraints c^{order} *den Variablen* $x_i^{occc}, \forall i \leq d$ *der Wert 1 und den Variablen* $x_i^{occ}, \forall i > d$ *der Wert 0 zugeordnet sein. Aufgrund der Constraint* $C^{binding}$ *ergibt sich, dass der Variablen* $x^B_{occ,d}$ *der Wert 1 zugeordnet ist und wegen* $x_{occ} = d \leftrightarrow x^B_{occ,j} = 1, d_j = d \in D(occ)$ *folgt, dass der Variablen* x_{occ} *der Wert d zugeordnet werden muss. Da d viele Variablen in* $x^B_{i,j_i}, \forall i \in \{1, ..., n\}$ *den Wert 1 zugeordnet ist, folgt nach Fall 1, dass d viele ursprüngliche Variablen in* $x_1, ..., x_n$ *den Wert v mit* $v = d_{j_i} \in D_i,\ \forall i \in \{1, ..., n\}$*, annehmen.* □

5.2.2 Das *Global Cardinality*-Constraint

Das *Global Cardinality*-Constraint $c = gcc(\{x_1, ..., x_n\}, \{c_1, ..., c_m\})$ kann, wie in Kapitel 2.5 beschrieben, als Kombination mehrerer *Count*-Constraints angesehen werden. Demzufolge genügt es, eine Substituierung der zum *Global Cardinality*-Constraint äquivalenten *Count*-Constraints durchzuführen, um das Constraint c zu transformieren. Die Regularisierung bzw. die boolesche Skalarisierung kann dabei, wie zuvor beschrieben, auf den einzelnen *Count*-Constraints durchgeführt werden.

Bei der Regularisierung entstehen dabei mehrere *Regular*-Constraints, welche sich durch Überschneiden der zugehörigen levelbasierten DFAs zu einem levelbasierten DFA vereinigen lassen. Dieses Vorgehen führt in der Praxis allerdings häufig zu signifikant größeren levelbasierten DFAs, weswegen auf eine solche Verschmelzung in der weiteren Arbeit verzichtet wird.

5.2.3 Das *AllDifferent*-Constraint

Das *AllDifferent*-Constraint $c = AllDifferent(\{x_1, ..., x_n\})$ garantiert für eine geordnete Menge von Variablen $\{x_1, ..., x_n\}$, dass diese nur unterschiedliche Werte annehmen können. Das *AllDifferent*-Constraint kann neben der Beschreibung durch eine Menge paarweiser Ungleichheit-Constraints $C^{unequal} = \{c_{i,j} = (x_i \neq x_j) \mid \forall i, j \in \{1, ..., n\}, i < j\}$ auch durch eine Menge von *Count*-Constraints $C^{count} = \{c_j = Count(\{x_1, ..., x_n\}, occ, v_j) \mid \forall v_j \in \bigcup_{i=1}^{n} D_i, D_{occ} = \{0, 1\}\}$ dargestellt werden.

Demzufolge genügt es, eine Substituierung der zum *AllDifferent*-Constraint äquivalenten *Count*-Constraints mittels Regularisierung oder boolescher Skalarisierung durchzuführen.

Regularisierung

Über den Zwischenschritt der *Count*-Constraints entstehen bei der Regularisierung $k = |\bigcup_{i=1}^{n} D_i|$ viele *Regular*-Constraints $c_1^r = (X, M_1), ..., c_k^r = (X, M_k)$ über der gleichen, geordneten Variablenmenge $X = \{x_1, ..., x_n\}$. Sie drücken die zum *AllDifferent*-Constraint äquivalenten *Count*-Constraints aus und müssen alle in Konjunktion gelten. Mittels levelbasierter DFA-Schnittmengenbildung können die DFAs $M_1, ..., M_k$ zu einem DFA

M' kombiniert werden, so dass lediglich ein *Regular*-Constraint $c^r = (X, M')$ als Substituierung für das ursprüngliche *AllDifferent*-Constraint erzeugt wird. In der Praxis wird dieser DFA allerdings in der Regel so groß, dass es zu einer Verlangsamung, sowohl bei der Erzeugung der Constraints als auch bei deren Propagation, gegenüber den einzelnen *Regular*-Constraints $c_1^r, ..., c_k^r$ kommt. Aus diesem Grund wird im Zuge dieser Arbeit bei der Regularisierung von *AllDifferent*-Constraints von der ersten Variante mit k levelbasierten DFAs ohne anschließender Verschmelzung ausgegangen.

Boolesche Skalarisierung

Für die boolesche Skalarisierung von *AllDifferent*-Constraints kann die Substituierung ebenfalls über eine Transformation in *Count*-Constraints, mit anschließender boolescher Skalarisierung dieser *Count*-Constraints, durchgeführt werden.

Alternativ besteht für den Fall, dass die Anzahl der Variablen gleich der Anzahl aller möglichen Werte dieser Variablen ist ($n = |\bigcup_{i=1}^n D_i|$), die Möglichkeit, das *AllDifferent*-Constraint durch ein einzelnes boolesches *Scalar*-Constraint zu repräsentieren. Ist die Anzahl der Variablen n kleiner der Anzahl aller möglichen Werte dieser Variablen ($n < |\bigcup_{i=1}^n D_i|$), so kann das *AllDifferent*-Constraint durch zwei boolesche *Scalar*-Constraints repräsentiert werden.

Fall 1: $n = |\bigcup_{i=1}^n D_i|$. Aufgrund dessen, dass in dieser Arbeit von FD-CSPs ausgegangen wird, kann ohne Beschränkung der Allgemeinheit davon ausgegangen werden, dass die Domänenwerte aller Domänen vergleichbar sind bzw. linear geordnet werden können. Sei $X^<$ die Folge von Variablen in $x_{i,j}^B \in X^B$, die sich nach folgender Ordnung $\prec$ ergibt:

$$x_{i,j} \prec x_{i',j'} \leftrightarrow d_j < d_{j'} \vee (d_j = d_{j'} \wedge i < i'), d_j \in D_i, d_{j'} \in D_{i'}.$$

Sei $\overrightarrow{c} = (c_1, ..., c_{|X^<|})^T$ ein Vektor, wobei die ersten k_1 vielen Elemente gleich dem Wert 1, die nächsten k_2 Elemente gleich dem Wert 2 und weiterführend die jeweils nächsten k_i Elemente gleich dem Wert 2^{i-1} sind. Die Werte k_i entsprechen dabei gleich der Anzahl an Variablen, deren Domänen den i-größten Wert aus $\bigcup_{i=1}^n D_i$ enthalten. Zur Veranschaulichung ist das Beispiel 5.3 gegeben.

Beispiel 5.3: Eine boolesche Skalarisierung des *AllDifferent*-Constraints. *Gegeben sei das Constraint* AllDifferent$(\{x_1, x_2, x_3, x_4\})$ *mit* $D_1 = \{1, 2, 4\}, D_2 = \{1, 2, 3, 4\}, D_3 = \{2, 3, 4\}, D_4 = \{1, 2, 3, 4\}$. *Die booleschen Variablen in* $x_{i,j}^B \in X^B, \forall i \in \{1, 2, 3, 4\}, j \in \{1, ..., |D_i|\}$ *repräsentieren jeweils die Zuordnung des Wertes* $d_j \in D_i$ *zu Variable* x_i. *Die Variable* $x_{1,3}^B$ *repräsentiert somit die Zuweisung des Wertes* $d_3 = 4$ *an Variable* x_1. *Die Variablen in* $X^<$ *seien nach* $\prec$ *wie folgt geordnet* $x_{1,1}^B, x_{2,1}^B, x_{4,1}^B, x_{1,2}^B, x_{2,2}^B, x_{3,1}^B, x_{4,2}^B, x_{2,3}^B, x_{3,2}^B, x_{4,3}^B, x_{1,3}^B, x_{2,4}^B, x_{3,3}^B, x_{4,4}^B$. *Der zugehörige Vektor* $\overrightarrow{c}$ *hat dementsprechend den Wert* $(1, 1, 1, 2, 2, 2, 2, 4, 4, 4, 8, 8, 8, 8)^T$.

Das *Scalar*-Constraint $c^{\text{allDifferent}}$ in Gleichung 5.10 über den Variablen $X^<$ ist eine boolesche Skalarisierung des gegebenen *AllDifferent*-Constraints c.

$$c^{\text{allDifferent}} = Scalar(X^<, \overrightarrow{c}, =, 2^n - 1) \tag{5.10}$$

Für das Beispiel 5.3 ergibt sich dementsprechend das *Scalar*-Constraint:

$$\begin{aligned} c^{\text{allDifferent}} = Scalar((&x^B_{1,1}, x^B_{2,1}, x^B_{4,1}, x^B_{1,2}, x^B_{2,2}, x^B_{3,1}, x^B_{4,2}, x^B_{2,3}, x^B_{3,2}\; x^B_{4,3}, x^B_{1,3}, x^B_{2,4}, x^B_{3,3}, x^B_{4,4}),\\ &(1,1,1,2,2,2,2,4,4,4,8,8,8,8)^T,\ =,\ 15). \end{aligned}$$

Damit das Skalarprodukt aus $X^<$ und $\overrightarrow{c}$ gleich den Wert 15 ergibt, muss jeweils genau einer Variablen aus $\{x^B_{1,1}, x^B_{2,1}, x^B_{4,1}\}$, $\{x^B_{1,2}, x^B_{2,2}, x^B_{3,1}, x^B_{4,2}\}$, $\{x^B_{2,3}, x^B_{3,2}\; x^B_{4,3}, x^B_{1,3}\}$ und $\{x^B_{2,4}, x^B_{3,3}, x^B_{4,4}\}$ der Wert 1 zugeordnet werden ($1 + 2 + 4 + 8 = 15$). Ein Zusammensetzen der Summe aus mehr als 4 Werten (z.B. $1+2+2+2+8$) ist aufgrund von Voraussetzung 5.2 nicht möglich, da diese für genau n Variablen in X^B bzw. $X^<$ fordert, dass diese mit dem Wert 1 belegt werden. Eine gültige Lösung ist beispielsweise ϕ' mit $\phi'(x^B_{1,1}) = 1, \phi'(x^B_{3,1}) = 1, \phi'(x^B_{2,3}) = 1$ und $\phi'(x^B_{4,4}) = 1$, wobei allen anderen Variablen $x^B \in X^B$ der Wert 0 zugeordnet ist. Diese Lösung entspricht der Lösung ϕ mit $\phi(x_1) = 1$, $\phi(x_2) = 3$, $\phi(x_3) = 2$ und $\phi(x_4) = 4$ des ursprünglichen CSPs.

Beweis 5.3. ***Teil 1*** *($c \rightarrow sub(c)$)***:** *Wenn das* AllDifferent-*Constraint erfüllt ist, ist jeder ursprünglichen Variable $x_1, ..., x_n$ ein Wert v_i mit $v_i \in D_i, i \in \{1, ..., n\}, v_1 \neq v_2 \neq ... \neq v_n$ zugeordnet. Aufgrund der Ordnung $\prec$ repräsentieren genau die ersten k_1 booleschen Variablen in $X^<$ die Zuordnung der ursprünglichen Variablen $x_i \in X$ zu dem Wert v'_1 mit $v'_1 = min(v_1, ..., v_n)$, bei denen v'_1 in der Domäne D_i liegt. Genauso liegen in den jeweils k_i folgenden Variablen in $X^<$ genau die booleschen Variablen, welche die Zuordnung der ursprünglichen Variablen $x_i \in X$ zu dem Wert v'_i repräsentieren. Der Wert v'_i ist dabei jeweils der kleinste Wert in $v_1, ..., v_n$ ohne die Werte $v'_1, ..., v'_{i-1}$.*

In jeder dieser n Folgen von booleschen Variablen der Länge $k_i, i \in \{1, ..., n\}$ in $X^<$ hat jeweils genau eine boolesche Variable den Wert eins zugeordnet, da aufgrund des erfüllten AllDifferent-*Constraints genau eine ursprüngliche Variable den Wert v_i zugeordnet bekommen hat. Im* Scalar-*Constraint $c^{allDifferent}$ haben alle Variablen der i-ten Variablenfolge das Skalar 2^{i-1} als Gewicht zugeordnet. Daraus ergibt sich, dass jeder Wert $2^{i-1}, \forall i \in \{1, ..., n\}$ genau einmal in der Summe berücksichtigt werden muss. Das* Scalar-*Constraint $c^{allDifferent}$ spiegelt dadurch die Gleichung $\Sigma^n_{i=1} 2^{i-1} = 2^i - 1$ wider, wobei es sich um eine wahre Aussage handelt.* □

Teil 2 *($sub(c) \rightarrow c$)***:** *Wenn das* Scalar-*Constraint $c^{allDifferent}$ erfüllt ist, muss die Summe der Produkte aus Variablen in $X^<$ und $\overrightarrow{c}$ gleich $2^n - 1$ sein. Das kann nur der Fall sein, wenn jedes unterschiedliche Gewicht $2^0, ..., 2^{n-1}$ genau einmal als Summand auftritt.*

Widerspruchsbeweis: Angenommen alle k_i Variablen, die dem Gewicht 2^{i-1} zugeordnet sind, bekommen den Wert 0 zugewiesen, dann wäre der Wert 2^{i-1} nicht in der Summe berücksichtigt und müsste durch das möglicherweise mehrfache Vorkommen anderer Zweierpotenzen 2^j mit $j \in \{1, ..., n-1\}, j \neq i-1$ ersetzt werden, um die Summe $2^n - 1$ zu erreichen. Der Wert 2^{i-1} kann allerdings nicht durch das Vorkommen einer oder mehrerer größerer Zweierpotenzen ausgeglichen werden, da diese dann immer zu einer Übersteigung der Summe führen würden. Es verbleibt die

Möglichkeit, den Wert 2^{i-1} *durch das Vorkommen mehrerer kleinerer Zweierpotenzen zu ersetzen. Dafür würde z.B. zweimal der Wert* 2^{i-2} *oder viermal der Wert* 2^{i-3} *benötigt. Hierzu müsste aber mehr als n booleschen Variablen in* $X^{<}$ *der Wert eins zugeordnet werden, was aufgrund der Bedingung* $\sum_{j=1}^{|D_i|} x_{i,j}^{B} = 1, \forall i \in \{1, ..., n\}$ *nicht möglich ist. Diese garantiert, dass genau einer booleschen Variable* $x_{i,j}^{B}$ *für* $j \in \{1, ..., |D_i|\}$*, die die Variablenzuordnungen der ursprünglichen Variable* x_i *repräsentieren, der Wert eins zugeordnet ist. Da die booleschen Variablen in* $X^{<}$ *genau alle Variablenzuordnungen für* $x_1, ..., x_n$ *repräsentieren, müssen genau n davon erfüllt sein.* ↯

Somit muss, damit $c^{allDifferent}$ *erfüllt ist, in jeder der n Folgen einer Variable der Wert* 1 *zugeordnet werden. Die erste Folge beinhaltet dabei die ersten* k_1 *Werte aus* $X^{<}$*, die zweite die nächsten* k_2 *Werte usw. Jede dieser Folgen mit Index i repräsentiert dabei einen anderen Wert* v_i*, den eine der ursprünglichen Variablen annimmt. Somit erhält jede ursprüngliche Variable einen anderen Wert* v_i *und das* AllDifferent-*Constraint c ist erfüllt.* □

Fall 2: $n \neq |\bigcup_{i=1}^{n} D_i|$. Seien $X^{<}$ und $\overrightarrow{c} = (c_1, ..., c_{|X^{<}|})$ wie im Fall 1 gegeben. Sei m gleich die Anzahl an unterschiedlichen Werten, die die Variablen $x_1, ..., x_n$ annehmen können ($m = |\bigcup_{i=1}^{n} D_i|, m > n$). $X^{<'}$ ist gleich die Variablenfolge $X^{<}$ zuzüglich m neuer boolescher Variablen $y_1, ..., y_m$, die an das Ende der Folge $X^{<}$ angefügt werden. Der Vektor $\overrightarrow{c'} \in \mathbb{N}^{|\overrightarrow{c}|+m}$ sei definiert durch $\overrightarrow{c'_i} = \overrightarrow{c_i}, \forall i \in \{1, ..., |\overrightarrow{c}|\}$ und $\overrightarrow{c'_i} = -2^{j-1}, \forall i \in \{|\overrightarrow{c}|+1, ..., |\overrightarrow{c}|+m\}, j = i-|\overrightarrow{c}|$. Die beiden *Scalar*-Constraints 5.11 und 5.12 sind eine boolesche Skalarisierung des *AllDifferent*-Constraints *c*.

$$c^{\text{allDifferent}} = Scalar(X^{<'}, \overrightarrow{c'}, =, 0) \tag{5.11}$$

$$c^{\text{count}} = Scalar(\{y_1, ..., y_m\}, (1, 1, 1, ..., 1)^T, =, n) \tag{5.12}$$

Beispiel 5.4: Eine boolesche Skalarisierung des *AllDifferent*-Constraints für $n \neq |\bigcup_{i=1}^{n} D_i|$**.** *Gegeben sei das Constraint* $c = \text{AllDifferent}(x_1, x_2)$ *mit* $D_1 = \{1, 2\}$ *und* $D_2 = \{1, 3\}$*. Die boolesche Skalarisierung*

$$c^{allDifferent} = \text{Scalar}(\{x_{1,1}^{B}, x_{2,1}^{B}, x_{1,2}^{B}, x_{2,3}^{B}, y_1, y_2, y_3\}, (1, 1, 2, 4, -1, -2, -4)^T, =, 0)$$

und

$$c^{count} = \text{Scalar}(\{y_1, y_2, y_3\}, (1, 1, 1)^T, =, 3)$$

kann als Substituierung von c verwendet werden.

Beweis 5.4. *(Beweisskizze) Das Constraint* c^{count} *entspricht einem Constraint* Count($\{y_1, ..., y_m\}$, *n*, 1) *(siehe Abschnitt 5.2.1). Die Summe, die das* Scalar-*Constraint* $c^{allDifferent}$ *repräsentiert, kann mit einer Hilfsvariablen p in zwei Summen aufgeteilt werden:* $X_1^{<} * 1 + ... + X_{k_1}^{<} * 1 + X_{k_1+1}^{<} * 2 + ... + X_{k_1+k_2}^{<} * 2 + ... + X_{k_1+...+k_m}^{<} * 2^{m-1} = p$ *und* $y_1 * 1 + y_2 * 2 + ... y_m * 2^{m-1} = p$*. Wird die zweite Summe in Zusammenhang mit dem Constraint* c^{count} *aus Gleichung 5.12 betrachtet, so kann sich p nur aus der Summe von n verschiedenen Zweierpotenzen ergeben. Mit diesem*

Wissen können die Beweise für beide Richtungen ($c \rightarrow sub(c)$ und $sub(c) \rightarrow c$) analog zu Fall 1 durchgeführt werden.

5.2.4 Das *AllEqual*-Constraint

Das *AllEqual*-Constraint $c = \mathit{AllEqual}(\{x_1, ..., x_n\})$ garantiert für eine geordnete Menge von Variablen $\{x_1, ..., x_n\}$, dass diese alle den gleichen Wert annehmen.

Regularisierung

Bei der Regularisierung kann der Sachverhalt genutzt werden, dass sich das *AllEqual*-Constraint auch durch die Verknüpfung mehrerer *Count*-Constraints mittels dem Oder-Operator ($\vee$) darstellen lässt:

$$c = \mathit{AllEqual}(\{x_1, ..., x_n\}) := \bigvee_{\forall v \in D'} \mathit{Count}(\{x_1, ..., x_n\}, n, v), \text{ mit } D' = D_1 \cap ... \cap D_n \tag{5.13}$$

Die Darstellung durch *Count*-Constraints ermöglicht eine effektive Umwandlung in ein *Regular*-Constraint. Dafür werden zunächst die einzelnen *Count*-Constraints in *Regular*-Constraints überführt. Seien M_1 bis $M_{|D'|}$ die levelbasierten DFAs zu den so erzeugten *Regular*-Constraints. Mittels levelbasierter DFA-Vereinigung $\bigcup_{i=1}^{|D'|} M_i$ lässt sich ein DFA $M^{AllEqual}$ erzeugen, der genau die Werte als Eingaben akzeptiert, die eine gültige Belegung für das *AllEqual*-Constraint c darstellen. Das Constraint $\mathit{Regular}(\{x_1, ..., x_n\}, M^{AllEqual})$ ist demzufolge eine Regularisierung des ursprünglichen *AllEqual*-Constraints c.

Boolesche Skalarisierung

Die Konjunktion der *Scalar*-Constraints in Gleichung 5.14 dient als Substituierung für das *AllEqual*-Constraint c.

$$\mathit{Scalar}(\{x^B_{i,j_i}, x^B_{i+1,j_{i+1}}\}, (1, -1)^T, =, 0), \forall i \in \{1, ..., |X| - 1\}, d_{j_i} \in D_i \cap D_{i+1} \text{ für die gilt } d_{j_i} = d_{j_{i+1}} \tag{5.14}$$

Sie drücken jeweils die paarweise Gleichheit zweier boolescher Variablen x^B_{i,j_i} und $x^B_{i+1,j_{i+1}}$ aus, die der ursprünglichen Belegung $x_i = d$ und $x_{i+1} = d$, mit d ist das j_i-te Element in D_i und das j_{i+1}-te Element in D_{i+1}, entsprechen.

Beweis 5.5. ***Teil 1*** ($c \rightarrow sub(c)$)*: Ohne Beschränkung der Allgemeinheit ist allen Variablen $x_1, ..., x_n$ bei erfülltem* AllEqual*-Constraint c der Wert v zugeordnet. Aufgrund von Voraussetzung 5.1 folgt, dass den Variablen $x^B_{1,j_1}, ..., x^B_{n,j_n}$ mit $d_{j_1} = ... = d_{j_n} = v, d_{j_i} \in D_i, \forall i \in \{1, ..., n\}$ der Wert 1 zugeordnet wird. Für alle anderen Werte $d_k \in D_i$ mit $d_k \neq v$ für alle Domänen $D_i, i \in \{1, ..., n\}$ ergibt sich ebenfalls aus Voraussetzung 5.1, dass den Variablen $x^B_{i,k}$ der Wert 0 zugeordnet wird. Für die in 5.14 angegebenen* Scalar*-Constraints folgt dadurch entweder die*

Summe $1 * 1 - 1 * 1 = 0$ *oder* $1 * 0 - 1 * 0 = 0$. *In beiden Fällen sind die* Scalar-*Constraints erfüllt.* □

Teil 2 ($sub(c) \rightarrow c$)**:** *Aufgrund von Voraussetzung 5.2 ist genau einer Variablen* $x^B_{i,1}, ..., x^B_{i,|D_i|}$ *der Wert 1 zugeordnet. Ohne Beschränkung der Allgemeinheit sei* $x^B_{i,j}$ *mit* $d_j = v \in D_i$ *die Variable, der der Wert 1 zugeordnet ist.*

Wenn die Scalar-*Constraints in Gleichung 5.14 erfüllt sind, dann wurde allen Variablen* x^B_{i,j_i} *mit* $d_{j_i} = v, \forall i \in \{1, ..., n\}$ *der Wert 1 zugeordnet. Aufgrund von Voraussetzung 5.2 müssen damit allen Variablen* x^B_{i,j_i} *mit* $d_{j_i} \neq v, \forall i \in \{1, ..., n\}$ *der Wert 0 zugewiesen worden sein. Nach Voraussetzung 5.1 haben folglich alle Variablen* $x_1, ..., x_n \in X$ *den Wert* v *erhalten.* □

5.2.5 Das *Scalar*-Constraint

Das *Scalar*-Constraint $c = Scalar(\{x_1, ..., x_n\}, \overrightarrow{c}, \Re, x_r)$, $x_r, x_i \in X, \overrightarrow{c} \in \mathbb{Z}^n, \forall i \in \{1, ..., n\}$, $\Re \in \{<, \leq, =\geq, >, \neq\}$ garantiert, dass das Skalarprodukt der Variablen $x_1, ..., x_n$ mit dem Vektor $\overrightarrow{c}$ in Relation $\Re$ zur Variable x_r steht.

Regularisierung

In [151] ist ein Ansatz vorgestellt, der ein allgemeines Vorgehen zur Überführung eines *Scalar*-Contraints in einen levelbasierten DFA M beschreibt. Dabei beinhaltet die Eingabe allerdings keine Variable x_r, sondern eine konstante untere und obere Grenze l und u. Jedes *Scalar*-Constraint der Form $c = Scalar(\{x_1, ..., x_n\}, (c_1, ..., c_n)^T, \Re, x_r)$ kann in die Form $c' = Scalar(\{x_1, ..., x_n, x_r\}, (c_1, ..., c_n, -1)^T, \Re, 0)$ überführt werden, wodurch das angegebene Verfahren mit $l = u = 0$ durchgeführt werden kann. Im Folgenden wird von einem *Scalar*-Constraint der Form c' ausgegangen.

Der resultierende levelbasierte DFA M in [151] ist nicht in jedem Fall minimal. In der hier vorliegenden Arbeit wurde die Anzahl der Knoten und Kanten von M dahingehend verringert, dass alle akzeptierenden Knoten der letzten Ebene von M zu einem akzeptierenden Knoten verschmolzen wurden und anschließend eine Minimierung von M durchgeführt wurde. Sei M^{min} der resultierende minimale levelbasierte DFA. Die Minimierung hat zur Folge, dass in der vielfach ausgeführten Propagation mit einer kleineren Datenstruktur gearbeitet werden kann, was bei der Propagation in der Regel sowohl Arbeitsspeicher als auch Rechenzeit einspart. Die Rechenzeitersparnis kann zum Beispiel dann eintreten, wenn durch andere Constraints oder die Suche Werte aus einer Domäne ausgeschlossen werden. Bei einem levelbasierten DFA hat dies bestenfalls zu Folge, dass kaskadierend Kanten ausgeschlossen werden. Ist der levelbasierte DFA kleiner, so müssen weniger Kanten überprüft und gegebenenfalls ausgeschlossen werden, um ähnlich viele Werteausschlüsse aus Domänen zu schlussfolgern.

Für die Regularisierung können die Variablen $X' = \{x_1, ..., x_n\}$ bzw. $X' = \{x_1, ..., x_n, x_r\}$ und der levelbasierte DFA M^{min} als Eingabe berücksichtigt werden. Das Constraint

$Regular(X', M^{min})$ kann somit als Substituierung für das *Scalar*-Constraint c verwendet werden.

Boolesche Skalarisierung

Das *Scalar*-Constraint in Gleichung 5.15 dient als Substituierung für das *Scalar*-Constraint $c' = Scalar(\{x_1, ..., x_n, x_r\}, (c_1, ..., c_n, -1)^T, \mathfrak{R}, s)$.

$$\begin{aligned} &Scalar(\{x^B_{1,1}, ..., x^B_{1,|D_1|},\ x^B_{2,1}, ..., x^B_{2,|D_2|},\ ..., x^B_{n,1}, ..., x^B_{n,|D_n|},\ x^B_{r,1}, ..., x^B_{r,|D_r|}\}, \\ &\quad (c_1 * d_{1,1}, ..., c_1 * d_{1,|D_1|},\ c_2 * d_{2,1}, ..., c_2 * d_{2,|D_2|},\ ..., c_n * d_{n,1}, ..., c_n * d_{n,|D_n|}, \\ &\quad - d_{r,1}, ..., -d_{r,|D_r|})^T, \mathfrak{R}, s) \end{aligned} \tag{5.15}$$

Beispiel 5.5: Eine boolesche Skalarisierung eines *Scalar*-Constraints. *Gegeben sei das Constraint* $c = \text{Scalar}(\{x_1, x_2\}, (1, 2), =, 3)$ *mit* $D_1 = D_2 = \{1, 2, 3\}$. *Die boolesche Skalarisierung* $\text{Scalar}(\{x^B_{1,1}, x^B_{1,2}, x^B_{1,3}, x^B_{2,1}, x^B_{2,2}, x^B_{2,3}\}, (1*1, 1*2, 1*3, 2*1, 2*2, 2*3)^T, =, 3)$ *dient als Substituierung von* c. *Aufgrund von Voraussetzung 5.2 muss genau eine Variable in* $\{x^B_{1,1}, x^B_{1,2}, x^B_{1,3}\}$ *und eine in* $\{x^B_{1,1}, x^B_{1,2}, x^B_{1,3}\}$ *den Wert 1 zugewiesen bekommen. Es existiert somit nur eine Lösung* ϕ' *mit* $\phi'(x_{1,1}) = 1$, $\phi'(x_{2,1}) = 1$, *bei der allen anderen Variablen* $x^B \in X^B$ *der Wert 0 zugeordnet ist. Diese Lösung entspricht der Belegung* ϕ *mit* $\phi(x_1) = 1$ *und* $\phi(x_2) = 2$, *des ursprünglichen* Scalar-*Constraints* c.

Beweis 5.6. ***Teil 1*** ($c \rightarrow sub(c)$)**:** *Wenn das* Scalar-*Constraint* c' *erfüllt ist, wurde jeder Variablen* $x_1, ..., x_n, x_r$ *ein Wert* $d_1, ..., d_n, d_r$ *mit* $d_i \in D_i, \forall i \in \{1, ..., n\}$ *und* $d_r \in D_r$ *zugeordnet, so dass die Summe* $d_1 * c_1 + ... + d_n * c_n - 1 * d_r$ *gleich 0 ist. Aufgrund von Voraussetzung 5.1 ergibt sich, dass den Variablen* $x^B_{1,j_1}, ..., x^B_{n,j_n}, x^B_{r,j_r}$ *mit* $d_{j_i} = d_i \in D_i$ *und* $d_{r,j_r} = d_r \in D_r$ *der Wert 1 und allen anderen Variablen* $x^B_{i,k_i}, \forall i \in \{1, ..., n\}, k_i \in \{1, ..., |D_i|\}$ *mit* $d_{k_i} \neq d_i$ *und* $x^B_{r,k_r}, \forall k_i \in \{1, ..., |D_r|\}$ *mit* $d_{k_r} \neq d_r$ *der Wert 0 zugeordnet ist. Demzufolge repräsentiert das* Scalar-*Constraint in Gleichung 5.15 die Summe:*

$$\begin{aligned} &0 * c_1 * d_{1,1} + ... + 0 * c_1 * d_{1,j_1-1} + 1 * c_1 * d_{1,j_1} + 0 * c_1 * d_{1,j_1+1} + ... + 0 * c_1 * d_{1,j_{|D_1|}} + \\ &0 * c_2 * d_{2,1} + ... + 0 * c_2 * d_{2,j_2-1} + 1 * c_2 * d_{2,j_2} + 0 * c_2 * d_{1,j_2+1} + ... + 0 * c_2 * d_{1,j_{|D_2|}} + \\ &\hfill ... = 0 \\ &0 * c_n * d_{n,1} + ... + 0 * c_n * d_{n,j_n-1} + 1 * c_n * d_{1,j_n} + 0 * c_n * d_{1,j_n+1} + ... + 0 * c_n * d_{1,j_{|D_n|}} + \\ &0 * -1 * d_{r,1} + ... + 0 * -1 * d_{r,j_r-1} + 1 * -1 * d_{r,j_r} + 0 * -1 * d_{r,j_r+1} + ... + 0 * -1 * d_{1,j_{|D_r|}} \end{aligned} \tag{5.16}$$

Wird die Summe in 5.16 um die Nullwerte gekürzt, so ergibt sich die Summe: $1 * c_1 * d_{1,j_1} + 1 * c_2 * d_{2,j_2} ... + 1 * c_n * d_{1,j_n} + 1 * -1 * d_{r,j_r} = 0$. *Was unter Berücksichtigung von* $d_{j_i} = d_i \in D_i, \forall i \in \{1, ..., n\}$ *und* $d_{r,j_r} = d_r \in D_r$ *gleich die Summe ist, die durch das ursprüngliche* Scalar-*Constraint* c' *repräsentiert wird.* □

Teil 2 ($sub(c) \rightarrow c$)**:** *Aufgrund von Voraussetzung 5.2 wird genau einer Variablen jeder Folge* $x^B_{i,1}, ..., x^B_{i,|D_i|}$ *der Wert 1 und allen anderen der Wert 0 zugeordnet. Daraus ergibt sich, dass das* Scalar-*Constraint in Gleichung 5.15 die gleiche Summe repräsentiert, die auch das ursprüngliche* Scalar-*Constraint* c' *darstellt.* □

5.2.6 Das *Sum*-Constraint

Das *Sum*-Constraint $c = Sum(\{x_1, ..., x_n\}, \Re, x_r)$ mit $x_i, x_r \in X, \forall i \in \{1, ..., n\}, \Re \in \{<, \leq, =\geq, >, \neq\}$ garantiert, dass die Summe der den Variablen $x_1, ..., x_n$ zugeordneten Werte in Relation $\Re$ zum Wert der Variablen x_r steht.

Das *Sum*-Constraint ist eine Spezialisierung des *Scalar*-Constraints, bei dem die Werte des Skalarvektors $\overrightarrow{c}$ alle den Wert 1 haben $\overrightarrow{c} = (1, ..., 1)^T$. Dementsprechend wurden bereits im vorherigen Abschnitt Möglichkeiten der Regularisierung und der booleschen Skalarisierung diskutiert.

5.2.7 Das *Table*-Constraint

Ein *Table*-Constraint $c = Table(\{x_1, ..., x_n\}, T)$ garantiert, dass die Variablen $x_1, ..., x_n$ nur Werte $d_1, ..., d_n$ annehmen können, für die das Tupel $(d_1, ..., d_n)$ ein Element der Tupelliste T ist.

Regularisierung

Für die Regularisierung kann der in Abschnitt 5.1.1 vorgestellte Ansatz verwendet werden. Die Tupel in T können an dieser Stelle direkt als Lösungen S betrachtet werden, so dass diese nicht zusätzlich berechnet werden müssen. Der aus Algorithmus 2 erzeugte levelbasierte DFA M kann im Constraint $Regular(\{x_1, ..., x_n\}, M)$ als Substituierung für c eingesetzt werden.

Boolesche Skalarisierung

Für die boolesche Skalarisierung eines *Table*-Constraints können die bereits vorgestellten Verfahren zur booleschen Skalarisierung von kleinen Constraint-Mengen genutzt werden (die konfliktbasierte boolesche Skalarisierung, siehe Abschnitt 5.1 und die Support-basierte boolesche Skalarisierung, siehe Abschnitt 5.2).

5.2.8 Das *Regular*-Constraint

Das *Regular*-Constraint $c = regular(\{x_1, ..., x_n\}, M)$ garantiert, dass den Variablen $x_1, ..., x_n$ nur Werte $d_1, ..., d_n$ zugeordnet werden können, deren Konkatenation $d_1 \circ ... \circ d_n$ der DFA M als Eingabe akzeptiert.

Regularisierung

Für die Regularisierung kann das Constraint c beibehalten werden, eine Umformung ist nicht notwendig.

Boolesche Skalarisierung

Die hier beschriebene Umwandlung eines *Regular*-Constraints in ein boolesches *Scalar*-Constraint entspricht einer Weiterentwicklung der im Zusammenhang mit dieser Arbeit entwickelten und in [99] publizierten Transformation.

Im Folgendem beschreibt die Notation $q_{i,j} \xrightarrow{d_i} q_{i+1,k}$ einen Übergang von Zustand $q_{i,j} \in Q_i$ zu Zustand $q_{i+1,k} \in Q_{i+1}$ mit Wert $d_i \in D_i$ eines levelbasierten DFAs $M = (Q, \Sigma, \delta, q_0, F)$, wobei die Zustandsmenge Q in $(n+1)$ viele Level $Q_0, ..., Q_n$ aufgeteilt ist. Jeder Pfad $q_{0,0} \xrightarrow{d_1} q_{1,j_1} \xrightarrow{d_2} ... \xrightarrow{d_n} q_{n,0}$ in M vom Startzustand $q_{0,0}$ zum finalen Zustand $q_{n,0}$ repräsentiert genau eine gültige Belegung ϕ mit $\phi(x_1) = d_1, ..., \phi(x_n) = d_n$ des *Regular*-Constraints c. Wenn alle Variablen instantiiert sind und das *Regular*-Constraint c erfüllt ist, dann verbleibt genau ein solcher Pfad in M.

Genauso wie die Zustände Q in einem levelbasierten DFA in disjunkte Zustandsmengen $Q_0, ..., Q_n$ partitioniert werden können, können auch die Übergänge δ in Level $\delta_1, ..., \delta_n$ aufgeteilt werden. Ein Übergang $e_{i,j} = (q_{i-1,j_1}, d_{i,j_2}, q_{i,j_3}) \in \delta$ gehört genau dann zur Menge δ_i, wenn dieser ein Übergang eines Zustandes $q_{i-1,j_1} \in Q_{i-1}$ mit Wert $d_{i,j_2} \in D_i$ zu einem Zustand $q_{i,j_3} \in Q_i$ ist. Bei der folgenden Transformation wird für jeden Übergang $e_{i,j} \in \delta_i, \forall i \in \{1, ..., n\}$ des levelbasierten DFAs $M = (Q, \Sigma, \delta, q_0, F)$ eine weitere boolesche Variable $x^{\delta}_{i,j} \in X^{\delta}$ erzeugt, die repräsentiert, ob die Kante e Teil des Lösungspfades von $q_{0,0}$ nach $q_{n,0}$ ist ($x^{\delta}_{i,j} = 1$) oder nicht ($x^{\delta}_{i,j} = 0$).

Die Menge $X^{\delta}_{i,v}$ umfasst alle Variablen $x^{\delta}_{i,j} \in X^{\delta}$, für die gilt, dass der zugehörige Übergang $e_{i,j} = (q_{i-1,j_1}, d_{i,j_2}, q_{i,j_3})$ in δ_i mit Wert $d_{i,j_2} = v$ erfolgt. Die Mengen $X^{in}_{i,j}$ und $X^{out}_{i,j}$ für $\forall i \in \{0, ..., n\}, j \in \{1, ..., |Q_i|\}$ beinhalten genau die Variablen $x^{\delta}_{i,j} \in X^{\delta}$, die Übergänge $e \in \delta_i$ repräsentieren und über den Zustand $q_{i,j}$ als Zielzustand bzw. Ausgangszustand verfügen. Die Constraints in Gleichung 5.17 und Gleichung 5.18 dienen als boolesche Substituierung des gegebenen *Regular*-Constraints c.

$$C^{\text{Arc}} = \{c^{\text{Arc}}_{i,v} = \mathit{Scalar}(\{x^{B}_{i,j}, X^{\delta}_{i,v}\}, (-1, 1, ..., 1)^T, =, 0), v = d_j \in D_i, \forall i \in \{1, ..., n\}, j \in \{1, ..., |D_i|\}\} \tag{5.17}$$

$$\begin{aligned} C^{\text{Node}} = \{c^{\text{Node}_{i,j}} = \mathit{Scalar}(X^{in}_{i,j} \cup X^{out}_{i,j}, (-1, ..., -1, 1, ..., 1)^T, =, r), \forall i \in \{0, ..., n\}, j \in \{1, ..., |Q_i|\} \\ \text{mit } r = 1 \text{ für } i = 0, r = -1 \text{ für } i = n, \text{ und } r = 0 \text{ sonst}\} \end{aligned} \tag{5.18}$$

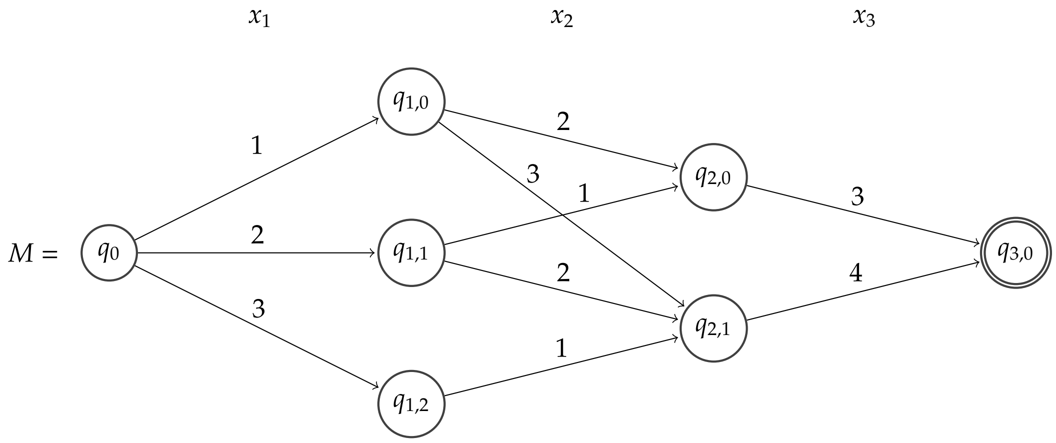


Abbildung 5.1: Der zu dem *Summen*-Constraint *Sum*$(\{x_1, x_2\}, =, x_3)$ äquivalente minimale levelbasierte DFA M.

Die Constraints C^{Arc} repräsentieren, dass nur eine Kante pro Level Teil des Lösungspfades sein kann. Die Constraints int C^{Node} bewirken, dass für jeden Knoten des Lösungspfades genau eine ausgehende (außer beim Endknoten $q_{n,0}$) und genau eine eingehende Kante (außer beim Startknoten $q_{n,0}$) Teil des Lösungspfades ist.

Beispiel 5.6: Die boolesche Skalarisierung eines levelbasierten DFAs. *Der in Abbildung 5.1 dargestellte levelbasierte DFA M entspricht dem Summen-Constraint* Sum$(\{x_1, x_2\}, =, x_3)$ *mit* $D_1 = D_2 = \{1, 2, 3\}$ *sowie* $D_3 = \{3, 4\}$ *und wird im Folgenden mittels boolescher Skalarisierung substituiert.*

Nach Voraussetzung 5.1 und 5.2 ergibt sich das folgende boolesche Ausgangs-CSP 12:

CSP 12: $P = (X^B, D^B, C^S)$ mit:

$$\begin{aligned} X^B &= \{x^B_{1,1}, x^B_{1,2}, x^B_{1,3},\ x^B_{2,1}, x^B_{2,2}, x^B_{2,3},\ x^B_{3,1}, x^B_{3,2}\} \\ D^B &= \{D_{1,1} = D_{1,2} = D_{1,3} = D_{2,1} = D_{2,2} = D_{2,3} = D_{3,1} = D_{3,2} = \{0, 1\}\} \\ C^S &= \{c^{\text{oneOff}}_1 = (x^B_{1,1} + x^B_{1,2} + x^B_{1,3} = 1), \\ &\quad\ c^{\text{oneOff}}_2 = (x^B_{2,1} + x^B_{2,2} + x^B_{2,3} = 1), \\ &\quad\ c^{\text{oneOff}}_3 = (x^B_{3,1} + x^B_{3,2} = 1)\} \end{aligned}$$

Um eine Äquivalenz zu dem ursprünglichen levelbasierten DFA (in Abbildung 5.1) zu erzeugen, müssen die folgenden Äquivalenzen nach Voraussetzung 5.1 erfüllt sein:

$(x_1 = 1) \leftrightarrow (x^B_{1,1} = 1),\ (x_1 = 2) \leftrightarrow (x^B_{1,2} = 1),\ (x_1 = 3) \leftrightarrow (x^B_{1,3} = 1),$
$(x_2 = 1) \leftrightarrow (x^B_{2,1} = 1),\ (x_2 = 2) \leftrightarrow (x^B_{2,2} = 1),\ (x_2 = 3) \leftrightarrow (x^B_{2,3} = 1),$
$(x_3 = 3) \leftrightarrow (x^B_{3,1} = 1),\ (x_3 = 4) \leftrightarrow (x^B_{3,2} = 1)$

Für jeden Übergang in δ *des DFAs M wird eine neue boolesche Variable* $x_{i,j}$ *angelegt, wobei i das Level des Übergang und j den Index des Übergangs in der Übergangsmenge* δ_j *widerspiegelt. Es*

Variable	$x^{\delta}_{1,1}$	$x^{\delta}_{1,2}$	$x^{\delta}_{1,3}$
δ	$q_{0,0} \xrightarrow{1} q_{1,0}$	$q_{0,0} \xrightarrow{2} q_{1,1}$	$q_{0,0} \xrightarrow{3} q_{1,2}$

Variable	$x^{\delta}_{2,1}$	$x^{\delta}_{2,2}$	$x^{\delta}_{2,3}$	$x^{\delta}_{2,4}$	$x^{\delta}_{2,5}$
δ	$q_{1,0} \xrightarrow{2} q_{2,0}$	$q_{1,0} \xrightarrow{3} q_{2,1}$	$q_{1,1} \xrightarrow{1} q_{2,0}$	$q_{1,1} \xrightarrow{2} q_{2,1}$	$q_{1,2} \xrightarrow{1} q_{2,1}$

Variable	$x^{\delta}_{3,1}$	$x^{\delta}_{3,2}$
δ	$q_{2,0} \xrightarrow{3} q_{3,0}$	$q_{2,1} \xrightarrow{4} q_{3,0}$

Tabelle 5.4: Die Zuordnung neuer boolescher Variablen zu den Übergängen des levelbasierten DFAs M.

ergeben sich die Variablen X^{δ} nach der in Tabelle 5.4 angegebenen Zuordnung der Variablen an die Übergänge des levelbasierten DFAs M:

Level 1:
$X^{\delta}_{1,1} = \{x^{\delta}_{1,1}\}$
$X^{\delta}_{1,2} = \{x^{\delta}_{1,2}\}$
$X^{\delta}_{1,3} = \{x^{\delta}_{1,3}\}$

Level 2:
$X^{\delta}_{2,1} = \{x^{\delta}_{2,3}, x^{\delta}_{2,5}\}$
$X^{\delta}_{2,2} = \{x^{\delta}_{2,1}, x^{\delta}_{2,4}\}$
$X^{\delta}_{2,3} = \{x^{\delta}_{2,2}\}$

Level 3:
$X^{\delta}_{3,1} = \{x^{\delta}_{3,1}\}$
$X^{\delta}_{3,2} = \{x^{\delta}_{3,2}\}$

Nach Ermittlung der Variablenmengen $X^{\delta}_{i,v}$ für $i \in \{1, ..., n\}$ und $v \in D_i$ lassen sich die Kanten-Constraints C^{Arc} angeben:

Level 1:
$c^{Arc}_{1,1} = \quad (-1 * x^{B}_{1,1} \quad +1 * x^{\delta}_{1,1} = 0)$
$c^{Arc}_{1,2} = \quad (-1 * x^{B}_{1,2} \quad +1 * x^{\delta}_{1,2} = 0)$
$c^{Arc}_{1,3} = \quad (-1 * x^{B}_{1,3} \quad +1 * x^{\delta}_{1,3} = 0)$

Level 2:
$c^{Arc}_{2,1} = \quad (-1 * x^{B}_{2,1} \quad +1 * x^{\delta}_{2,3} \quad +1 * x^{\delta}_{2,5} = 0)$
$c^{Arc}_{2,2} = \quad (-1 * x^{B}_{2,2} \quad +1 * x^{\delta}_{2,1} \quad +1 * x^{\delta}_{2,4} = 0)$
$c^{Arc}_{2,3} = \quad (-1 * x^{B}_{2,3} \quad +1 * x^{\delta}_{2,2} = 0)$

Level 3:
$c^{Arc}_{3,3} = \quad (-1 * x^{B}_{3,1} \quad +1 * x^{\delta}_{3,1} = 0)$
$c^{Arc}_{3,4} = \quad (-1 * x^{B}_{3,2} \quad +1 * x^{\delta}_{3,2} = 0)$

Anhand der einzelnen $c^{Arc}_{i,v}$-Constraints lässt sich sehr gut erkennen, dass die Zuordnung eines Wertes $d_j \in D_i$ zu einer ursprünglichen Variable x_i, aufgrund von Voraussetzung 5.1 $x_i = d_j \leftrightarrow x^{B}_{i,j} = 1$, zur Folge hat, dass eine Kante mit Wert d_j in Level i Teil des Lösungspfades sein muss. Nimmt zum Beispiel die ursprüngliche Variable x_2 den Wert 2 an, so wird der booleschen Variable $x^{B}_{2,2}$ der Wert 1 zugeordnet, was aufgrund von $c^{Arc}_{2,2}$ zur Folge hat, dass *entweder der Variablen* $x^{\delta}_{2,1}$ *oder* $x^{\delta}_{2,4}$ *der Wert 1 zugewiesen wird. Diese Variablenzuweisung wiederum verdeutlicht, dass entweder der Übergang* $q_{1,0} \xrightarrow{2} q_{2,0}$ *oder der Übergang* $q_{1,1} \xrightarrow{2} q_{2,1}$ *Teil des Lösungspfades von M sein muss.*

Umgekehrt lässt sich ebenfalls nachvollziehen, dass, wenn der j-te Übergang $\delta_{i,j}$ *auf Level i Teil des Lösungspfades ist (repräsentiert durch die boolesche Variable* $x^{\delta}_{i,j}$*), aufgrund der* $c^{Arc}_{i,v}$*-Constraints auch die boolesche Variable* $x^{B}_{i,k}$ *den Wert 1 zugewiesen bekommen muss. Der Index k entspricht dabei dem Index des Wertes* $v = d_k \in D_i$ *von* $\delta_{i,j}$*. Ist beispielsweise der Übergang* $q_{1,1} \xrightarrow{2} q_{2,1}$ *Teil des Lösungspfades, so ergibt sich aufgrund von* $c^{Arc}_{2,2}$*, dass die boolesche Variable* $x^{B}_{2,2}$ *den Wert 1 annehmen muss, was gleichbedeutend damit ist, dass die ursprüngliche Variable* x_2 *den Wert 2 annimmt.*

Die Kanten-Constraints C^{Arc} *realisieren somit eine Bindung der einzelnen Übergänge an die zu den ursprünglichen Variablen äquivalenten booleschen Variablen. Die Knoten-Constraints* C^{Node} *sorgen anschließend dafür, dass nur Übergänge ausgewählt werden können, die einen zusammenhängenden Lösungspfad in M repräsentieren. Für jeden Zustand* $q_{i,j}$ *in M wird ein neues boolesches* Scalar*-Constraint* $c^{Node}_{i,j}$ *angelegt, das die eingehenden und ausgehenden Übergänge von* $q_{i,j}$ *in Relation setzt. Jedes* $c^{Node}_{i,j}$*-Constraint realisiert, dass jeder Knoten (abgesehen von Start und vom Endknoten) genau so viele eingehende, wie ausgehende Übergänge besitzen muss. Eine Festlegung der Anzahl der Übergänge auf eins wird dabei nicht explizit vorgenommen. Diese erfolgt aber aufgrund von Voraussetzung 5.2 und den* C^{Arc}*-Constraints. Für M ergeben sich basierend auf den Variablen-Übergängen-Zuordnungen aus Tabelle 5.4 die folgenden* C^{Node}*-Constraints:*

Level 0:

$$c^{Node}_{0,0} = (1 * x^{\delta}_{1,1} + 1 * x^{\delta}_{1,2} + 1 * x^{\delta}_{1,3} = 1)$$

Level 1:

$$\begin{aligned} c^{Node}_{1,0} &= (-1 * x^{\delta}_{1,1} \quad +1 * x^{\delta}_{2,1} \quad +1 * x^{\delta}_{2,2} = 0) \\ c^{Node}_{1,1} &= (-1 * x^{\delta}_{1,2} \quad +1 * x^{\delta}_{2,3} \quad +1 * x^{\delta}_{2,4} = 0) \\ c^{Node}_{1,2} &= (-1 * x^{\delta}_{1,3} \quad +1 * x^{\delta}_{2,5} = 0) \end{aligned}$$

Level 2:

$$\begin{aligned} c^{Node}_{2,0} &= (-1 * x^{\delta}_{2,1} \quad -1 * x^{\delta}_{2,3} \quad +1 * x^{\delta}_{3,1} = 0) \\ c^{Node}_{2,1} &= (-1 * x^{\delta}_{2,2} \quad -1 * x^{\delta}_{2,4} \quad -1 * x^{\delta}_{2,5} \quad +1 * x^{\delta}_{3,2} = 0) \end{aligned}$$

Level 3:

$$c^{Node}_{3,0} = (-1 * x^{\delta}_{3,1} \quad -1 * x^{\delta}_{3,2} = -1)$$

Durch Hinzufügen der Constraints aus C^{Arc} *und* C^{Node} *zu CSP 12 entsteht ein CSP, welches eine boolesche Skalarisierung des levelbasierten DFAs M bzw. eines dazu gehörenden* Regular*-Constraints darstellt.*

Beweis 5.7. ***Teil 1*** ($c \rightarrow sub(c)$)***:*** *Es muss gezeigt werden, dass mit Erfüllung des* Regular-*Constraints c auch die Constraints in C^{Arc} und C^{Node} erfüllt sind. Ist c erfüllt, so sind den Variablen $x_1, ..., x_n$ Werte $v_1, ..., v_n$ zugeordnet, so dass M das Wort $v_1 v_2 ... v_n$ als Eingabe akzeptiert. Demzufolge existiert ein Lösungspfad $q_{0,0} \xrightarrow{v_1} q_{1,j_1} \xrightarrow{v_2} ... \xrightarrow{v_n} q_{n,0}$ mit $q_{0,0} \in Q_0, q_{i,j} \in Q_i, i \in \{1, ..., n\}, j \in \{1, ..., |Q_i|\}$ und $v_i \in D_i$ in M. Per Definition beinhaltet ein Lösungspfad genau n viele Kanten und dabei je eine pro Level $i \in \{1, ..., n\}$.*

Jeder ursprünglichen Variable $x_i \in X$ wurde ein Wert v_i zugeordnet. Nach Voraussetzung 5.1 ist der booleschen Variable $x^B_{i,j} \in X^B$ mit $v_i = d_j \in D_i$ dadurch der Wert 1 und allen anderen booleschen Variablen $x^B_{i,k} \in X^B$ mit $v_i \neq d_k \in D_i$ der Wert 0 zugeordnet. Für die Constraints C^{Arc} ergeben sich dadurch zwei Fälle.

Fall 1*: Die Variable $x^B_{i,j}$ hat den Wert 1 zugeordnet bekommen. Der Lösungspfad hat dadurch beim i-ten Übergang den Wert $v_i = d_j \in D_i$. Die Variablen in X^{δ}_{i,v_i} repräsentieren gerade alle Übergänge in Level i mit Wert v_i. Da nur ein Übergang pro Level im Lösungspfad enthalten ist, folgt, dass genau eine Variable in X^{δ}_{i,v_i}, die den Übergang des Lösungspfads in Level i repräsentiert, den Wert 1 zugeordnet bekommen muss und alle anderen Variablen in X^{δ}_{i,v_i} den Wert 0. Somit lässt sich die Summe in den* Scalar-*Constraints in C^{Arc} auf $-1*1+0*1+...+0*1+1*1+0*1... = 0$ bzw. $-1+1=0$ kürzen, wodurch erkennbar wird, dass die Constraints in C^{Arc} für diesen Fall erfüllt sind.*

Fall 2*: Die Variable $x^B_{i,j}$ hat den Wert 0 zugeordnet bekommen. Der Lösungspfad hat dadurch beim i-ten Übergang einen Wert $d_j \in D_i$ mit $d_j \neq v_i$. Die Variablen in X^{δ}_{i,d_j} repräsentieren gerade alle Übergänge in Level i mit Wert d_j. Keiner dieser Übergänge ist Teil des Lösungspfades. Daraus folgt, dass alle Variablen in X^{δ}_{i,d_j} den Wert 0 zugeordnet bekommen. Somit lässt sich die Summe in den* Scalar-*Constraints in C^{Arc} auf $-1*0+0*1+...+0*1=0$ bzw. $0=0$ kürzen, wodurch erkennbar wird, dass die Constraints in C^{Arc} für diesen Fall erfüllt sind.*

Da ein Lösungspfad $q_{0,0} \xrightarrow{v_1} q_{1,j_1} \xrightarrow{v_2} ... \xrightarrow{v_n} q_{n,0}$ in M instantiiert wurde, folgt für jeden Zustand $q_{i,j} \in Q$, der Teil des Lösungspfades ist, dass sowohl ein eingehender (abgesehen vom Startzustand) als auch ein ausgehender Übergang (abgesehen von finalen Zustand) von $q_{i,j}$ Teil des Lösungspfades sind. Da die Variablen in $X^{in}_{i,j}$ alle eingehenden und die Variablen in $X^{out}_{i,j}$ alle ausgehenden Übergänge von $q_{i,j}$ repräsentieren und genau dann den Wert 1 zugeordnet bekommen, wenn diese Teil des Lösungspfades sind, folgt daraus, dass genau einer Variablen in $X^{in}_{i,j}$ und einer Variablen in $X^{out}_{i,j}$ der Wert 1 zugeordnet wird. Somit lässt sich die Summe in den Scalar-*Constraints in C^{Node} für $i \in \{1, ..., n_1\}$ auf $-1*0+...-1*1-1*0+...-1*0+0*1+...+0*1+1*1+0*1+...+0*1=0$ bzw. $-1+1=0$ kürzen, wodurch erkennbar wird, dass die Constraints in C^{Node} für diesen Fall erfüllt sind.*

Für jeden Zustand $q_{i,j} \in Q$, der nicht Teil des Lösungspfades ist, folgt, dass weder ein eingehender, noch ein ausgehender Übergang von $q_{i,j}$ Teil des Lösungspfades ist. Dadurch lässt sich die Summe in den Scalar-*Constraints in C^{Node} für $i \in \{1, ..., n_1\}$ auf $-1*0+...-1*0+0*1+...+0*1=0$*

bzw. $0 = 0$ *kürzen, wodurch erkennbar wird, dass die Constraints in* C^{Node} *für diesen Fall erfüllt sind. Die Gültigkeit von* $c^{Node_{0,0}}$ *und* $c^{Node_{n,0}}$ *kann analog nachgewiesen werden, wodurch gezeigt werden konnte, dass alle Constraints in* C^{Arc} *und* C^{Node} *erfüllt sein müssen, wenn das gegebene* Regular-*Constraint* c *erfüllt ist.* □

Teil 2 *(sub(c) → c)*: *Es ist zu zeigen, dass bei Erfüllung der Constraints in* C^{Arc} *und in* C^{Node} *auch das* Regular-*Constraint* c *erfüllt ist. In jedem Constraint* $c^{Arc}_{i,v}$ *kann, da die boolesche Variable* $x^B_{i,j}$ *nur den Wert 1 oder 0 zugeordnet bekommen kann, auch nur eine bzw. keine Variable in* $X^\delta_{i,v}$ *den Wert 1 annehmen. Nach Voraussetzung 5.2 ist genau einer Variablen* $x^B_{i,j} \in \{x^B_{i,1}, ..., x^B_{i,|D_i|}\}$ *der Wert 1 zugewiesen, woraus folgt, dass auch genau einer Variablen* $x^\delta_{i,j} \in \{x^\delta_{i,1}, ..., x^\delta_{i,|\delta_i|}\}$ *der Wert 1 zugeordnet sein muss. Da die Variablen* $x^\delta_{i,k_i} \in X^{delta}$ *Übergänge* $e \in \delta$ *in* M *repräsentieren, die genau dann zum Lösungspfad gehören, wenn die jeweilige Variable* x^δ_{i,k_i} *mit Wert 1 belegt ist, folgt, dass genau ein Übergang* $e_{i,k_i} = (q_{i-1,j_1}, d_{i,j_2}, q_{i,j_3})$ *pro Level* 1 *bis* n *Teil des Lösungspfades ist. Aufgrund von Voraussetzung 5.1 ist somit jeder Variablen* $x_i \in X$ *der Wert* $d_{i,j_2} \in D_i$ *zugeordnet.*

Es konnte bisher gezeigt werden, dass, wenn die Voraussetzungen 5.1 und 5.2 und die Constraints in C^{Arc} *erfüllt sind, jeder ursprünglichen Variable* $x_i \in X$ *ein Wert* d_{i,j_i} *zugeordnet wurde und jede dieser Variablenzuordnungen einen Übergang* e_{i,k_i} *eines unterschiedlichen Levels* $i \in \{1, ..., n\}$ *widerspiegelt. Wird jetzt noch gezeigt, dass diese* n *Übergänge einen zusammenhängenden Pfad vom Startzustand* $q_{0,0}$ *zum finalen Zustand* $q_{n,0}$ *darstellen, so konnte gezeigt werden, dass das* Regular-*Constraint* c *erfüllt ist.*

Aus den bisherigen Betrachtungen geht hervor, dass für jedes $i \in \{1, ...n\}$ *jeweils genau eine boolesche Variable* $x^\delta_{i,j} \in \{x^\delta_{i,1}, ..., x^\delta_{i,|\delta_i|}\}$ *den Wert 1 angenommen hat. Da die Variablenmengen* $X^{in}_{i,j}$ *disjunkte Teilmengen von* X^δ *in Level* i *sind, hat auf jedem Level* $i \in \{1, ..., n-1\}$ *genau eine Variable in den Mengen* $X^{in}_{i,j}$ *für* $j \in \{1, ..., |Q_i|\}$ *den Wert 1 zugeordnet. Analog hat in den Mengen* $X^{out}_{i,j}$ *für* $j \in \{1, ..., |Q_i|\}$ *genau eine Variable den Wert 1 zugewiesen bekommen. Dementsprechend existiert pro Level* $i \in \{1, .., n-1\}$ *genau ein Constraint* $c^{Node_{i,j}}$*, bei dem in der Menge* $X^{in}_{i,j}$ *eine Variable* x_{i-1,j_1} *enthalten ist, die mit Wert 1 belegt wurde. Um die Summe in* $c^{Node_{i,j}}$ *zu erfüllen, muss demnach genau eine Variable* x_{i,j_2} *in* $X^{out}_{i,j}$ *den Wert 1 annehmen. Die zugehörigen Übergänge* e_{i-1,j_1} *und* e_{i,j_2} *sind demzufolge beide Teil des Lösungspfades und zusammenhängend. Da dies für alle* $i \in \{1, ..., n-1\}$ *gilt, folgt daraus, dass der gefundene Lösungspfad zusammenhängend sein muss.* □

5.2.9 Beispiele für die Effektivität spezieller Constraint-Substituierungen

Nachdem in Abschnitt 5.1 zunächst die allgemeinen Transformationen mit Beispielen eingeführt wurden und in diesem Abschnitt 5.2 direkte Transformationen von verschiedenen globalen Constraints in *Regular*-Constraints und boolesche *Scalar*-Constraints erläutert und diskutiert wurden, werden im Folgenden für die direkten Transformationen Beispiele gegeben und ihre Effektivität betrachtet.

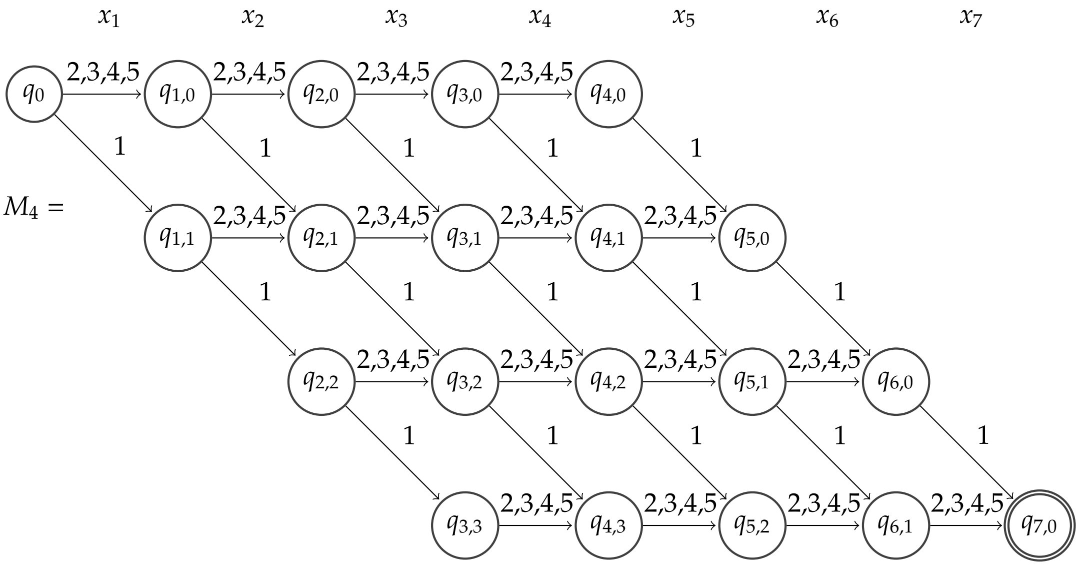


Abbildung 5.2: Der zum Constraint c_4 äquivalente minimale levelbasierte DFA M_4.

Das CSP 8 aus Beispiel 4.1 wird an dieser Stelle erneut verwendet, um die Effektivität der direkten Substituierung des *Sum*- und des *Count*-Constraints durch ein *Regular*-Constraint bzw. eine Menge von booleschen *Scalar*-Constraints zu demonstrieren. Die *Sum*-Constraints $c_2 = (x_1 + x_2 = x_3)$ und $c_5 = (x_4 + x_5 + x_6 + x_7 \geq 8)$ werden ebenso wie das *Count*-Constraint $c_4 = Count(X, 1, 3)$ direkt transformiert. Das Constraint $c_1 = (x_1 \neq x_2)$ kann sowohl als *AllDifferent*- als auch als *Sum*-Constraint betrachtet werden. An dieser Stelle wird es wie ein *Sum*-Constraint substituiert. Für das verbleibende Constraint $c_3 = (x_1 * x_2 > x_3)$ wurde keine direkte Transformation angegeben, so dass stattdessen die allgemeine Substituierung (siehe Abschnitt 5.1) verwendet wird.

Es wird zunächst auf die einzelnen Transformationen eingegangen, beginnend mit der Regularisierung, gefolgt von der booleschen Skalarisierung, bevor eine Auswertung der transformierten CSPs erfolgt. Zum Lösen der im Beispiel dargestellten CSPs wurde jeweils der Choco-Solver mit der in Abschnitt 1.4 angegebenen Suchstrategie, Software und Hardware verwendet.

Die Regularisierung: Das gegebene Constraint $c_4 = Count(\{x_1, x_2, ..., x_7\}, 1, 3)$ mit $D_1 = D_2 = ... = D_7 = \{1, 2, 3, 4, 5\}$, welches für die Variablen $x_1, x_2, ..., x_7$ fordert, dass genau drei davon der Wert 1 zugeordnet wird, kann nach [95] und [1] (bzw. Abschnitt 5.2.1) in den in Abbildung 5.2 dargestellten minimalen levelbasierten DFA M_4 überführt werden. Das Constraint $Regular(\{x_1, x_2, ..., x_7\}, M_4)$ dient dann als Substituierung für c_4.

Die *Sum*-Constraints c_1, c_2 und c_5 werden mittels dem in [151] angegebenen Algorithmus, unter Berücksichtigung der in 5.2.5 aufgeführten Anpassungen, regularisiert. Die resultierenden levelbasierten DFAs M_1, M_2 und M_5 sind in Abbildung 5.3 dargestellt.

In Abbildung 5.4 ist der durch allgemeine Regularisierung (siehe Abschnitt 5.1.1) erzeugte minimale levelbasierte DFA M_3 für c_3 dargestellt. Das CSP 13 kann nun als Regularisierung für das ursprüngliche CSP 8 verwendet werden.

CSP 13: $P^R = (X, D, C)$ mit:

$X = \{x_1, ..., x_7\}$ (7 Variablen)

$D = \{D_1 = ... = D_7 = \{1,2,3,4,5\}\}$ (7 Domänen)

$C = \{c_1^r = Regular(\{x_1, x_2\}, M_1),$ (Regularisierungen von c_1 bis c_5)

$c_2^r = Regular(\{x_1, x_2, x_3\}, M_2),$

$c_3^r = Regular(\{x_1, x_2, x_3\}, M_3),$

$c_4^r = Regular(\{x_1, x_2, ..., x_7\}, M_4),$

$c_5^r = Regular(\{x_4, x_5, x_6, x_7\}, M_5)\}$

Die boolesche Skalarisierung: Bei der booleschen Skalarisierung wird für die Substituierung der Constraints c_1, c_2 und c_5 die boolesche Skalarisierung von *Sum*-Constraints, für c_4 die boolesche Skalarisierung von *Count*-Constraints und für c_3 die Support-basierte boolesche Skalarisierung verwendet. Das CSP P^{BS} stellt das durch die aufgeführten Substituierungen erzeugte boolesche *Scalar*-CSP des ursprünglichen CSP 8 dar.

CSP 14: $P^{BS} = (X^B, D^B, C^S)$ mit:

$X^B = \{x^B_{1,1}, x^B_{1,2}, x^B_{1,3}, x^B_{1,4}, x^B_{1,5}, x^B_{2,1}, x^B_{2,2}, ..., x^B_{7,5}, x^C_{3,1}, x^C_{3,2}, ..., x^C_{3,98}\}$

$D^B = \{D^B_{1,1} = ... = D^B_{7,5} = D^C_{3,1} = ... = D^C_{3,98} = \{0,1\}\}$

$C^S = \{c_1^{oneOff} = Scalar(\{x^B_{1,1}, x^B_{1,2}, x^B_{1,3}, x^B_{1,4}, x^B_{1,5}\}, (1,1,1,1,1)^T, =1),$

$... ,$

$c_7^{oneOff} = Scalar(\{x^B_{7,1}, x^B_{7,2}, x^B_{7,3}, x^B_{7,4}, x^B_{7,5}\}, (1,1,1,1,1)^T, =1),$

$c_1^S = (x^B_{1,1} + x^B_{1,2} + x^B_{1,3} + x^B_{1,4} + x^B_{1,5} - x^B_{2,1} - x^B_{2,2} - x^B_{2,3} - x^B_{2,4} - x^B_{2,5} \neq 0),$

// *Sum*-Constraint, siehe Abschnitt 5.2.6

$c_2^S = (x^B_{1,1} + x^B_{1,2} + x^B_{1,3} + x^B_{1,4} + x^B_{1,5} + x^B_{2,1} + x^B_{2,2} + x^B_{2,3} + x^B_{2,4} + x^B_{2,5}$

$-x^B_{3,1} - x^B_{3,2} - x^B_{3,3} - x^B_{3,4} - x^B_{3,5} \neq 0),$

// *Sum*-Constraint, siehe Abschnitt 5.2.6

$c_{3,1}^S = Scalar(\{x_{1,1}, x_{2,2}, x_{3,1}, x^C_{3,1}\}, (1,1,1,3)^T, \geq, 3),$

$... ,$

$c_{3,98}^S = Scalar(\{x_{1,5}, x_{2,5}, x_{3,5}, x^C_{3,98}\}, (1,1,1,3)^T, \geq, 3),$

$c_3^{oneNot} = Scalar(\{x^C_{3,1}, x^C_{3,2}, ..., x^C_{3,98}\}, (1,1,...,1)^T, =, 97),$

// Support-basierte boolesche Skalarisierung, siehe Abschnitt 5.1.2

$c_4^S = (x^B_{1,1} + x^B_{2,1} + x^B_{3,1} + x^B_{4,1} + x^B_{5,1} + x^B_{6,1} + x^B_{7,1} = 3),$

// *Count*-Constraint, siehe Abschnitt 5.2.1

$c_5^S = (x^B_{4,1} + x^B_{4,2} + x^B_{4,3} + x^B_{4,4} + x^B_{4,5} + x^B_{5,1} + x^B_{5,2} + x^B_{5,3} + x^B_{5,4} + x^B_{5,5}$

$x^B_{6,1} + x^B_{6,2} + x^B_{6,3} + x^B_{6,4} + x^B_{6,5} + x^B_{7,1} + x^B_{7,2} + x^B_{7,3} + x^B_{7,4} + x^B_{7,5} \geq 8)\}$

// *Sum*-Constraint, siehe Abschnitt 5.2.6

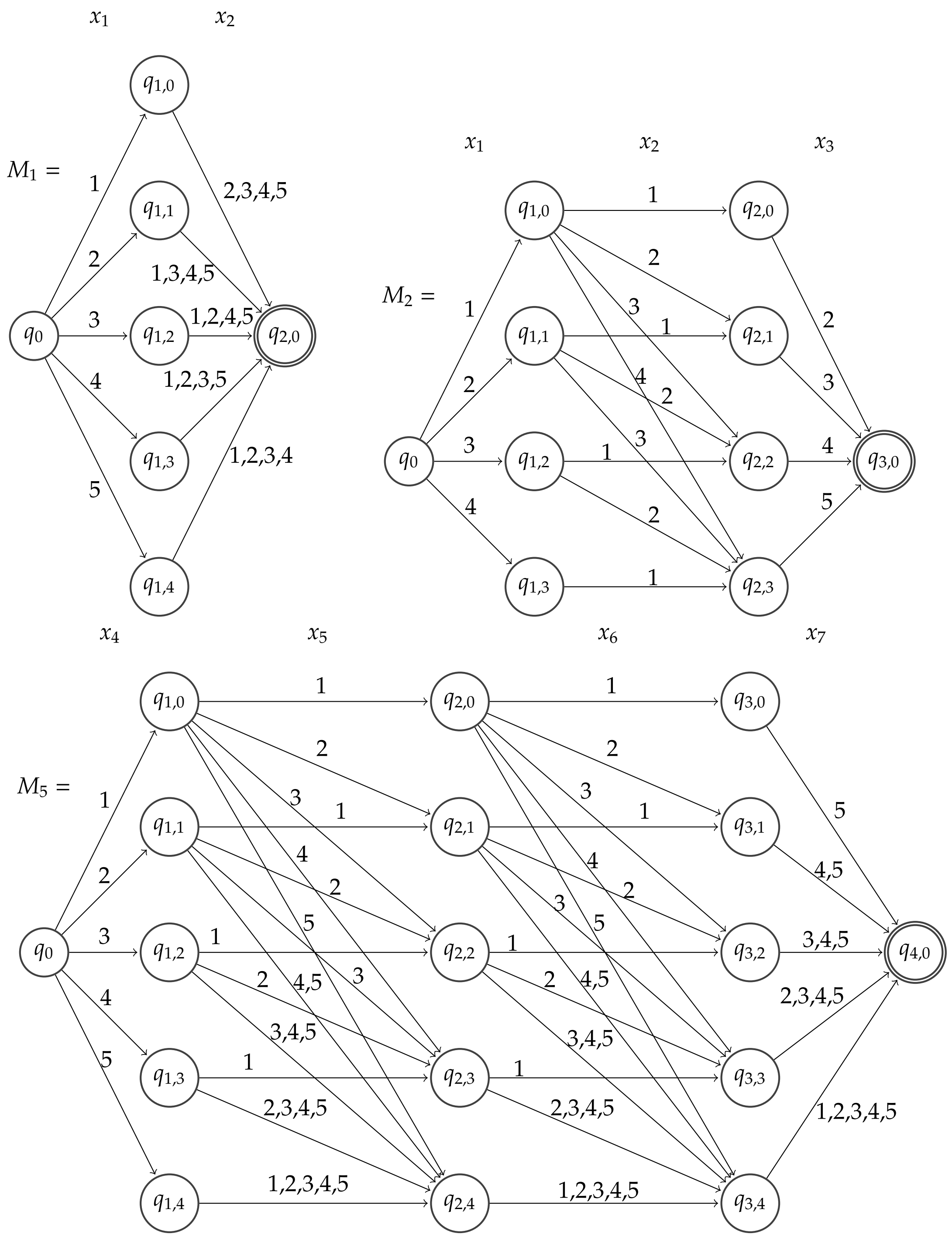


Abbildung 5.3: Die zu den *Sum*-Constraints c_1, c_2 und c_5 äquivalenten minimalen levelbasierten DFAs M_1, M_2 und M_5.

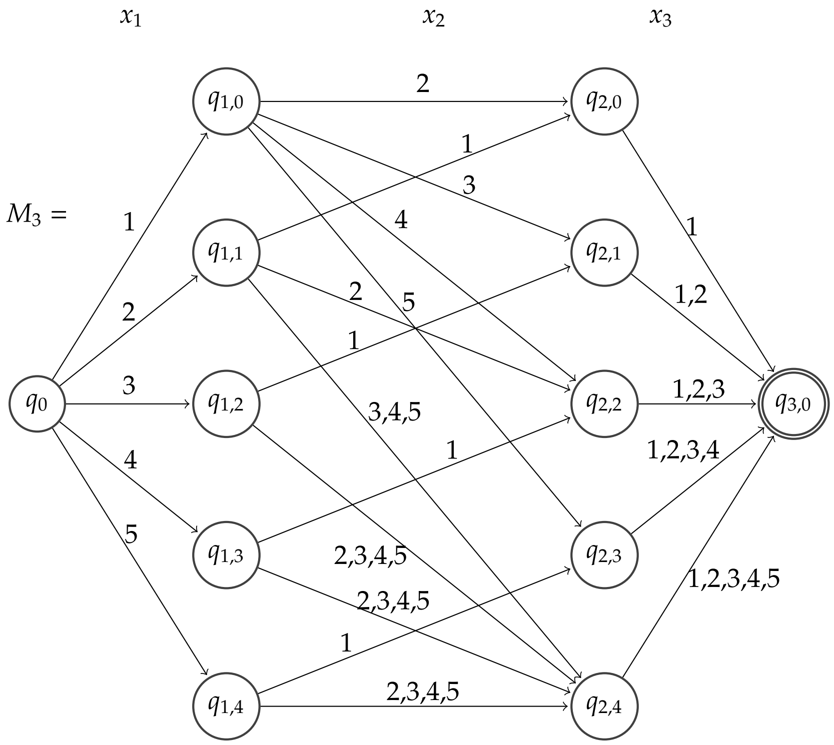


Abbildung 5.4: Der zu dem Constraint $c_3 = (x_1 * x_2 > x_3)$ äquivalente minimale levelbasierte DFA M_3.

Die Constraints c_1, c_2, c_4 und c_5 können dabei direkt in jeweils ein *Scalar*-Constraint transformiert werden ($c_1^S, c_2^S, c_4^S, c_5^S$). Das Constraint c_3 wird durch Support-basierte boolesche Skalarisierung in 98 Support-Constraints $c_{3,1}^S, ..., c_{3,98}^S$ und ein *oneNot*-Constraint c_3^{oneNot} überführt, welches sicherstellt, dass nur eines der 98 erlaubten Tupel erfüllt ist. Des Weiteren wird, wie immer bei der booleschen Skalarisierung, für jede der sieben ursprünglichen Variablen ein *oneOff*-Constraint $c_1^{oneOff}, ..., c_7^{oneOff}$ angelegt. Aufgrund der 98 Support-Constraints $c_{3,1}^S, ..., c_{3,98}^S$ werden 98 neue boolesche Variablen $x_{3,1}^C, ..., x_{3,98}^C$ benötigt, die zur Variablenmenge hinzugefügt werden.

Auswertung der direkten Transformationen

Nachdem die Transformation von CSP 8 in das *Regular*-CSP 13 und das boolesche *Scalar*-CSP 14 vorgestellt wurde, soll im Folgenden deren Transformations- und Lösungsgeschwindigkeit mit den vorherigen Ansätzen verglichen werden. Für die direkten Transformationen wurde die Namensgebung aus Beispiel 4.1 um den V-Wert „*D*", welcher für die **d**irekte Transformation steht, erweitert. Die Namensgebung setzt sich somit wie folgt zusammen:

$$Substituierungsart(Gruppe1^{V_1}, Gruppe2^{V_2}, ...).$$

Dabei ist die Substituierungsart entweder T = *Tabularisierung*, R = *Regularisierung* oder B = *boolesche Skalarisierung*. Die Gruppen geben an, welche ursprünglichen Constraints durch ein gemeinsames Constraint substituiert werden und die V-Werte (Varianten) spezifizieren die Substituierung der jeweiligen Gruppe. Als Spezifikation sind die Werte 1 oder 2 für die jeweilige in Abschnitt 4.1.1 vorgestellte Variante 1 bzw. 2 sowie K oder S für konfliktbasiert oder Support-basiert und D für direkt transformiert möglich. Für zu substituierende Constraint-Gruppen, die nur ein Constraint enthalten und mittels Variante 1 oder 2 regularisiert oder tabularisiert werden sollen, wird auf die Angabe eines V-Wertes verzichtet, da in diesem Fall beide Varianten identisch agieren.

In Tabelle 5.5 sind die durchschnittlichen Zeitaufwände in Sekunden für das Erstellen $t(create_P)$, Transformieren $t(sub)$ und Lösen $t(solve(P^t))$, die akkumulierte Zeit $t(total)$ sowie die Anzahl der genutzten Variablen und Constraints von *CSP* 8 bei 1000 Durchläufen dargestellt. Es sind dabei jeweils untereinander die Werte für das CSP 8 (Zeile 1), nach Verwendung der allgemeinen Substituierungsverfahren (nach Abschnitt 5.1: Regularisierung (Zeile 2), konfliktbasierter boolescher Skalarisierung (Zeile 4) und Support-basierter boolescher Skalarisierung (Zeile 6)) und die Werte für die jeweiligen direkten Substituierungen (Zeilen 3, 5 und 7) angegeben. Die beiden Varianten $B(\{\{c_1\}^D, \{c_2\}^D, \{c_3\}^K, \{c_4\}^D, \{c_5\}^D\})$ (Zeile 5) und $B(\{\{c_1\}^D, \{c_2\}^D, \{c_3\}^S, \{c_4\}^D, \{c_5\}^D\})$ (Zeile 7) unterscheiden sich in der Umsetzung nur darin, dass für das Constraint $c_3 = (x_1 * x_2 > x_3)$, für das keine direkte Transformation angegeben wurde, einmal die konfliktbasierte und einmal die Support-basierte boolesche Skalarisierung durchgeführt wurde.

	Variante	$t(create_P)$	$t(sub)$	$t(solve(P^t))$	$t(total)$	#Vars	#Cons
1	CSP 8	0,1710	0,0000	0,0160	0,1870	8	6
2	$R(\{\{c_1\},\{c_2\},\{c_3\},\{c_4\},\{c_5\}\})$	0,1743	0,4066	0,0039	0,5848	7	5
3	$R(\{\{c_1\}^D,\{c_2\}^D,\{c_3\}^D,\{c_4\}^D,\{c_5\}^D\})$	0,1700	0,0398	0,0035	0,2133	7	5
4	$B(\{\{c_1\}^K,\{c_2\}^K,\{c_3\}^K,\{c_4\}^K,\{c_5\}^K\})$	0,1703	0,7666	1,8173	2,7542	35	69354
5	$B(\{\{c_1\}^D,\{c_2\}^D,\{c_3\}^K,\{c_4\}^D,\{c_5\}^D\})$	0,1762	0,0257	0,0131	0,2150	35	38
6	$B(\{\{c_1\}^S,\{c_2\}^S,\{c_3\}^S,\{c_4\}^S,\{c_5\}^S\})$	0,1723	0,4455	2,1171	2,7349	9831	9808
7	$B(\{\{c_1\}^D,\{c_2\}^D,\{c_3\}^S,\{c_4\}^D,\{c_5\}^D\})$	0,1700	0,0351	0,0193	0,2244	133	110

Tabelle 5.5: Zeitaufwände für verschiedene Varianten der direkten Regularisierung und der booleschen Skalarisierung.

Es ist zu erkennen, dass die reine Transformationszeit $t(sub)$ in allen drei Fällen durch die direkte Transformation deutlich verringert werden konnte (Vergleich jeweils von Zeile 2 mit 3, 4 mit 5 und 6 mit 7). Bei der Regularisierung konnte die Transformationszeit auf ca. ein Zehntel, bei der konfliktbasierten booleschen Skalarisierung auf ca. ein Dreißigstel und bei der Support-basierten booleschen Skalarisierung auf ca. ein Zwölftel reduziert werden. Bei der konfliktbasierten booleschen Skalarisierung (Zeilen 4 und 5) ist des Weiteren zu erkennen, dass die Anzahl der benötigten *Scalar*-Constraints signifikant reduziert werden konnte (von 69354 auf 38). Bei der Support-basierten booleschen Skalarisierung (Zeilen 6 und 7) konnte die Anzahl der benötigten booleschen Variablen (von 9831 auf 133) und *Scalar*-Constraints (von 9808 auf 110) ähnlich signifikant reduziert werden. Die Reduzierung der benötigten Constraints (und im zweiten Fall auch der Variablen) zieht eine deutliche Reduktion der Lösungsdauer nach sich, so dass die booleschen Skalarisierungen das CSP ähnlich schnell lösen können wie die Originalversion (Zeile 1, Vergleich der $t(solve(P^t)$-Werte).

Im Gegensatz zur Regularisierung konnte die benötigte Zeit zum Lösen des CSPs $t(solve(P^t))$ bei der booleschen Skalarisierung, im Vergleich zum Eingabe-CSP, nicht deutlich reduziert werden (Vergleich der Zeile 1 mit den Zeilen 5 und 7). An dieser Stelle wurde bisher allerdings weiterhin der Choco-Solver zum Lösen des transformierten Problems verwendet, welcher keine Spezialisierung auf boolesche *Scalar*-CSPs anbietet. Es kann daher vermutet werden, dass die Verwendung eines linearen pseudo-booleschen Solvers bei den in boolesche *Scalar*-CSPs transformierten Problemen zu einer deutlich erhöhten Lösungsgeschwindigkeit führt.

Die Regularisierung führt bei Anwendung des allgemeinen Algorithmus und der direkten Substituierung (Zeilen 2 und 3) gleichermaßen zu minimalen levelbasierten DFAs. Bei der booleschen Skalarisierung hingegen verringert die direkte Transformation in der Regel die Größe (Anzahl Constraints oder Anzahl Constraints und Variablen) des resultierenden CSPs (Zeilen 4 und 5 sowie 6 und 7). Dieses Verhalten hat zur Folge, dass die reine Lösungsgeschwindigkeit bei der Regularisierung relativ konstant bleibt, wohingegen sie sich bei der booleschen Skalarisierung signifikant unterscheidet. Folglich ist ein boolesches *Scalar*-CSP, bei dem nicht alle Constraints direkt in boolesche *Scalar*-Constraints überführt wurden, in der Regel langsamer zu lösen als ein äquivalentes, bei dem alle Constraints direkt überführt werden konnten (Zeilen 4 und 5 sowie 6 und 7). Dieser Annahme zur Folge kann die Lösungsgeschwindigkeit (und auch die Transformationsgeschwindigkeit) der booleschen Skalarisierungen für CSP 8 noch weiter signifikant erhöht werden, wenn eine direkte Transformation für $c_3 = (x_1 * x_2 > x_3)$ gefunden werden kann. Die boolesche Skalarisierung scheint somit besonders erstrebenswert, wenn ein linearer pseudo-boolescher Solver verwendet wird und die jeweiligen CSPs nur Constraints enthalten, die direkt substituiert werden können.

Nachdem für ein spezielles Beispiel der Nutzen der direkten Transformationen aufgezeigt wurde, folgt in Kapitel 6 sowohl eine Evaluation der einzelnen direkten Transformationen als auch eine Auswertung des Einflusses der Verwendung von direkten Transformationen in realistischen Problemstellungen.

5.3 Die Substituierung logischer Meta-Constraints

Im vorherigen Abschnitt wurden Substituierungen für globale Constraints präsentiert, die häufig für die Modellierung realer Probleme genutzt werden. Eine andere Art von Constraints, die für die Modellierung praktischer Probleme relevant ist, sind logische Constraints wie zum Beispiel die Implikation (aus A folgt B: $A \rightarrow B$). Als logische Constraints werden in dieser Arbeit Constraints bezeichnet, die logische Operationen wie *not* ($\neg x$), *and* ($x \wedge y$) und *or* ($x \vee y$) über booleschen Variablen (hier x und y) repräsentieren. Dies beinhaltet auch aus diesen Operatoren kombinierbare Operatoren, wie das logische *and* ($\bigwedge$) oder *or* ($\bigvee$) über mehr als zwei Variablen oder die Implikation ($x \rightarrow y$) oder Äquivalenz ($x \Leftrightarrow y$) von zwei Variablen (x und y).

Zusätzlich zu logischen Constraints, die boolesche Variablen als Eingaben erhalten, wird bei der Modellierung realer Probleme, wie dem in Abschnitt 2.5 vorgestelltem Schichtplanungsproblem (Beispiel 2.5), auch auf logische Constraints zugegriffen, welche Constraints als Eingabe erhalten. Letztere solche Constraints werden als *Meta-Constraints* bezeichnet. Logische Meta-Constraints sind demnach Constraints, die logische Operatoren über Constraints repräsentieren. Ein $or(c_1, c_2)$-Constraint mit den Constraints c_1 und c_2 als Eingabe fordert zum Beispiel, dass c_1 oder c_2 erfüllt sein muss. Die Implikation $c_1 \rightarrow c_2$ fordert, dass c_2 erfüllt sein muss, wenn c_1 erfüllt ist.

Die Substituierung logischer Constraints über booleschen Variablen ist in der Regel trivial und kann über Aufzählen der gültigen oder ungültigen Tupel erfolgen. Dementsprechend können die allgemeinen Substituierungen aus Abschnitt 5.1 verwendet werden. Die Substituierung von logischen Meta-Constraints hingegen ist komplexer und erfolgt im Folgenden in zwei Schritten.

1. Substituieren und Binden der Eingabe-Constraints an entsprechende boolesche Variablen
2. Umwandeln des logischen Meta-Constraints in ein logisches Constraint über booleschen Variablen

Im ersten Schritt werden die Eingabe-Constraints $c_1, ..., c_n$ des logischen Meta-Constraints C^{Meta} zunächst durch Regularisierung oder durch boolesche Skalarisierung substituiert. Es entstehen dabei bei der Regularisierung *Regular*-Constraints $c_1^R, ..., c_n^R$ und bei der booleschen Skalarisierung Mengen von booleschen *Scalar*-Constraints $C_1^{BS}, ..., C_n^{BS}$. Anschließend wird die Erfüllung der *Regular*-Constraints c_i^R bzw. der Mengen der booleschen *Scalar*-Constraints C_i^{BS} an boolesche Variablen x_i^{reif} gebunden. Ist ein Constraint oder eine Menge von Constraints erfüllt, nimmt die gebundene Variable x_i^{reif} den Wert 1 an, andernfalls den Wert 0. Umgekehrt muss das gebundene Constraint bzw. die gebundene Constraint-Menge erfüllt sein, wenn die Variable x^{reif} den Wert 1 annimmt und ungültig, wenn die Variable x^{reif} den Wert 0 annimmt. In der Literatur wird dieses Binden einer Variable an die Erfüllbarkeit eines Constraints oder einer Constraint-Menge *Reification* genannt [30]. Nachdem die ursprünglichen Eingabe-Constraints $c_1, ..., c_n$ substituiert und an boolesche Variablen gebunden wurden (1.), wird darauf folgend das ursprüngliche Meta-Constraint c^{Meta} durch ein logisches Constraint c^{Logic} mit den booleschen Variablen $x_1^{reif}, ..., x_n^{reif}$ ersetzt, welches die logische Relation (z.B. *or*, *and*, *Implikation* etc.) des ursprünglichen Meta-Constraints c^{Meta} erhält (2.). Nachfolgend wird zunächst auf die Transformation von Meta-Constraints mit Hilfe der Regularisierung und anschließend auf eine entsprechende Substitution mittels boolescher Skalarisierung eingegangen.

5.3.1 Die Regularisierung

Die Erfüllung eines *Regular*-Constraints $Regular(\{x_1, ..., x_n\}, M)$ mit minimalem levelbasierten DFA M kann an eine Variable x^{reif}, die die Erfüllung des Constraints widerspiegelt, gebunden werden. Hierzu wird die Variable x^{reif} hinten an die geordnete Variablenmenge $\{x_1, ..., x_n\}$ angefügt und der levelbasierte DFA $M = (Q, \Sigma, \delta, q_{0,0}, F = \{q_{n,0}\})$ wie folgt angepasst:

- Das neue Alphabet Σ' enthält alle Zeichen aus $\Sigma \cup \{0, 1\}$.
- Die neue Zustandsmenge Q' enthält alle Zustände aus Q und $(n + 1)$ neue Zustände $q'_1, ..., q'_{n+1}$, welche Fehlerzustände des ursprünglichen levelbasierten DFAs repräsentieren.
- Der Startzustand $q_{0,0}$ von M ist auch in M' der Startzustand.
- Der Automat M' enthält ein Level mehr als M. Jeder alte Zustand $q \in Q$ hat in Q' das gleiche Level wie in Q. Jeder neue Zustand q'_i für $i \in \{1, ..., n + 1\}$ wird dem Level i und somit der Zustandsmenge Q'_i zugeordnet.

- Die Menge der finalen Zustände $F = \{q_{n,0}\}$ wird durch $F' = \{q'_{n+1}\}$ ersetzt.
- Die neue Menge der Übergänge δ' beinhaltet alle Übergänge aus δ und wird um die folgenden Übergänge erweitert:
  - alle Übergänge $(q'_{i-1}, d_i) \rightarrow q'_i, \forall i \in \{2, ..., n\}, d_i \in D_i$, wobei jedes D_i gerade die Domäne der Variable x_i repräsentiert,
  - alle Übergänge, $(q_{i,j}, d_i) \rightarrow q'_{i+1}$ mit $q_{i,j} \in Q, d_i \in D_i$, für die nicht bereits ein Übergang von $q_{i,j}$ mit Wert d_i in δ existiert,
  - einen Übergang $(q_{n,0}, 1) \rightarrow q'_{n+1}$ und
  - einen Übergang $(q'_n, 0) \rightarrow q'_{n+1}$

Das Constraint $Regular(\{x_1, ..., x_n, x^{reif}\}, M')$ mit dem neuen levelbasierten DFA $M' = (Q', \Sigma', \delta', F', q_0)$ bindet die Variable x^{reif} an das Erreichen eines finalen Zustands in M. Der beschriebene levelbasierte DFA M' kann Zustände q'_i enthalten, die nicht erreichbar sind. Nach Eliminierung dieser nicht erreichbaren Zustände und unter der Voraussetzung, dass der ursprüngliche levelbasierte DFA M minimal ist, ist auch M' minimal. Im folgendem Beispiel 5.7 wird die Automatenbildung M' veranschaulicht, bevor im Anschluss ein Beweis 5.8 für die Korrektheit der Automatenbildung gegeben wird.

Beispiel 5.7: Die Bindung der Erfüllbarkeit eines *Regular*-Constraints an eine boolesche Variable. *Gegeben sei der Automat* $M_2 = (Q = \{q_{0,0}, q_{1,0}, q_{1,1}, q_{1,2}, q_{2,0}, q_{2,1}, q_{2,2}, q_{3,0}\}$, $\Sigma = \{1, 2, 3\}$, $\delta = \{(q_0, 1, q_{1,0}), (q_0, 2, q_{1,1}), (q_0, 3, q_{1,2}), (q_{1,0}, 2, q_{2,0}), (q_{1,0}, 3, q_{2,1}), (q_{1,1}, 1, q_{2,0}), (q_{1,1}, 3, q_{2,2}), (q_{1,2}, 1, q_{2,1}), (q_{1,2}, 2, q_{2,2}), (q_{2,0}, 3, q_{3,0}), (q_{2,1}, 2, q_{3,0}), (q_{2,2}, 1, q_{3,0})\}$, $F = \{q_{3,0}\}$, $q_{0,0})$ *aus Beispiel 4.4, welcher ein* AllDifferent-*Constraint über den Variablen* x_1, x_2 *und* x_3 *mit Domänen* $D_1 = D_2 = D_3 = \{1, 2, 3\}$ *realisiert. Der neue levelbasierte DFA* $M' = (Q', \Sigma', \delta', F', q_{0,0})$ *bildet sich wie folgt:*

- *Das neue Alphabet* Σ' *ergibt sich aus dem alten Alphabet* Σ *vereinigt mit* $\{0, 1\}$: $\Sigma' = \{1, 2, 3\} \cup \{0, 1\} = \{0, 1, 2, 3\}$.
- *Die neue Zustandsmenge* Q' *ergibt sich aus der alten Zustandsmenge* Q, *zu der die neuen Zustände* $\{q'_1, q'_2, q'_3, q'_4\}$ *hinzugefügt werden:* $Q' = \{q_{0,0}, q_{1,0}, q_{1,1}, q_{1,2}, q'_1, q_{2,0}, q_{2,1}, q_{2,2}, q'_2, q_{3,0}, q'_3, q'_4\}$.
- *Die Menge der finalen Zustände ist gleich* $F' = \{q'_4\}$.
- *Die Menge der neuen Übergänge* δ' *ergibt sich aus:*
  - *den bisherigen Übergängen* δ *des Automaten* M_2,
  - *plus den Übergängen* (q'_{i-1}, d_i, q'_i) *mit* $i \in \{2, 3\}$ *und* $d_i \in D_i$, *wobei* D_2 *die Domäne der Variable* x_2 *und* D_3 *die Domäne der Variable* x_3 *ist.*
  - *plus allen Übergängen* $(q_{i,j}, d_i, q'_{i+1})$ *für jedes Paar aus Zustand* $q_{i,j} \in Q$ *und Wert* $d_i \in D_i$, *wobei für* $q_{i,j}$ *kein ausgehender Übergang mit Wert* d_i *in* M *existiert,*

- *plus den Übergang* $(q_{3,0}, 1, q'_4)$,
- *plus den Übergang* $(q'_3, 0, q'_4)$.

Es entsteht somit die neue Übergangsmenge:

$$\begin{aligned}\delta' = \{ & (q_{0,0}, 1, q_{1,0}),\ (q_{0,0}, 2, q_{1,1}),\ (q_{0,0}, 3, q_{1,2}),\ (q_{1,0}, 1, q'_2),\ (q_{1,0}, 2, q_{2,0}),\ (q_{1,0}, 3, q_{2,1}),\\ & (q_{1,1}, 1, q_{2,0}),\ (q_{1,1}, 2, q'_2),\ (q_{1,0}, 3, q_{2,2}),\ (q_{1,2}, 1, q_{2,1}),\ (q_{1,2}, 2, q_{2,2}),\ (q_{1,2}, 3, q'_2),\\ & (q_{2,0}, 1, q'_3),\ (q_{2,0}, 2, q'_3),\ (q_{2,0}, 3, q_{3,0}),\ (q_{2,1}, 1, q'_3),\ (q_{2,1}, 2, q_{3,0}),\ (q_{2,1}, 3, q'_3),\\ & (q_{2,2}, 1, q_{3,0}),\ (q_{2,2}, 2, q'_3),\ (q_{2,2}, 3, q'_3),\ (q'_1, 1, q'_2),\ (q'_1, 2, q'_2),\ (q'_1, 3, q'_2),\ (q'_2, 1, q'_3),\\ & (q'_2, 2, q'_3),\ (q'_2, 3, q'_3),\ (q_{3,0}, 1, q_{4'}),\ (q'_3, 0, q_{4'})\}\end{aligned}$$

Das neue Constraint Regular$(\{x_1, x_2, x_3, x^{reif}\}, M')$ *mit* $M' = (Q', \Sigma', \delta', F', q_{0,0})$ *repräsentiert eine Bindung des ursprünglichen* Regular-*Constraints an die Variable* x^{reif}*. Die Variable* x^{reif} *bekommt genau dann den Wert 1 zugeordnet, wenn die Variablen* x_1, x_2, x_3 *Werte* d_1, d_2, d_3 *annehmen, deren Konkatenation* $d_1d_2d_3$ *von* M *akzeptiert wird. In Abbildung 5.5 ist eine grafische Repräsentation des Automaten* M' *abgebildet. Die neuen Zustände und Übergänge im Vergleich zu* M *sind dabei fett hervorgehoben. Es ist ersichtlich, dass der Zustand* q'_1 *nicht erreichbar ist. Bevor der Automat an das neue* Regular-*Constraint übergeben wird, sollten alle nicht erreichbaren Zustände und die damit verbundenen Übergänge entfernt werden, um in der wiederholt auftretenden Propagation nicht zusätzlichen Aufwand zu verursachen.*

Beweis 5.8. *Zunächst wird gezeigt, dass für jedes Wort* $w = d_1...d_n$*, welches als Eingabe in* M *einen finalen Zustand erreicht, auch das Wort* $w \circ 1$ *als Eingabe für den neuen levelbasierten DFA* M' *einen finalen Zustand erreicht. Sollte das Wort* w *als Eingabe für* M *zu keinem finalen Zustand gelangen, führt das Wort* $w \circ 0$ *in* M' *in einen finalen Zustand.*

Anschließend wird gezeigt, dass für jedes Wort $w = d_1...d_n1$*, welches in einem finalen Zustand in* M' *mündet, auch das Wort* $w' = d_1...d_n$ *in* M *einen finalen Zustand erreicht und dass für jedes Wort* $w = d_1...d_n0$*, welches in* M' *in einen finalen Zustand führt, das Wort* $w' = d_1...d_n$ *in* M *zu keinen finalen Zustand führt.*

Teil 1 $(c \longrightarrow sub(c))$***:*** *Da alle Zustände* Q *und alle Übergänge* δ *von* M *auch in* M' *enthalten sind, sind alle Pfade von* $q_{0,0}$ *zu* $q_{n,0}$ *auch in* M' *enthalten. In* M *waren diese Pfade Lösungspfade, also Pfade vom Startzustand* q_0 *zum einzigen finalen Zustand* $q_{n,0} \in F$*. In* M' *wird aus diesen Pfaden mit dem Übergang von* $q_{n,0}$ *zu* q'_{n+1} *mit Wert 1 ein Lösungspfad. Jeder ursprüngliche Lösungspfad fordert somit, dass der Variable zum letzten Level, also* x^{reif}*, der Wert 1 zugeordnet wird.*

Da alle neuen Übergänge in M' *als Ziel einen neuen Zustand* $q'_i \in Q'$ *mit* $i \in \{1, ..., n+1\}$ *haben, existieren in* M' *nur Pfade von* $q_{0,0}$ *zu* q'_n*, die mindestens einen neuen Übergang enthalten und somit den ursprünglichen DFA* M *nicht erfüllen. Da der Zustand* q'_n *nur einen Übergang mit Wert 0 für die Variable* x^{reif} *besitzt um zum finalen Zustand* q'_{n+1} *zu gelangen, ordnen alle Pfade, die keine Lösung in* M *repräsentieren, der Variablen* x^{reif} *den Wert 0 zu.* □

Teil 2 $(sub(c) \rightarrow c)$***:*** *Wenn der Variable* x^{reif} *der Wert 1 zugeordnet wird, dann muss der Übergang* $e = (q_{n,0}, 1, q'_{n+1})$ *in* M' *Teil der Lösungspfade von* $q_{0,0}$ *zu* q'_{n+1} *sein. Da die Pfade von* $q_{0,0}$ *zu*

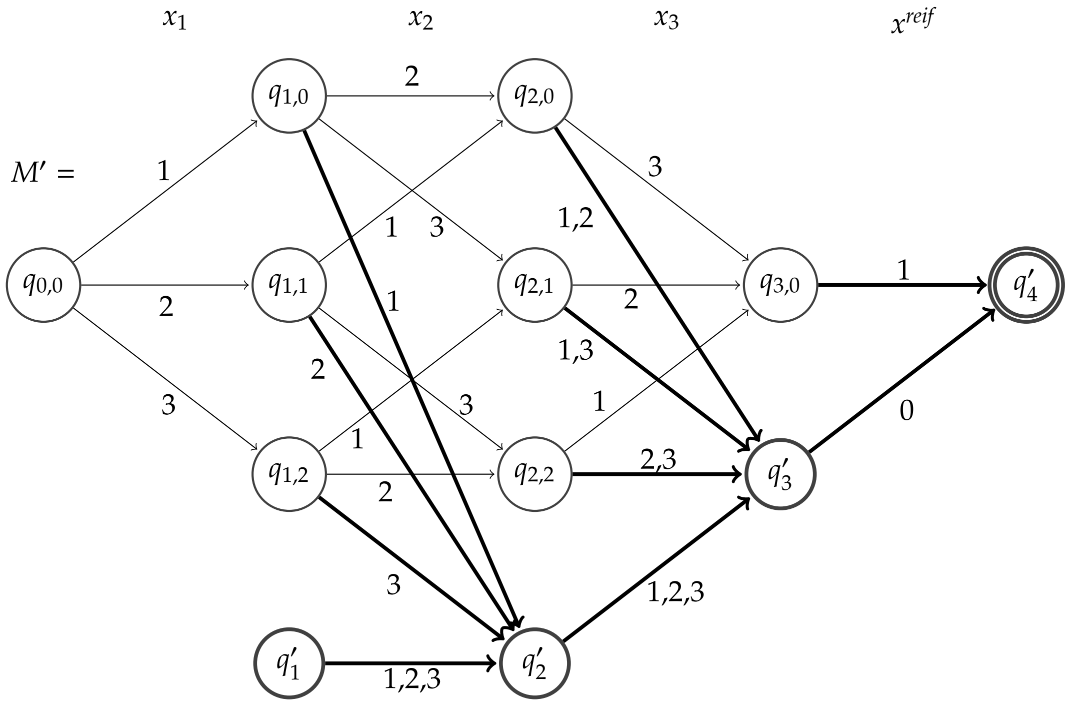


Abbildung 5.5: Der an die Variable x^{reif} gebundene levelbasierte DFA M'.

$q_{n,0}$ in M' gerade alle Lösungspfade von M sind und die Zustände entlang dieser Pfade keine neuen Übergänge als Eingang haben, folgt daraus, dass der DFA M erfüllt sein muss, wenn x^{reif} den Wert 1 annimmt.

Wird der Variablen x^{reif} der Wert 0 zugeordnet, so muss der Übergang $e = (q'_n, 0, q'_{n+1})$ in M' Teil der Lösungspfade von $q_{0,0}$ zu q'_{n+1} sein. Da alle Pfade von $q_{0,0}$ zu q'_n mindestens einen Übergang enthalten, der nicht in M vorhanden ist, können diese Pfade nicht Teil eines Lösungspfades in M gewesen sein. Wenn der Variablen x^{reif} der Wert 0 zugeordnet wird, kann folglich der DFA M nicht erfüllt sein. □

Das *And*-Constraint

Das Constraint *And*($\{x_1, ..., x_n\}$) fordert für eine Menge von booleschen Variablen, dass diese alle den Wert *Wahr* bzw. 1 annehmen. Eine Erweiterung dieses Constraints *And*($\{c_1, ..., c_n\}$) auf eine Meta-Ebene fordert, dass alle Constraints $c_1, ..., c_n$ erfüllt sein müssen. Wird das *And*-Constraint dabei selbst nicht als Eingabe für ein weiteres Constraint verwendet, so genügt es die Constraints c_1 bis c_n einzeln durch *Regular*-Constraints zu substituieren und dem CSP hinzuzufügen.

Wird das zu substituierende *And*-Constraint hingegen als Eingabe für andere Constraints verwendet, so müssen zunächst die Constraints $c_1, ..., c_n$ regularisiert und an boolesche

Variablen $x_1^{reif}, ..., x_n^{reif}$ gebunden werden, bevor das eigentliche *And*-Constraints durch das Constraint $And(x_1^{reif}, ..., x_n^{reif})$ ersetzt wird. Dieses letztere Constraint muss abschließend ebenfalls regularisiert werden. Ein *And*-Constraint mit booleschen Variablen $x_1, ..., x_n$ ist äquivalent zu einem *Count*-Constraint, welches für alle Variablen $x_1, ..., x_n$ fordert, dass diese den Wert 1 (Wahr) annehmen (siehe Gleichung 5.19). Somit kann die Regularisierung eines *And*-Constraints über booleschen Variablen mit der in Abschnitt 5.2.1 vorgestellten Regularisierung des *Count*-Constraints durchgeführt werden. Eine Substituierung der Meta-Version des *And*-Constraints, mit Constraints $c_1, ..., c_n$ als Eingaben, kann durch Regularisierung und Bindung der Eingabe-Constraints an boolesche Variablen $x_1^{reif}, ..., x_n^{reif}$ und anschließender Regularisierung des $And(x_1^{reif}, ..., x_n^{reif})$-Constraints erfolgen.

$$And(\{x_1, ..., x_n\}) \leftrightarrow Count(\{x_1, ..., x_n\}, n, 1) \tag{5.19}$$

Das *Or*-Constraint

Das Constraint $Or(x_1, ..., x_n)$ fordert für eine Menge von booleschen Variablen, dass mindestens einer dieser der Wert 1 zugeordnet wird. Das Meta-Constraint dazu erhält als Eingabe eine Menge von Constraints $c_1, ..., c_n$ und fordert, dass mindestens eines dieser Constraints erfüllt ist. Im Gegensatz zum *And*-Constraint ist somit ein direktes Hinzufügen der Constraints $c_1, ..., c_n$ zum CSP nicht möglich.

Das weitere Vorgehen ist ähnlich dem beim *And*-Constraint. Die Constraints $c_1, ..., c_n$ werden in *Regular*-Constraints transformiert und an boolesche Variablen $x_1^{reif}, ..., x_n^{reif}$ gebunden. Statt des ursprünglichen Constraints werden die gebundenen *Regular*-Constraints und die regularisierte Form des Constraints $Or(\{x_1^{reif}, ..., x_n^{reif}\})$ zum CSP hinzugefügt. Das *Or*-Constraint über booleschen Variablen kann ebenfalls durch ein *Count*-Constraint repräsentiert werden (siehe Gleichung 5.20), dessen Regularisierung bereits in Abschnitt 5.2.1 erläutert wurde.

$$Or(\{x_1, ..., x_n\}) \leftrightarrow Count(\{x_1, ..., x_n\}, occ, 1) \text{ mit } D_{occ} = \{1, ..., n\} \tag{5.20}$$

Das *Not*-Constraint

Das Constraint $Not(x)$ fordert für eine boolesche Variable x, dass dieser der Wert 0 zugeordnet wird. Das Meta-Constraint dazu erhält als Eingabe ein Constraint c und fordert, dass dieses Constraint nicht erfüllt ist. Ein direktes Hinzufügen des Constraints c zum CSP ist somit nicht möglich.

Für eine Regularisierung eines Constraints $Not(c)$ wird zunächst das Constraint c regularisiert. Der entstandene minimale levelbasierte DFA M kann dann, ähnlich dem Vorgehen bei der Bindung eines levelbasierten DFAs M an eine Variable x^{reif}, transformiert werden um einen $Not(c)$ repräsentierenden levelbasierten DFA M^{not} zu erzeugen. Der DFA $M^{not} = (Q', \Sigma', \delta', q_{0,0}, F')$ ergibt sich dabei wie folgt aus $M = (Q, \Sigma, \delta, q_{0,0}, F = \{q_{n,0}\})$:

- Das neue Alphabet Σ' enthält alle Zeichen aus Σ.
- Die neue Zustandsmenge Q' enthält n neue Zustände $q'_1, ..., q'_n$ und alle Zustände aus Q ohne $q_{n,0}$.
- Der Startzustand $q_{0,0}$ von M ist auch in M' der Startzustand.
- Der Automat M^{not} enthält gleich viele Level wie M. Jeder alte Zustand $q \in Q$ hat in Q' das gleiche Level wie in Q. Jeder neue Zustand $q'_1, ..., q'_n$ für $i \in \{1, ..., n\}$ wird dem Level i und somit der Zustandsmenge Q'_i zugeordnet.
- Die Menge der finalen Zustände $F = \{q_{n,0}\}$ wird durch $F' = \{q'_n\}$ ersetzt.
- Die neue Menge der Übergänge δ' beinhaltet alle Übergänge aus δ, die nicht $q_{n,0}$ als Zielzustand haben und wird um die folgenden Übergänge erweitert:
  - alle Übergänge $(q'_{i-1}, d_i) \rightarrow q'_i, \forall i \in \{2, ..., n\}, d_i \in D_i$
  - alle Übergänge, $(q_{i,j}, d_i) \rightarrow q'_{i+1}$ mit $q_{i,j} \in Q, d_i \in D_i$, für die nicht bereits ein Übergang von $q_{i,j}$ mit Wert d_i in δ existiert.

Durch Minimierung des levelbasierten DFAs M^{not} können gegebenenfalls noch Zustände und Übergänge ausgeschlossen und der DFA somit verkleinert werden. Auf den Beweis, dass der levelbasierte DFA M^{not} der komplementäre Automat zu M ist, wird an dieser Stelle verzichtet, da dieser bereits implizit in Beweis 5.8 für die Bindung eines levelbasierten DFAs an eine boolesche Variable enthalten ist.

Zur Veranschaulichung ist in Abbildung 5.6 der komplementäre levelbasierte DFA M_2^{not} zu dem in Beispiel 4.4 gegebenen levelbasierten DFA M_2 dargestellt. Bei den hervorgehobenen Übergängen und Zuständen handelt es sich um die neu hinzugefügten. Werden die beiden levelbasierten DFAs M' und M_2^{not} aus Abbildung 5.5 und Abbildung 5.6 verglichen, so fällt auf, dass die Zustandsmenge und die Übgergangsmenge von M'_2 jeweils Teilmengen der entsprechenden Mengen von M' sind.

Mit Hilfe der aufgeführten Umwandlung eines levelbasierten DFAs kann jedes Constraint $Not(c)$ in ein Constraint $Regular(X, M^{not})$ mit $X = scope(c)$ überführt werden. Dafür wird zunächst das Constraint c regularisiert und zu dem entstandenen levelbasierten DFA M der komplementäre levelbasierte DFA M^{not} erzeugt.

Weitere Constraints

Alle weiteren logischen Meta-Constraints können durch Kombination der drei bereits vorgestellten logischen Meta-Constraints (*And*, *Or* und *Not*) repräsentiert werden. Eine Regularisierung dieser kann dann durch Überführung in *And*-, *Or*- und *Not*-Constraints und anschließender Regularisierung erfolgen.

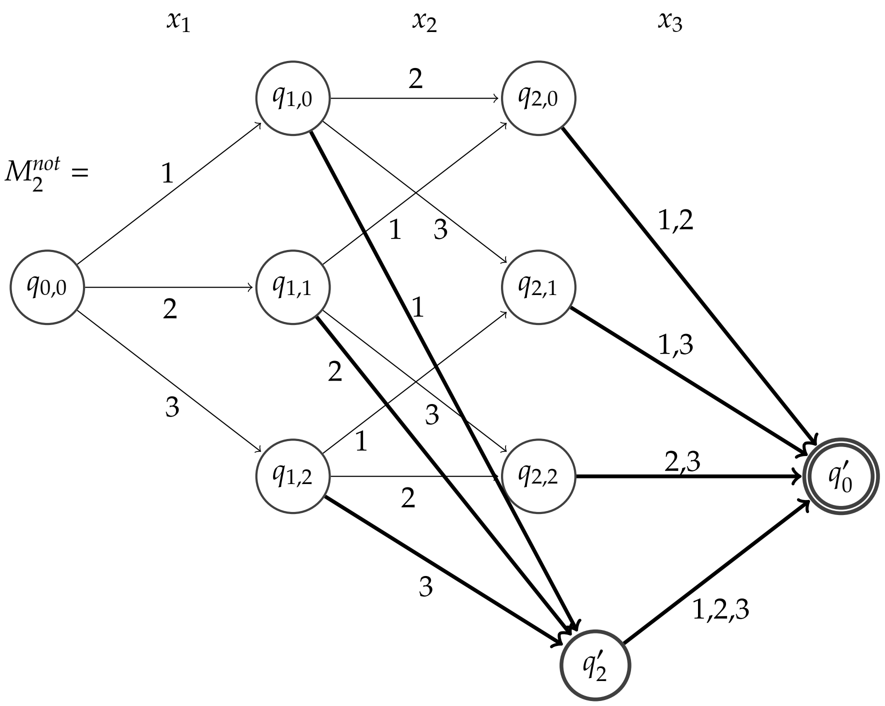


Abbildung 5.6: Der komplementäre levelbasierte DFA zu dem levelbasierten DFA M_2 aus Beispiel 4.4.

Weitere Umsetzungsmöglichkeiten von logischen Meta-Constraints

Neben den bisher gezeigten Regularisierungen von logischen Meta-Constraints bieten sich auch alternative Regularisierungen an. Spannt das zu substituierende logische Meta-Constraint zum Beispiel nur einen kleinen Suchraum auf, so kann mit dem in Abschnitt 5.1.1 vorgestellten Algorithmus 2 eine Regularisierung des logischen Meta-Constraints mit nur einem *Regular*-Constraint erfolgen. Eine Regularisierung als einzelnes Constraint führt die in Abschnitt 4.2 aufgeführten Vorteile mit sich und ist in der Regel bei kleinem Suchraum zu bevorzugen.

Gilt auf der Variablenmenge, die in allen beteiligten Constraints eines Meta-Constraints vorkommen, die gleiche Ordnung, so können die *And*- und *Or*-Constraints auch durch Schnittmengenbildung bzw. Vereinigung der levelbasierten DFAs der regularisierten ursprünglichen Constraints erzeugt werden. Die Änderung der Variablenreihenfolge innerhalb eines levelbasierten DFAs ist zwar möglich und hat auch einen signifikanten Einfluss auf die Größe des levelbasierten DFAs, allerdings ist die Problemstellung des Findens einer optimierten Variablenordnung NP-schwer [32]. Aufgrund der Komplexität der Variablenneuordnung wird im weiteren Verlauf der Arbeit darauf verzichtet, Variablen für die Regularisierung von logischen Meta-Constraints neu zu ordnen. Des Weiteren wird für *And*- und *Or*-Constraints, wenn möglich, die levelbasierte Schnittmengenbildung bzw. Vereinigung verwendet, sollte dies nicht möglich sein, so wird jeweils die Reifikationsvariante verwendet.

5.3.2 Die boolesche Skalarisierung

Die boolesche Skalarisierung eines logischen Meta-Constraints c^{Meta} mit Eingabe-Constraints $c_1, ..., c_n$ erfolgt in mehreren Schritten.

1. Umwandeln der Eingabe-Constraints $c_1, ..., c_n$ in Mengen boolescher *Scalar*-Constraints $C_1^{BS}, ..., C_n^{BS}$ mittels den in den Abschnitten 5.1.2 und 5.2 vorgestellten Verfahren.
2. Binden jedes booleschen *Scalar*-Constraints $c_1, ..., c_k$ jeder booleschen *Scalar*-Constraint-Menge C_i^{BS} an eine Variable $x_j^{success}$, $j \in \{1, ..., k\}$.
3. Binden der k $x_j^{success}$-Variablen an eine neue Variable x_i^{reif+} für jede Menge $C_1^{BS}, ..., C_n^{BS}$.
4. Verwenden eines zu c^{Meta} äquivalenten booleschen Constraints mit den booleschen Variablen $x_1^{reif+}, ..., x_n^{reif+}$ als Eingaben.

Da das Umwandeln von Constraints in Mengen boolescher *Scalar*-Constraints (1.) bereits in den Abschnitten 5.1.2 und 5.2 erläutert wurde, wird an dieser Stelle direkt mit den Erklärungen zur Bindung einer Variablen $x_i^{success}$ an ein einzelnes boolesches *Scalar*-Constraint c_i (2.) fortgefahren. Für jede solche Variable $x_i^{success}$ wird eine weitere Variable x_i^{fail} eingeführt, so dass gilt:

$$x_i^{success} = 1 \leftrightarrow x_i^{fail} = 0 \leftrightarrow \text{das boolesche } \mathit{Scalar}\text{-Constraint } c_i \text{ ist erfüllt}$$

bzw.

$$x_i^{success} = 0 \leftrightarrow x_i^{fail} = 1 \leftrightarrow \text{das boolesche } \mathit{Scalar}\text{-Constraint } c_i \text{ ist nicht erfüllt.}$$

Zunächst werden an dieser Stelle boolesche *Scalar*-Constraints mit „<"-Relation betrachtet. Durch die Gleichungen 5.21, 5.22 und 5.23 lässt sich jedes boolesche *Scalar*-Constraint der Form $c = Scalar(\{x_1^B, ..., x_n^B\}, (c_1, ..., c_n)^T, <, r)$, wobei $x_1, ..., x_n$ boolesche Variablen und $c_1, ..., c_n, r$ ganze Zahlen sind, an eine boolesche Variable $x^{success}$ binden.

$$(\sum_{i=1}^{n} c_i * x_i^B) + k * x^{success} \geq r \text{ mit } k \geq r - max(\sum_{i=1}^{n} c_i * x_i^B) \tag{5.21}$$

$$(\sum_{i=1}^{n} c_i * x_i^B) - k * x^{fail} < r \text{ mit } k > max(\sum_{i=1}^{n} c_i * x_i^B) - r \tag{5.22}$$

$$x^{success} + x^{fail} = 1 \tag{5.23}$$

Ist das ursprüngliche *Scalar*-Constraint c erfüllt, dann ist die Summe $\sum_{i=1}^{n} c_i * x_i^B$ kleiner als r. Damit die Gleichung 5.21 erfüllt wird, muss die boolesche Variable $x^{success}$ (bei einem ausreichend großen Wert für k) den Wert 1 annehmen. Ähnlich verhält es sich bei Gleichung 5.22. Ist das ursprüngliche *Scalar*-Constraint c unerfüllt, dann ist die Summe $\sum_{i=1}^{n} c_i * x_i^B$ größer oder gleich r. Damit muss x^{fail} bei ausreichend großem Wert für k den Wert 1 annehmen. Würden $x^{success}$ und x^{fail} beide den Wert 1 annehmen, dann wären die beiden Gleichungen 5.21 und 5.22 unabhängig von der Summe $\sum_{i=1}^{n} c_i * x_i^B$ erfüllt. Um dies zu vermeiden, sichert Gleichung 5.23, dass nur einer der beiden Variablen $x^{success}$ und x^{fail} der Wert 1 zugeordnet wird. Die beiden Werte für k müssen dabei jeweils so groß gewählt werden, dass die Gleichungen 5.21 und 5.22 für jede mögliche Belegung der Werte x_i^B erfüllt sind, wenn $x^{success}$ bzw. x^{fail} mit Wert 1 belegt sind.

Beweis 5.9. *Der Beweis erfolgt in vier Schritten.*

1. *Es wird zunächst gezeigt, dass aufgrund der Gleichungen 5.21, 5.22 und 5.23 die Variable $x^{success}$ den Wert 1 erhält, wenn das ursprüngliche Constraint c erfüllt ist.*
2. *Ist das ursprüngliche Constraint c nicht erfüllt, so wird der Variablen $x^{success}$ der Wert 0 zugeordnet.*
3. *Ist der Variable $x^{success}$ der Wert 1 zugeordnet, so muss das ursprüngliche Constraint c erfüllt sein.*

4. *Ist der Variable $x^{success}$ der Wert 0 zugeordnet, so kann das ursprüngliche Constraint c nicht erfüllt sein.*

Fall 1: *Wenn das ursprüngliche Constraint $c = \mathrm{Scalar}(\{x_1^B, ..., x_n^B\}, (c_1, ..., c_n)^T, <, r)$ erfüllt ist, dann muss die Summe $\sum_{i=1}^{n} c_i * x_i^B$ einen Wert $r^<$ kleiner als r angenommen haben ($r^< < r$). Aus Gleichung 5.21 folgt dann durch Umstellen: $k * x^{success} \geq r - r^<$. Da $x^{success}$ nur den Wert 0 oder 1 annehmen kann und k per Definition größer als $r - r^>$ ist, muss die Variable $x^{success}$ den Wert 1 erhalten.* □

Fall 2: *Wenn das ursprüngliche Constraint c nicht erfüllt ist, dann muss die Summe $\sum_{i=1}^{n} c_i * x_i^B$ einen Wert $r^\geq$ größer oder gleich r angenommen haben ($r^\geq > r$). Aus Gleichung 5.22 folgt dann durch Umstellen: $k * x^{fail} > r^\geq - r$. Da x^{fail} nur den Wert 0 oder 1 annehmen kann und k per Definition größer als $r^\geq - r$ ist, muss die Variable x^{fail} den Wert 1 zugeordnet bekommen. Aufgrund von Gleichung 5.23 wird der Variablen $x^{success}$ dann der Wert 0 zugeordnet.* □

Fall 3: *Wenn die Variable $x^{success}$ den Wert 1 erhält, so folgt aus Gleichung 5.23, dass der Variablen x^{fail} der Wert 0 zugewiesen wird. Aus Gleichung* 5.22 *folgt dadurch, dass die Summe $\sum_{i=1}^{n} c_i * x_i^B$ kleiner als r sein muss. Damit gilt, dass c erfüllt ist.* □

Fall 4: *Wenn der Variablen $x^{success}$ der Wert 0 zugeordnet wird, dann folgt aus Gleichung 5.21, dass die Summe $\sum_{i=1}^{n} c_i * x_i^B$ größer als oder gleich r sein muss. Somit folgt, dass c nicht erfüllt ist.* □

Für die weiteren Fälle, in denen c mit Relationssymbol $\mathfrak{R} \in \{\leq, >, \geq\}$ definiert ist, kann die jeweilige lineare Gleichung in eine äquivalente mit <-Relation umgeformt werden. Enthält $c = Scalar(\{x_1^B, ..., x_n^B\},\ (c_1, ..., c_n)^T,\ =,\ r)$ das „="-Relationssymbol, so kann dieses durch zwei *Scalar*-Constraints $c^\leq = Scalar(\{x_1^B, ..., x_n^B\}, (c_1, ..., c_n)^T, \leq, r)$ und $c^\geq = Scalar(\{x_1^B, ..., x_n^B\}, (c_1, ..., c_n)^T, \geq, r)$ ersetzt werden, welche dann, mithilfe des zuvor vorgestellten Vorgehens, an zwei boolesche Variablen gebunden werden können.

Für die ≠-Relation mit einer Ungleichung $(\sum_{i=1}^{n} c_i * x_i^B) \neq r$ erfolgt die Reifikation in zwei Schritten. Zunächst werden die beiden Ungleichungen $I = ((\sum_{i=1}^{n} c_i * x_i^B) < r)$ und $II = ((\sum_{i=1}^{n} c_i * x_i^B) > r)$ mittels der Gleichungen 5.21 bis 5.23 an zwei Variablen $x^{success_1}$ und $x^{success_2}$ gebunden. Anschließend wird eine Gleichung hinzugefügt, die dafür sorgt, dass die Summe aus den beiden Reifikationsvariablen $x^{success_1}$ und $x^{success_2}$ aus der Reifikation von I und II gleich $x^{success}$ ist ($x^{success_1} + x^{success_2} = x^{success}$). Das hat zur Folge, dass genau eine der beiden Gleichungen I oder II erfüllt sein muss, wenn $x^{success}$ der Wert 1 zugeordnet wird und anders herum. Zusätzlich folgt, dass die beiden Gleichungen I, II und somit auch die ursprüngliche Ungleichung $(\sum_{i=1}^{n} c_i * x_i^B) \neq r$ nicht erfüllt sein können, wenn $x^{success}$ den Wert 0 annimmt. Umgekehrt muss $x^{success}$ den Wert 0 erhalten, wenn die beiden Gleichungen I und II bzw. die ursprüngliche Ungleichung nicht erfüllt sind. Die Variable $x^{success}$ reifiziert somit das ≠-Constraint $(\sum_{i=1}^{n} c_i * x_i^B) \neq r$.

Wurden alle durch boolesche Skalarisierung erzeugten Constraints an boolesche Variablen $x_1^{success}, ..., x_k^{success}$ und $x_1^{fail}, ..., x_k^{fail}$ gebunden (2., Seite 149), so können diese an eine boolesche Variable x^{reif+} gebunden werden (3., Seite 149), die genau dann den Wert 1 annimmt, wenn allen Variablen $x_1^{success}, ..., x_k^{success}$ der Wert 1 zugeordnet wurde. Analog zum bisherigen Vorgehen wird dafür eine Variable x^{reif-} eingeführt, die genau dann den Wert 1 annimmt, wenn x^{reif+} den Wert 0 erhält, also mindestens ein boolesches *Scalar*-Constraint nicht erfüllt ist. Ist das der Fall, so ist auch das ursprünglich als Eingabe für das Meta-Constraint gegebene Constraint nicht erfüllt. Die Gleichungen 5.24, 5.25 und 5.26 realisieren dieses Vorgehen und können als boolesche *Scalar*-Constraints dargestellt werden.

$$x^{reif+} + (k-1) \geq \sum_{1}^{k} x_j^{success} \tag{5.24}$$

$$k * x^{reif-} \geq \sum_{1}^{k} x_j^{fail} \tag{5.25}$$

$$x^{reif+} + x^{reif-} = 1 \tag{5.26}$$

Die Variablen $x_j^{success}$ bzw. x_j^{fail} stellen dabei die Variablen $x^{success}$ bzw. x^{fail} aus den Gleichungen 5.21 bis 5.23 für das j-te Constraint (die j-te lineare Gleichung) dar. Aufgrund der Ähnlichkeit zu dem vorherigen Beweis wird auf diesen Beweis an dieser Stelle verzichtet.

Ein Constraint kann durch boolesche Skalarisierung in eine Menge von booleschen *Scalar*-Constraints überführt werden und die Erfüllbarkeit dieser Menge von Constraints kann, durch das zuvor beschriebene Verfahren, an eine boolesche Variable x^{reif+} gebunden werden. Dies dient als Grundlage für die im Folgenden vorgestellten booleschen Skalarisierungen von logischen Meta-Constraints (4., Seite 149).

Das *And*- und das *Or*-Constraint

Das $And(c_1, ..., c_n)$- und das $Or(c_1, ..., c_n)$-Constraint können wie folgt in boolesche *Scalar*-Constraints überführt werden. In den zuvor beschriebenen Schritten 1. bis 3. (Seite 149) werden die Eingabe-Constraints $c_1, ..., c_n$ des Meta-Constraints in boolesche *Scalar*-Constraints umgewandelt und an boolesche Variablen $x_1^{reif+}, ..., x_n^{reif+}$ gebunden. Das ursprüngliche Meta-Constraint wird im 4. Schritt (Seite 149) durch das boolesche Constraint $And(x_1^{reif+}, ..., x_n^{reif+})$- bzw. $Or(x_1^{reif+}, ..., x_n^{reif+})$ ersetzt und durch boolesche Skalarisierung des zu dem *And* bzw. *Or*-Constraint äquivalenten *Count*-Constraints (siehe Gleichungen 5.19 und 5.20) in boolesche *Scalar*-Constraints überführt.

Das *Not*-Constraint

Für das *Not*-Constraint wird das Eingabe-Constraint mittels den zuvor vorgestellten Verfahren in ein boolesches *Scalar*-Constraint transformiert und an eine Variable x^{reif+} gebunden (Schritte 1. bis 3., Seite 149). Das boolesche *Scalar*-Constraint $x^{reif+} = 0$ fungiert demzufolge als boolesche Skarlarisierung des ursprünglichen *Not*-Constraints.

Weitere logische Meta-Constraints

Genauso wie bei der Regularisierung können auch bei der booleschen Skalarisierung weitere logische Meta-Constraints durch Kombination von *And*-, *Or*- und *Not*-Constraints und anschließender Überführung in boolesche *Scalar*-Constraints substituiert werden.

Anmerkung 6. Umfassen die Meta-Constraints nur wenige Variablen mit kleinen Domänenwerten, so kann es vorteilhaft sein, die Meta-Constraints als Ganzes, mit den in Abschnitt 5.1 vorgestellten Verfahren, zu substituieren. Künftige Forschungen umfassen das Entwickeln eines geeigneten Klassifikators um zu entscheiden, wann welche Variante besser geeignet ist.

5.4 Schlussfolgerungen aus der Regularisierbarkeit und booleschen Skalarisierbarkeit von CSPs

In diesem Abschnitt werden Schlussfolgerungen bezüglich der in den vorherigen Abschnitten vorgestellten Substituierungen gezogen. Im Folgenden wird zunächst auf die Regularisierung und im Anschluss daran auf die boolesche Skalarisierung eingegangen und erläutert, welche neuen Möglichkeiten der Lösung von FD-CSPs sich dadurch ergeben.

5.4.1 Schlussfolgerungen bezüglich der Regularisierung von CSPs

Wie mit den vorgestellten Regularisierungen in den Abschnitten 5.1.1, 5.2 und 5.3 gezeigt wurde, kann jedes CSP in ein *Regular*-CSP überführt werden. Die speziellen Regularisierungen in den Abschnitten 5.2 und 5.3 sind dabei gegenüber der allgemeinen Regularisierung zu bevorzugen, da diese in der Regel schneller zu einem minimalen levelbasierten DFA führen. Ein Beispiel zur Veranschaulichung dieser schnelleren Substituierung wurde in Abschnitt 5.2.9 gegeben. In Kapitel 6 werden weitere Untersuchungen aufgeführt, die diese Aussage bekräftigen.

Beachtet werden muss an dieser Stelle, dass die direkte Regularisierung im Vergleich zu der allgemeinen Regularisierung zwar zu einer signifikanten Zeitverringerung bei der Substituierung führt, die Art der Regularisierung in der Regel allerdings keinen Einfluss auf die reine Lösungsdauer (ohne Zeit für das Substituieren) hat. Dies liegt daran, dass beim Minimieren der erzeugten levelbasierten DFAs (beider Vorgehen) im Allgemeinen der gleiche oder ein gleich großer levelbasierter DFA erzeugt wird. Die in der Regel schnellere und deutlich ressourcensparendere direkte Regularisierung ermöglicht es allerdings die Regularisierung deutlich häufiger anzuwenden, da durch sie viel seltener in Zeit- oder Ressourcenengpässe gelangt wird.

Es ergeben sich verschiedene Möglichkeiten zur weiteren Verarbeitung der erzeugten Regular-Constraints. Eine erste Möglichkeit ist es, das substituierte CSP mit dem ursprünglichen FD-Solver zu lösen. Aufgrund der Regularisierung kann sich, wie in Abschnitt 4.2.1 beschrieben, die Lösungsgeschwindigkeit bereits erhöhen. In vielen Fällen kann eine Vereinigung der levelbasierten DFAs verschiedener, sich überlagernder *Regular*-Constraints die Lösungsgeschwindigkeit noch weiter verbessern (siehe Abschnitt 4.2). Durch die Vereinigung wird das Konsistenzniveau des CSPs weiter erhöht, die Anzahl der Propagationsaufrufe im Durchschnitt verringert und verlangsamende Redundanz entfernt. Werden alle levelbasierten DFAs $M_1, ..., M_m$ aller Constraints $C = \{c_1, ..., c_m\}$ eines *Regular*-CSPs $P^{regular} = (X, D, C)$ zu einem einzigen minimalen levelbasierten DFA M' vereinigt, so sind alle Pfade vom Startzustand zum finalen Zustand in M' Lösungen des ursprünglichen CSPs. Somit ist ein Algorithmus, der alle levelbasierten DFAs eines *Regular*-CSPs vereint, gleichzeitig ein *Regular*-CSP-Solver.

Die Frage, die sich daraus ableitet, ist, ob es schneller ist alle DFAs zu vereinigen oder nur einzelne Constraints zu vereinigen und das transformierte CSP anschließend mit dem ursprünglichen Solver zu lösen. Die Frage kann nicht allgemein beantwortet werden, sondern muss von Fall zu Fall separat untersucht werden. In Anschnitt 6.2.5: Weitergehende Betrachtungen zur Regularisierung II wird ein erstes Vorgehen erläutert, das diese Entscheidung unterstützen kann.

Eine weitere Möglichkeit, Regularisierung weiterzuverwenden, ist die Umwandlung der levelbasierten DFAs in binäre Entscheidungsdiagramme (BDDs). Ein Algorithmus dafür ist beispielsweise in [156] angegeben. Einige SAT-Solver, wie zum Beispiel der in SICStus-Prolog und SWI-Prolog verwendete clp(B)-Solver, wandeln SAT-Probleme in BDDs um, um diese zu lösen [161]. Somit liefert die Regularisierung mit anschließender Überführung in BDDs eine weitere Möglichkeit, FD-CSPs in SAT-Probleme zu überführen.

Neben der Umwandlung von *Regular*-Constraints in BDDs wurde in Abschnitt 5.2.8 bereits die Umwandlung von *Regular*-Constraints in boolesche *Scalar*-Constraints vorgestellt und deren Korrektheit nachgewiesen. Somit stellt sich die Frage, ob das *Regular*-Constraint in weitere Constraints überführt werden kann, um damit weitere direkt oder indirekt konvertierte FD-vollständige CSPs ableiten zu können. Für jedes solche FD-vollständige CSP können anschließend spezialisierte Solver entwickelt werden, die die entsprechenden CSPs schnellstmöglich lösen.

5.4.2 Schlussfolgerungen bezüglich der booleschen Skalarisierung von CSPs

Ausgehend davon, dass jedes FD-CSP in ein *Regular*-CSP und jedes *Regular*-Constraint in boolesche *Scalar*-Constraints überführt werden kann, bietet die boolesche Skalarisierung von *Regular*-Constraints eine weitere Möglichkeit der Umwandlung von FD-CSPs über *Regular*-CSPs in boolesche *Scalar*-CSPs. Diese Form der booleschen Skalarisierung wird nachfolgend als *Regular*-basierte boolesche Skalarisierung bezeichnet. Somit ergeben sich bisher vier verschiedene Möglichkeiten ein Constraint in boolesche *Scalar*-Constraints zu überführen:

1. Die konfliktbasierte boolesche Skalarisierung (siehe Abschnitt 5.1.2) ist für alle Constraints anwendbar. Bei der Substituierung werden keine zusätzlichen Variablen aber sehr viele Constraints erzeugt.

2. Die Support-basierte boolesche Skalarisierung (siehe Abschnitt 5.1.2) ist ebenfalls für alle Constraints anwendbar. Bei der Substituierung werden sowohl zusätzliche Variablen als auch zusätzliche Constraints erzeugt.

3. Die boolesche Skalarisierung spezieller Constraints ist auf der einen Seite nur für die Constraints anwendbar, für die solch ein Verfahren entwickelt wurde (siehe Abschnitt 5.2). Auf der anderen Seite können die Constraints dadurch sehr schnell und unter Verwendung sehr weniger Constraints und Variablen transformiert werden (siehe Abschnitt 5.2.9). Die verringerte Anzahl von Variablen und Constraints kann dabei, wie in Abschnitt 5.2.9 gezeigt, zu einer deutlichen Verringerung der reinen Lösungszeit führen.

4. Die *Regular*-basierte boolesche Skalarisierung (siehe Abschnitt 5.2.8) erlaubt immer dann eine sehr schnelle Substituierung mit wenigen Variablen und Constraints, wenn die Regularisierung sehr schnell durchgeführt werden kann und dabei ein levelbasierter DFA mit wenigen Übergängen entsteht.

Aus der Existenz verschiedener boolescher Skalarisierungen ergibt sich die Frage, wann welches Verfahren verwendet werden sollte. Die boolesche Skalarisierung spezieller Constraints (3) ist die der Konflikt- und Support-basierten (1 und 2) vorzuziehen, da sie in der Regel erstens schneller durchgeführt werden kann und zweitens zu CSPs mit weniger Constraints und/oder Variablen führt. Solche CSPs können aufgrund ihrer größeren Kompaktheit oft schneller gelöst werden.

Der *Regular*-basierte Ansatz (4) scheint auf Grund der sehr schnellen Regularisierung und im Resultat sehr kleinen levelbasierten DFAs konkurrenzfähig zur booleschen Skalarisierung spezieller Constraints (3) zu sein. Insbesondere wenn das *Regular*-Constraint bereits mehrere ursprüngliche Constraints vereinigt, kann eine boolesche Skalarisierung des vereinigten *Regular*-Constraints zu einem schneller zu lösenden booleschen *Scalar*-CSP führen, als dies über die direkte boolesche Skalarisierung der einzelnen Constraints möglich ist.

Unabhängig von ihrer Erzeugung (1. bis 4.) können alle booleschen *Scalar*-CSPs mittels eines linearen pseudo-booleschen Solver gelöst werden. Ziel ist es einen möglichst spezialisierten Solver verwenden zu können, in der Annahme, dass dieser das Problem besonders schnell lösen kann. Unter diesem Gesichtspunkt erscheint es sinnvoll die resultierenden booleschen *Scalar*-CSPs detailliert zu betrachten, um herauszufinden, ob diese mit noch spezielleren Solvern gelöst werden können. Aus dieser Betrachtungsweise ergeben sich *Regular*-basierte boolesche *Scalar*-CSPs und boolesche *Count*-CSPs als Unterkategorien der booleschen *Scalar*-CSPs, welche nachfolgend erläutert werden.

Regular-basierte boolesche *Scalar*-CSPs

Die *Regular*-basierte boolesche Skalarisierung von CSPs zeigt eine speziellere Form der booleschen *Scalar*-CPSs auf, als sie bisher in dieser Arbeit definiert wurde. Wir bezeichnen diese Form der CSPs nachfolgend *Regular*-basierte boolesche *Scalar*-CSPs. In *Regular*-basierten booleschen *Scalar*-CSPs befinden sich nur noch boolesche Variablen sowie Constraints zum Erfüllen der Voraussetzung 5.2 und der Constraints C^{Arc} und C^{Node}. Da bei einem booleschen *Scalar*-CSP alle ursprünglichen Variablen entfernt wurden, wird kein Constraint mehr benötigt, um die Voraussetzung 5.1 zu erfüllen. Zur Absicherung der verbliebenen drei Constraint-Arten (Voraussetzung 5.2, C^{Arc} und C^{Node}) werden nur Constraints der folgenden Form benötigt:

$$Scalar(X, (-1, ..., -1, 1, ..., 1)^T, =, r) \text{ mit } r \in \{-1, 0, 1\} \tag{5.27}$$

Es werden also nur noch Gleichheit-Constraints benötigt, die Skalare sind nur noch Werte aus $\{-1, 1\}$ und die Ergebnisse r nur noch Werte aus $\{-1, 0, 1\}$. Aufgrund dieser signifikanten Eigenschaften wird vermutet, dass ein speziellerer Solver als ein linearer pseudo-boolescher Solver zur Lösung von *Regular*-basierten booleschen *Scalar*-CSPs entwickelt werden kann.

Count-basierte boolesche CSPs

Die Constraints aus Voraussetzung 5.2, C^{Arc} und C^{Node} lassen sich auch durch *Count*-Constraints über booleschen Variablen darstellen (siehe Gleichungen 5.28 bis 5.30). Daher stellen boolesche *Count*-CSPs eine weitere Form der indirekt konvertierten FD-vollständigen CSPs dar. Die Funktion ++ vereinigt dabei zwei geordnete Mengen A und B derart, dass in der zusammengefügten geordneten Menge alle Elemente der Menge A vor den Elementen aus Menge B vorkommen.

$$Scalar(\{x^B_{i,1}, ..., x^B_{i,|D_i|}\}, (1, ..., 1)^T, =, 1) \leftrightarrow count(\{x^B_{i,1}, ..., x^B_{i,|D_i|}\}, 1, 1) \tag{5.28}$$

$$Scalar(\{x^B_{i,j}, X^\delta_{i,v}\}, (-1, 1, ..., 1)^T, =, 0) \leftrightarrow count(X^\delta_{i,v}, x^B_{i,j}, 1) \tag{5.29}$$

$$
\begin{aligned}
Scalar(X_{i,j}^{in} +\!\!+ X_{i,j}^{out}, (-1, ..., -1, 1, ..., 1)^T, =, 0) &\leftrightarrow Count(X_{i,j}^{in}, occ, 1) \land Count(X_{i,j}^{out}, occ, 1) \\
&\quad \text{mit } occ \text{ ist eine neue boolesche Variable} \\
Scalar(X_{0,j}^{out}, (1, ..., 1)^T, =, 1) &\leftrightarrow Count(X_{0,j}^{out}, 1, 1) \\
Scalar(X_{n,j}^{in}, (-1, ..., -1)^T, =, -1) &\leftrightarrow Count(X_{n,j}^{in}, 1, 1)
\end{aligned}
\tag{5.30}
$$

Die Äquivalenzen in Gleichung 5.28 und 5.29 lassen sich direkt nachvollziehen. Zum ersten Fall in Gleichung 5.30 muss gesagt werden, dass dies keine direkte Äquivalenz darstellt, sondern dass sich die Äquivalenz erst daraus ergibt, dass als Folgerung von Voraussetzung 5.2 und den Constraints in C^{Arc} nur eine Variable $x_{i,1}^{\delta}, ..., x_{i,|\delta_i|}^{\delta}$ pro Level i den Wert 1 annehmen kann. Da sowohl alle $X_{i,j}^{in}$ als auch alle $X_{i,j}^{out}$ Teilmengen von $x_{i,1}^{\delta}, ..., x_{i,|\delta_i|}^{\delta}$ sind, ergibt es sich, dass nur maximal einer Variable in $X_{i,j}^{in}$ bzw. $X_{i,j}^{out}$ der Wert 1 zugeordnet wird.

Auch für das boolesche *Count*-CSP können zusätzliche Einschränkungen an die verwendeten *Count*-Constraints angegeben werden (wie zum Beispiel, dass die Häufigkeitsvariable nie einen Wert größer 1 annimmt), die eventuell den Einsatz eines noch spezifischeren Solvers erlauben.

Zusammenfassung

In Abschnitt 5.1.1 konnte gezeigt werden, dass sich jedes FD-CSP in ein *Regular*-CSP und auch in ein konfliktbasiertes oder Support-basiertes boolesches *Scalar*-CSP überführen lässt. In Abschnitt 5.2 wurden die allgemeinen Substituierungsverfahren um Verfahren für die Substituierung spezieller Constraints erweitert. Diese erhöhen die Gesamtgeschwindigkeit der Substituierung signifikant und führen im Falle der booleschen Skalarisierung zu einer kompakteren Modellierung als dies mit der allgemeinen booleschen Skalarisierung der Fall ist. In Abschnitt 5.3 konnten Substituierungen für eine weitere Klasse von Constraints, den logischen Meta-Constraints, angegeben werden. Abschließend wurden Schlussfolgerungen aus den verschiedenen Substituierungen gezogen und es wurden weitere FD-vollständige CSPs abgeleitet. In Abbildung 5.7 sind die verschiedenen Möglichkeiten zur Lösung eines beliebigen FD-CSPs, welche sich aus der Regularisierbarkeit und booleschen Skalarisierbarkeit von FD-CSPs ergeben, noch einmal dargestellt.

Nachdem die verschiedenen Substituierungsideen erläutert und Transformationen dafür erstellt wurden, sollen in der Zukunft Heuristiken entwickelt werden, die vorhersagen können, wann welche Substituierungen am erfolgversprechendsten sind.

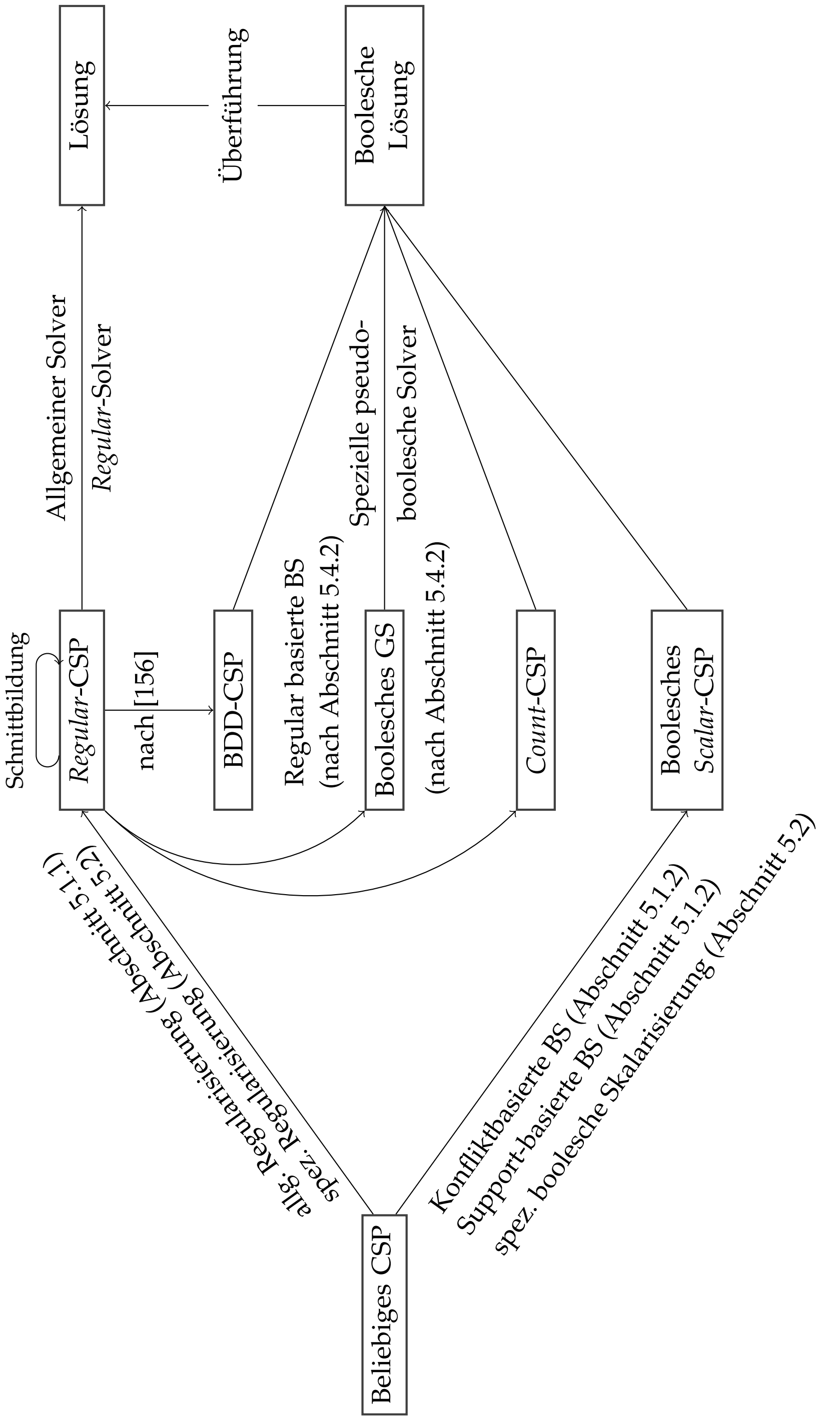


Abbildung 5.7: Der dreistufige Ablauf beim Lösen eines Constraint-Problems unter Verwendung von Remodellierungsansätzen.

6 Evaluierung der Substituierungsansätze

Nachdem in Kapitel 5 die verschiedenen Substituierungen vorgestellt wurden (allgemeine und spezielle Regularisierung sowie spezielle, konfliktbasierte, Suport-basierte, *Count*-basierte und *Regular*-basierte boolesche Skalarisierung), wird nachfolgend deren Einfluss auf die Lösungsdauer und den Substitutionsaufwand eines CSPs evaluiert. Die Evaluation teilt sich dabei in zwei Teile auf. Zunächst werden die erarbeiteten Substituierungen bei einzelnen globalen Constraints eingesetzt und miteinander verglichen (Abschnitt 6.1), anschließend werden die Substituierungen bei realen Problemstellungen angewendet und analysiert, welche Auswirkungen sie auf die benötigte Zeit für das Finden einer ersten bzw. einer optimalen Lösung haben (Abschnitt 6.2).

Für die Evaluation wurde die Namensgebung aus den Beispielen 4.1 und 5.2.9 erweitert, so dass sie final die folgende Form annimmt:

$$Substituierungsart(\{Gruppe_1^{V_1}, Gruppe_2^{V_2}, ...\})$$

mit

- $Substituierungsart \in \{T = Tabularisierung,\ R = Regularisierung,\ B = boolesche\ Skalarisierung\}$
- $Gruppe_1, Gruppe_2, ...$ = Gruppen von Constraints, die jeweils gemeinsam substituiert werden
- $V_1, V_2, ...$ = konkrete Angaben darüber, wie die zugehörige Gruppe transformiert wird:
  - K = nur bei der booleschen Skalarisierung anwendbar, es wurde die konfliktbasierte boolesche Skalarisierung verwendet (siehe Abschnitt 5.1.2).
  - S = nur bei der booleschen Skalarisierung anwendbar, es wurde die Support-basierte boolesche Skalarisierung verwendet (siehe Abschnitt 5.1.2).
  - D = nur bei der Regularisierung und der booleschen Skalarisierung anwendbar, es wurden direkte Transformationen bei der Substituierung verwendet (siehe Abschnitte 5.2 und 5.3).
  - DFA = nur bei der booleschen Skalarisierung anwendbar, es wurde die *Regular*-basierte boolesche Skalarisierung verwendet (siehe Abschnitt 5.4.2).

  - *Count* = nur bei der booleschen Skalarisierung anwendbar, es wurde ein boolesches *Count*-CSP erzeugt (siehe Abschnitt 5.4.2).

  Bei der bisherigen Verwendung der Namensgebung in den Beispielen 4.1 und 5.2.9 wurde bei der Tabularisierung und Regularisierung bisher der V-Wert 1 und 2 verwendet. Diese Werte gaben an, dass die in Abschnitt 4.1.1 angegebene erste oder zweite Variante verwendet wurde. Da sich diese im Zeitaufwand allerdings nur geringfügig unterschieden, wird in der Folge immer von Variante 1 ausgegangen und auf eine entsprechende Notation verzichtet.

Zum Lösen aller im Folgenden erzeugten CSPs wurde jeweils der CHOCO-Solver mit der in Abschnitt 1.4 angegebenen Suchstrategie, Software und Hardware verwendet.

6.1 Eine Evaluation ausgewählter direkter Substituierungen

In diesem Abschnitt werden die verschiedenen, zuvor vorgestellten Substituierungen (allgemeine und spezielle Regularisierung sowie spezielle, konfliktbasierte, Suport-basierte, *Regular*-basierte boolesche Skalarisierung und das Erzeugen und Lösen von booleschen *Count*-Constraints, siehe Kapitel 5) auf einzelne, generierte, globale Constraints angewendet, welche jeweils in einem eigenen CSP untersucht werden. Dabei wird evaluiert, welchen zeitlichen Einfluss die verschiedenen Substituierungen auf das Finden einer ersten Lösung des jeweiligen CSPs haben. In diesem Zusammenhang werden sowohl die Gesamtlösungsdauern (inklusive der Zeit für die Transformationen) als auch die reinen Lösungszeiten (ohne Berücksichtigung der Transformationszeit) miteinander verglichen. Die Dauer für das Finden einer ersten Lösung eines CSPs ist maßgeblich von der verwendeten Suchstrategie abhängig. Die Verwendung verschiedener Variablen- und Werteauswahlheuristiken hat einen signifikanten Einfluss auf die Dauer für das Finden einer ersten Lösung eines CSPs. Analog hat auch die verwendete Modellierung einen signifikanten Einfluss auf die Lösungsdauer. Aufgrund der veränderten Struktur der Probleme und damit verbunden, dass jede Substituierung gegebenenfalls eine andere erste Lösung findet, kann der Zeitaufwand für das Finden einer ersten Lösung nicht direkt als Indikator dafür verwendet werden, ob die entsprechende Substituierung im generellen besonders effektiv ist oder nicht. Die Suche nach allen Lösungen anstelle einer ersten Lösung wäre an dieser Stelle aussagekräftiger, allerdings wurde sich aus den folgenden Gründen dennoch für die Suche nach einer ersten Lösung entschieden:

- In realen Problemstellungen wird in der Regel auch nur nach einer und nicht nach allen Lösungen gesucht.
- Das Suchen nach allen Lösungen wäre aufgrund von Ressourcenengpässen (Zeit) nur für sehr kleine globale Constraints realisierbar.

- Aufgrund dessen, dass die Testreihen verhältnismäßig viele CSPs umfassen (49 bis 400) und deren Durchschnittszeiten betrachtet werden, fallen einzelne Ausreißer, die sich durch besonders günstige oder ungünstige Variablenbelegungen ergeben, weniger ins Gewicht.

Nachfolgend wird auf die verschiedenen globalen Constraints des Abschnittes 5.2 eingegangen. Auf eine Evaluation der Substituierung logischer Meta-Constraints wird an dieser Stelle verzichtet, da diese in ihrer Auswertung zu sehr von den Constraints abhängen, die sie als Eingabe erhalten.

Zu jedem Constraint-Typ T (*Count*, *Global Cardinality* usw.) aus dem Abschnitt 5.2 werden einzelne Probleminstanzen $P = (\{x_1, ..., x_n\}, \{D_1, ..., D_n\}, \{c^T\})$ mit genau einem globalem Constraint c^T vom Typ T mit $scope(c^T) = \{x_1, ..., x_n\}$ generiert. Als Zeitlimit für das Substituieren und Finden einer ersten Lösung wurde eine Minute gesetzt. Benötigte die Substituierung mehr Zeit oder scheiterte sie an technischen Engpässen (zu wenig Arbeitsspeicher), so wurde bei der Berechnung des Durchschnittswertes, sowohl bei der Zeit für die Substituierung, als auch für das Finden einer ersten Lösung, eine Minute angenommen.

Für die Generierung zufälliger Instanzen eines Constraints werden Variablen benötigt. Wenn nicht anders beschrieben, wird davon ausgegangen, dass jede Variable x_i eine Domäne $D_i = \{0, 1, ..., k\}$ besitzt. Auf die Handhabung von Offsets und Lücken in der Domäne wurde aus Gründen der Vereinfachung und weil bei stichprobenartiger Kontrolle keine signifikanten zeitlichen Unterschiede erkennbar waren, verzichtet.

6.1.1 Das *Count*-Constraint

Die Propagationsdauer des $Count(\{x_1, ..., x_n\}, occ, v)$-Constraints ist in erster Linie von der Anzahl seiner Variablen $x_1, ..., x_n$ sowie occ und deren Domänen abhängig. Welcher Wert $v \in D_1 \cup ... \cup D_n$ konkret gezählt werden soll, hat in der Regel keinen oder nur einen vernachlässigbaren Einfluss auf die Propagationsdauer. Aus diesem Grund wird in der folgenden Versuchsreihe für v der Wert 0 angenommen.

Für die Generierung eines *Count*-Constraints wurden die folgenden Parameter beachtet:

- Die Anzahl Variablen $n \in \{2, 3, 4, 5, 10, 20, 50\}$
- Der maximale Domänenwert $k \in \{1, 2, 3, 4, 5, 10, 20, 50\}$ der Variablen $x_1, ..., x_n$, so dass sich die Domänen $D_1 = ... = D_n = \{0, 1, ..., k\}$ ergeben
- Die Domäne $D_{occ} \in \{\{0\}, \{1\}, \{\lfloor n/2 \rfloor\}, \{\lfloor n/4 \rfloor\}, \{n\}, \{1, 2, ..., \lfloor n/2 \rfloor\}, \{\lceil n/2 \rceil, ..., n\}, \{1, 3, 5, ..., \lceil (n/2) \rceil - 1\}, \{1, 2, 4, 8, 16, ...\}\}$

Für die Versuchsreihe wurde für alle Wertekombinationen der angegebenen Parameter ein *Count*-Constraint erzeugt und getestet. Die hier gewählten Werte sollen möglichst viele verschiedene Kombinationen von Eingaben repräsentieren. So wurden Instanzen mit wenigen (2) oder vielen (50) Variablen, mit wenigen (2) oder vielen (51) Domänenwerten pro Variable sowie mit sehr beschränkter ($\{1\}$) oder sehr offener Domäne ($\{1, 2, ..., \lfloor n/2 \rfloor\}$) für die Häufigkeitsvariable *occ* untersucht. Aus der Menge der möglichen Domänen für D_{occ} wurden die doppelten entfernt (z.B. $\{0\}$ und $\{\lfloor n/4 \rfloor\}$ für $n \in \{1, 2, 3\}$), so dass insgesamt 400 verschiedene *Count*-Constraints getestet wurden. Mit der Bezeichnung $Count - n - k - D_{occ}$ können die einzelnen Constraints eindeutig benannt werden. Mit $Count - 6 - 10 - \{1, 3, 5\}$ wird zum Beispiel das *Count*-Constraint beschrieben, welches fordert, dass die Anzahl der 0-en in den 6 Variablen $x_1, x_2, ..., x_6$ mit Domänen $D_1 = D_2 = ... = D_6 = \{0, 1, 2, ..., 10\}$ gleich einem ungeraden Wert ist.

In Tabelle 6.1 sind die durchschnittlichen Zeiten für das Finden einer ersten Lösung T_{Fst} und das Durchführen der Substituierungen T_{Sub} sowie die Anzahl der Fälle, in denen durch die Substituierung eine Beschleunigung #*Pos* oder eine Verlangsamung #*Neg* auftrat und die Anzahl der Fälle, in denen kein Ergebnis gefunden werden konnte #*Fail*, aufgeführt. Für jedes *Count*-Constraint konnte, wenn keine Substituierung vorgenommen wurde (Original), eine Lösung innerhalb des Zeitlimits von einer Minute gefunden werden. Bei den Substituierungen mittels allgemeinen Substituierungen war dies nicht immer der Fall (#*Fail*). Besonders häufig wurde bei der konfliktbasierten booleschen Skalarisierung keine Lösung gefunden (209 mal), gefolgt von der Support-basierten booleschen Skalarisierung (159 mal), der allgemeinen Regularisierung (153 mal) und der Tabularisierung (142 mal). Die anderen Verfahren, welche die Transformationen aus Abschnitt 5.2.1 verwendet haben, konnten zu allen *Count*-Constraints innerhalb des Zeitlimits eine Lösung finden.

Hervorzuheben ist, dass es bei den direkten Substituierungsarten Fälle gab, bei denen die substituierten CSPs schneller eine erste Lösung fanden als das ursprüngliche CSP (#*Pos*) und dies trotz Berücksichtigung der Zeit für die Substituierung. Am häufigsten war dies bei der direkten booleschen Skalarisierung der Fall (105 mal), gefolgt von der direkten Regularisierung (59 mal).

Es konnte festgestellt werden, dass Verlangsamungen (#*Neg* und #*Fail*) umso häufiger beim Finden einer ersten Lösung durch Substituierung auftreten, je größer der Suchraum $|D_1| * ... * |D_n| = (k + 1)^n$ ist. Dieses Phänomen tritt unabhängig von der verwendeten Substituierung auf. In Abbildung 6.1 ist die Häufigkeit des Auftretens von Verlangsamungen durch Substituierungen in Abhängigkeit von der Suchraumgröße dargestellt. Da die allgemeinen Substituierungen aufgrund der Ermittlung aller Lösungen zum Substituieren immer zu einer Verlangsamung führen, wurden diese in der Darstellung nicht berücksichtigt.

Ab einer Suchraumgröße von $2,39 * 10^{85}$ bei der Regularisierung und booleschen Skalarisierung mittels direkten Transformationen, von $1,19 * 10^{17}$ bei der *Regular*-basierten booleschen Skalarisierung und von $1,17 * 10^{30}$ bei der Erzeugung und Lösung von booleschen *Count*-Constraints, führen alle Substituierungen zu einer Verlangsamung.

Count	Original	$T(\{c\})$	$R(\{c\})$	$R(\{c\}^D)$	$B(\{c\}^S)$	$B(\{c\}^K)$	$B(\{c\}^D)$	$B(\{c\}^{DFA})$	$B(\{c\}^{Count})$
T_{Fst} in s	0,176	22,81	26,653	0,212	25,516	32,653	0,19	0,274	0,248
T_{Sub} in s	0	22,708	26,566	0,041	24,231	30,828	0,019	0,083	0,06
#Pos	-	0	0	59	0	0	105	18	38
#Neg	-	268	247	341	241	191	295	382	362
#Fail	0	142	153	0	159	209	0	0	0

Legende:
T_{Fst} = Zeit für das Finden einer ersten Lösung
T_{Sub} = Zeit für das Durchführen der Substituierung
#Pos = Anzahl Probleme, die schneller gelöst werden konnten
#Neg = Anzahl Probleme, die langsamer gelöst wurden
#Fail = Anzahl Probleme, die nicht innerhalb des Zeitlimits von einer Minute gelöst werden konnten

Tabelle 6.1: Zeitaufwände für das Substituieren von *Count*-Constraints.

Im Diagramm in Abbildung 6.1 ist zu erkennen, dass die Regularisierung und die boolesche Skalarisierung mittels direkten Transformationen auch bei deutlich größeren Suchräumen als bei der *Regular*-basierten booleschen Skalarisierung und der Erzeugung von booleschen *Count*-Constraints noch vereinzelt eine beschleunigende Wirkung haben können.

In Abbildung 6.2 sind die durchschnittlichen Substituierungs- (T_{Sub}) und Lösungszeiten (T_{Fst}) für das Finden einer ersten Lösung für die 400 generierten *Count*-Constraints dargestellt. Die Säulen stellen den Zeitaufwand für das Finden einer ersten Lösung mittels des ursprünglichen CSPs (Original), mittels eines tabularisierten CSPs $T(\{c\})$ oder eines regularisierten CSPs $R(\{c\})$ beziehungsweise mittels eines Support-basierten $B(\{c\}^S)$, eines konfliktbasierten $B(\{c\}^K)$, eines *Regular*-basierten $B(\{c\}^{DFA})$ oder eines *Count*-basierten booleschen *Scalar*-CSPs $B(\{c\}^{Count})$ dar. Die schraffierten (unteren) Säulenanteile repräsentieren dabei den Zeitaufwand, der für die Substituierung des Problems notwendig war. Die grünen Säulen veranschaulichen, dass die allgemeinen Substituierungen (siehe Abschnitt 5.1) verwendet wurden, wohingegen die gelben Säulen veranschaulichen, dass die speziellen Substituierungen (siehe Abschnitt 5.2 und 5.4.2) angewendet wurden. Im Verlauf der Auswertungen zeigte sich, dass die allgemeinen Substituierungen wesentlich zeitaufwendiger sind als die direkten Transformationen, weswegen für die beiden Verfahren $B(\{c\}^{DFA})$ und $B(\{c\}^{Count})$, die eine Regularisierung als Zwischenschritt benötigen, lediglich die direkte und nicht die allgemeine Regularisierung verwendet wurde.

Es ist zu erkennen, dass die allgemeinen Substituierungen zu einer inakzeptablen Verlangsamung (T_{Fst}) führen, die fast ausschließlich auf die hohe Transformationszeit

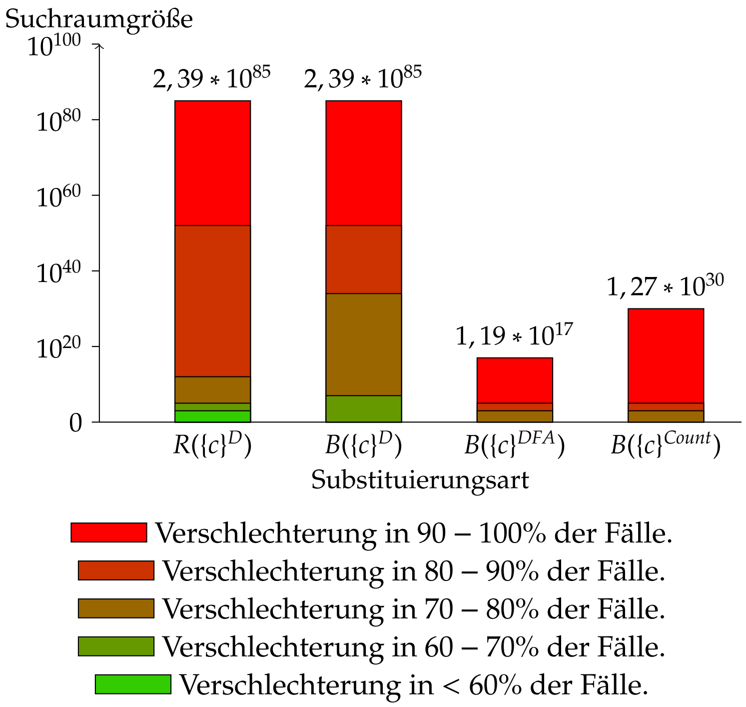


Abbildung 6.1: Die Wahrscheinlichkeit einer Verlangsamung beim Finden einer ersten Lösung für die generierten *Count*-Constraints in Abhängigkeit von der Suchraumgröße $(k+1)^n$.

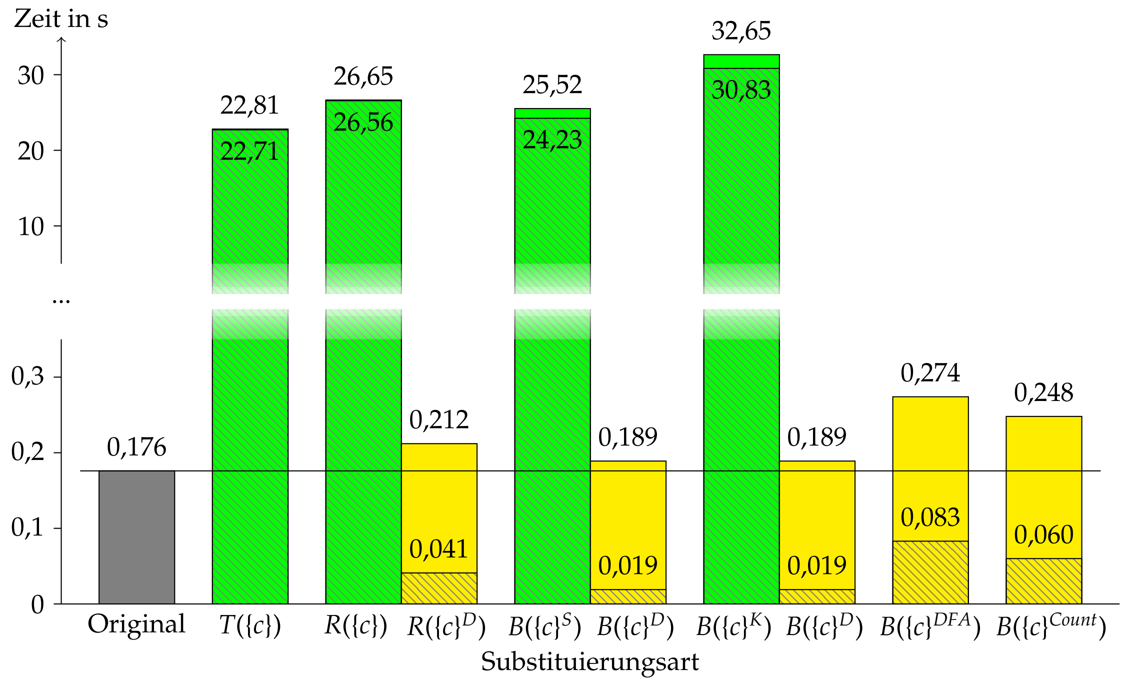


Abbildung 6.2: Die durchschnittlichen Substituierungs- (schraffiert) und Lösungszeiten (für das Finden einer ersten Lösung) für die generierten *Count*-Constraints.

(T_{Sub}) zurückzuführen ist (siehe den Anteil der schraffierten Flächen an den grünen Säulen). Es gilt dabei zu beachten, dass der Zeitaufwand noch höher wäre, wenn ein höheres Zeitlimit als eine Minute gewählt worden wäre. Auf der anderen Seite ist zu erkennen, dass die in Abschnitt 5.2.1 eingeführten direkten Substituierungen für das *Count*-Constraint deutlich bessere Perfomanz zeigen. Die Transformationszeit konnte drastisch verringert werden, so dass die Gesamtzeit bis zum Finden einer ersten Lösung im Schnitt nur noch 7% bis 55% langsamer ist als bei der direkten Verwendung des *Count*-Constraints (Vergleich Original mit $R(\{C\}^D)$, $B(\{C\}^D)$, $B(\{C\}^{DFA})$ und $B(\{C\}^{Count})$).

Eine geringe durchschnittliche Verschlechterung der Lösungszeit ist in diesem Fall kein Hinweis darauf, dass die Substituierungen ihr Ziel verfehlt haben. Dies begründet sich im Detail wie folgt:

1. Es traten mehrere Fälle auf, bei denen die substituierten Varianten schneller waren als die nicht substituierten.
2. Die Interaktionen und Überlagerungen von Constraints in einem CSP sind es, die dessen Lösungsprozess so zeitaufwendig gestalten, und die Interaktion von Constraints wurde an dieser Stelle noch nicht berücksichtigt.
3. Es wurde der Choco-Solver zum Lösen der CSPs verwendet, nicht aber ein spezialisierter boolescher *Scalar-*, *Regular-*, *Count-* oder gleichungsbasierter Solver.

Gleichzeitig konnte gezeigt werden, dass das *Count*-Constraint in akzeptabler Zeit substituiert werden kann (bei allen direkten Sustituierungen in unter 0,1 Sekunden). Dies gilt insbesondere auch für sehr große *Count*-Constraints.

6.1.2 Das *Global Cardinality*-Constraint

Die Propagationsdauer des *Global Cardinality*-Constraints $gcc(\{x_1, ..., x_n\}, \{c_0, ..., c_k\})$ hängt hauptsächlich von der Anzahl der Variablen n, den Domängrößen $|D_1|, ..., |D_n|$ der einzelnen Variablen und den Domänen $D(c_0), ..., D(c_k)$ der Häufigkeitsvariablen ab. Aus diesem Grund wurden für die Generierung der *Global Cardinality*-Constraints die folgenden Parameter berücksichtigt:

- Die Anzahl Variablen $n \in \{2, 3, 4, 5, 10, 20, 50\}$
- Der maximale Domänenwert $k \in \{1, 2, 3, 4, 5, 9, 19, 49\}$ der Variablen $x_1, ..., x_n$ mit $D_1 = ... = D_n = \{0, ..., k\}$
- Die Domänen $D(c_0)$ bis $D(c_k)$:
    1. $D(c_0) = ... = D(c_k) = \{\lfloor \frac{n}{k+1} \rfloor, \lceil \frac{n}{k+1} \rceil\}$
    2. $D(c_i) = \{0, 1, ..., i\}, \forall i \in \{0, ..., k\}$

3. $D(c_i) = \{2^i\}$ wenn $2^{i+1} - 1 \leq n$, $D(c_i) = \{n - 2^i - 1\}$ wenn $2^{i+1} - 1 < n < 2^{i+1}$, sonst $D(c_i) = \{0\}, \forall i \in \{1, ..., k\}$.

Alternativ existieren viele weitere Möglichkeiten für die Domänen der Häufigkeitsvariablen $D(c_i)$. Um die Testreihe überschaubar zu halten, wurden lediglich diese drei Belegungen ausgewählt. Diese decken sowohl Fälle ab, bei denen alle Häufigkeiten sehr konkret (1. und 3.) bzw. sehr offen (2.) sind, als auch Fälle, bei denen die Verteilung der einzelnen Werte sehr gleichmäßig (1.) bzw. ungleichmäßig (2. und 3.) erfolgt.

Die Daten der Testreihe zum *Global Cardinality*-Constraint wurden im Anhang in Tabelle A.1 festgehalten und im Diagramm in Abbildung B.1 visualisiert. Der Aufbau der Tabelle und auch der des Diagrammes entsprechen dabei denen der in Abschnitt 6.1.1 vorgestellten Tabelle 6.1 und des Diagrammes 6.2 zum *Count*-Constraint.

Im Gegensatz zur Substituierung des *Count*-Constraints ist kein direkter Zusammenhang zwischen Suchraumgröße und Zeitdauer bis zum Finden der ersten Lösung feststellbar. Es ergaben sich allerdings extreme Schwankungen bezüglich der Gesamtlösungszeiten (inklusive Zeit für die Substituierung). So waren die CSPs, die durch boolesche Skalarisierung mit direkten Transformationen, die durch *Regular*-basierte boolesche Skalarisierung und die durch Transformation in boolesche *Count*-Constraints erzeugt wurden, teilweise bis zu 330 mal schneller oder aber auch langsamer als das ursprüngliche CSP. Abgesehen von den bereits erwähnten Auffälligkeiten ist die Auswertung des *Global Cardinality*-Constraints im Wesentlichen vergleichbar mit der Auswertung des *Count*-Constraints.

Aus der Tabelle A.1 und dem zugehörigen im Diagramm in Abbildung B.1 gehen hervor, dass die allgemeinen Substituierungen, aufgrund des hohen Zeitaufwands für die Substituierung, immer zu einer Verlangsamung führen. Spannender sind die direkten Transformationen. In 17 ($R(\{c\}^D)$) bis 49 ($B(\{c\}^D)$) von 168 Fällen wurde für die substituierten CSPs schneller eine erste Lösung gefunden als bei dem ursprünglichen CSP. Das ursprüngliche *Global Cardinality*-Constraint hat somit zwar in den meisten Fällen am schnellsten eine Lösung gefunden, allerdings waren die direkt substituierten CSPs im Schnitt teilweise doppelt so schnell. Die direkte Regularisierung ($R(\{c\}^D)$) war im Schnitt lediglich $1,23\%$ langsamer, die anderen direkten Substituierungen hingegen um $17,64\%$ ($B(\{c\}^{Count})$), $19,03\%$ ($B(\{c\}^{DFA})$) bzw. $49,38\%$ ($B(\{c\}^D)$) schneller als das ursprüngliche CSP. Besonders erwähnenswert ist bei der direkten booleschen Skalarisierung ($B(\{c\}^{Count})$), dass die durchschnittliche Dauer zum Finden einer ersten Lösung um fast 50% verringert werden kann, obwohl in 119 von 168 Fällen durch die Substituierung eine Verlangsamung und nur in 49 Fällen eine Beschleunigung auftritt. Daraus lässt sich schlussfolgern, dass die Substituierungen gerade bei zeitaufwendigen Problemstellungen schneller sind als das ursprüngliche *Global Cardinality*-Constraint. Das ist besonders positiv zu sehen, da es genau das Ziel der Arbeit ist, langsame Teile eines CSPs durch schneller lösbare zu ersetzen.

6.1.3 Das *AllDifferent*-Constraint

Der Zeitaufwand für das Propagieren eines *AllDifferent*($\{x_1, ..., x_n\}$)-Constraints hängt, da es lediglich eine Menge von Variablen als Eingabe erhält, allein von dieser Variablenmenge ab. Daher wurden die Anzahl der Variablen n und deren maximaler Domänenwert k für die Domänen $D_1 = ... = D_n = \{0, ..., k\}$ als Parameter berücksichtigt:

- Die Anzahl Variablen $n \in \{2, 3, 4, 5, 10, 20, 50\}$
- Der maximale Domänenwert $k \in \{n, n+1, n+5, n+10, n*2, n*5, n*10\}$ der Variablen $x_1, ..., x_n$

Aus der sich im Anhang befindlichen Tabelle A.2 und dem zugehörigen im Diagramm in Abbildung B.2 lässt sich entnehmen, dass in Einzelfällen vor allem die direkte boolesche Skalarisierung $B(\{c\}^D)$ zu einer Beschleunigung führen kann (in 13 von 49). Wird allerdings die durchschnittlich benötigte Zeit zum Finden einer ersten Lösung betrachtet, so fällt auf, dass das ursprüngliche CSP mit dem *AllDifferent*-Constraint im Durchschnitt mindestens 38 mal so schnell eine erste Lösung finden konnte wie die substituierten CSPs. Diese signifikante Verlangsamung kommt in allen Fällen nicht nur durch den hohen Zeitaufwand für die Substituierung, sondern auch durch einen erhöhten Zeitaufwand für den eigentlichen Lösungsprozess zustande. Darüber hinaus wurde festgestellt, dass der Zeitmehraufwand mit steigender Suchraumgröße ($n * k$) ebenfalls ansteigt.

Die Ergebnisse legen nahe, dass eine Substituierung eines *AllDifferent*-Constraints, besonders eines mit vielen Variablen, nur in sehr wenigen Fällen erfolgen sollte. Es ist davon auszugehen, dass Probleme, bei denen *AllDifferent*-Constraints den wesentlichen Teil der Modellbeschreibung ausmachen (z.B. Sudoku), nicht für die vorgestellten Substituierungen geeignet sind. Auf der anderen Seite existieren gerade bei der direkten booleschen Skalarisierung Fälle, bei denen eine Beschleunigung eintritt. In der Testreihe war das bei 13 von 49 Fällen der Fall. Durch Analyse der vorliegenden Daten konnte dabei kein Zusammenhang ermittelt werden, der angibt, wann eine Beschleunigung vorliegt. Möglicherweise kann ein solcher Zusammenhang allerdings mit den Methoden des maschinellen Lernens ermittelt werden. Können diese Fälle, bei denen Beschleunigungen vorliegen, im Vorfeld der Substituierung ohne großen Zeitaufwand ermittelt werden, so kann die boolesche Substituierung beim *AllDifferent*-Constraint empfehlenswert sein.

6.1.4 Das *AllEqual*-Constraint

Analog zum *AllDifferent*-Constraint hängt der Zeitaufwand für das Propagieren eines *AllEqual*($\{x_1, ..., x_n\}$)-Constraints ebenfalls lediglich von der eingegebenen Variablenmenge ab. Es ergeben sich somit die gleichen Parameter:

- Die Anzahl Variablen $n \in \{2, 3, 4, 5, 10, 20, 50\}$

- Der maximale Domänenwert $k \in \{n, n+1, n+5, n+10, n*2, n*5, n*10\}$ der Variablen $x_1, ..., x_n$

Die Daten für das *AllEqual*-Constraint fallen bezüglich der Anzahl an Verbesserungen und Verschlechterungen ähnlich zu denen des *AllDifferent*-Constraints aus (siehe dazu im Anhang Tabelle A.3 und im Diagramm in Abbildung B.3).

Abgesehen vom konfliktbasierten booleschen Skalarisierungsansatz ($B(\{c\}^K)$), der mit einer Verlangsamung von 16.812,15% keinen geeigneten Substituierungsansatz für das *AllEqual*-Constraint darstellt, befinden sich die durchschnittlichen Verlangsamungen zur Findung einer ersten Lösung in einem akzeptablen Rahmen zwischen 28,17% ($T(\{c\})$) und 153,59% ($B(\{c\}^{DFA})$). Genauso wie beim *AllDifferent*-Constraint ist die Tendenz erkennbar, dass die Substituierungen im Vergleich zum ursprünglichen Constraint langsamer werden, je größer der Suchraum ($n * k$) des Constraints ist.

6.1.5 Das *Scalar*-Constraint

Die Propagationsdauer eines $Scalar(X', \vec{c}, \Re, x_r)$-Constraints ist von verschiedenen Faktoren abhängig. Für die Generierung unterschiedlicher *Scalar*-Constraints wurden die folgenden Parameter berücksichtigt:

- Die Anzahl Variablen $n \in \{2, 3, 4, 5, 10, 20, 50\}$
- Der maximale Domänenwert $k \in \{1, 2, 3, 4, 5, 10, 20, 50\}$ der Variablen $x_1, ..., x_n$
- Das Relationssymbol $\Re \in \{=\}$
- Die Werte für den Vektor $\vec{c} = (c_1, ..., c_n)^T$
    1. $(c_1, ..., c_n)^T = (1, 1, 1, ...)^T$
    2. $(c_1, ..., c_n)^T$ mit $c_i = 2^{i-1}, \forall i \in \{1, ..., n\}$
    3. $(c_1, ..., c_n)^T = (1, -1, 1, -1, 1...)^T$
    4. $(c_1, ..., c_n)^T = (-1, 3, -5, 7, -9...)^T$.
- Die Domäne $D(x_r) \in \{\{k\}, \{n * k\}, \{|c_n| + k\}, \{10\}\}$

Die Werte für den Vektor $\vec{c}$ wurden so gewählt, dass dieser sowohl nur positive (1. und 2.) als auch positiv und negativ gemischte Werte (3. und 4.) sowie gleiche (1.) und sehr unterschiedliche Werte (2., 3. und 4.) enthält. Ohne Beschränkung der Allgemeinheit werden für die Variable x_r nur konstante Werte angenommen. Diese sind so gewählt, dass möglichst viele Constraints Lösungen besitzen und die sich ergebenden Summen möglichst klein bzw. mittel bzw. groß sind. Durch lineare Umformungen kann jedes *Scalar*-Constraint in die beschriebene Form, in der x_r einen konstanten Wert annimmt, überführt werden. Ebenso wurde bewusst auf die weiteren Relationssymbole ($<, \leq, \neq, >, \geq$) verzichtet, da Constraints mit diesen Relationen durch Hinzufügen einer Variablen

x_{n+1} und lineares Umformen in ein Constraint mit =-Relationen überführt werden können. Beispielsweise lässt sich das Constraint

$$Scalar(\{x_1, x_2\}, (1, 2)^T, \leq, x_r) \text{ mit } D_1 = D_2 = D(x_r) = \{1, 2, 3, 4\}$$

in das Constraint

$$Scalar(\{x_1, x_2, x_3\}, (1, 2, 1)^T, =, x_r) \text{ mit } D_1 = D_2 = D(x_r) = \{1, 2, 3, 4\} \text{ und } D_3 = \{0, 1\}$$

überführen. Das ursprüngliche Constraint hat die Lösungen

ϕ_1 mit $\phi_1(x_1) = 1, \phi_1(x_2) = 1$ und $\phi_1(x_r) = 3$,
ϕ_2 mit $\phi_2(x_1) = 1, \phi_2(x_2) = 1$ und $\phi_2(x_r) = 4$ und
ϕ_3 mit $\phi_3(x_1) = 2, \phi_3(x_2) = 1$ und $\phi_3(x_r) = 4$.

und das umgeformte Constraint hat die äquivalenten Lösungen

ϕ'_1 mit $\phi'_1(x_1) = 1, \phi'_1(x_2) = 1, \phi'_1(x_r) = 3$ und $\phi'_1(x_3) = 0$,
ϕ'_2 mit $\phi'_2(x_1) = 1, \phi'_2(x_2) = 1, \phi'_2(x_r) = 4$ und $\phi'_2(x_3) = 1$ und
ϕ'_3 mit $\phi'_3(x_1) = 2, \phi'_3(x_2) = 1, \phi'_3(x_r) = 4$ und $\phi'_3(x_3) = 0$.

In Tabelle A.4 und im Diagramm in Abbildung B.4 im Anhang ist zu erkennen, dass die allgemeinen Substituierungen signifikant langsamer sind als die direkten Substituierungen (19,4s - 32,2s im Vergleich zu 0,026s bis 2,03s). Bei der direkten Regularisierung ($R(\{c\}^D)$), der booleschen Skalarisierung ($B(\{c\}^D)$) und der Erzeugung von booleschen *Count*-Constraints ($B(\{c\}^{Count})$) kann in knapp 25% der Fälle schneller eine erste Lösung gefunden werden, als beim ursprünglichen Constraint (nach Tabelle A.4 in 95, 91 bzw. 96 von jeweils 400 Fällen). Ein Blick auf die durchschnittlichen Zeiten für das Finden einer ersten Lösung offenbart allerdings, dass der ursprüngliche Ansatz, auch im Vergleich zu den direkten Transformationen, im Mittel mehr als zehn mal so schnell ist. Markant ist zusätzlich, dass die Substituierungsdauer bei der direkten booleschen Skalarisierung mit durchschnittlich 0,026s deutlich geringer ausfällt, als die Substituierungsdauern der anderen Verfahren (im Durchschnitt mindestens 1,91s). Eine Substituierung erscheint an dieser Stelle zumindest für die angesprochenen 25% empfehlenswert. Die Ermittlung dieser geeigneten Constraints in akzeptabler Zeit stellt wiederum ein interessantes Forschungsfeld dar.

Auch die Regularisierung, die *Regular*-basierte boolesche Skalarisierung und die Erzeugung von Booleschen *Count*-Constraints können bei genauerer Betrachtung sinnvolle Substituierungen sein. Im Diagramm in Abbildung B.4 ist zwar zu erkennen, dass diese Verfahren für die Versuchsreihe eine im Durchschnitt rund 12 mal so hohe Gesamtlösungsdauer wie das ursprüngliche Constraint haben, allerdings ergibt sich die Gesamtlösungsdauer für die drei Verfahren hauptsächlich aus der jeweiligen Substituierungsdauer. Die Substituierungsdauer ist an dieser Stelle zwar hoch, in einem realen Problem kommt diese allerdings nur einmalig zum Tragen, wohingegen die Dauer zum Finden einer Lösung in jedem Suchschritt von entscheidender Bedeutung sein kann.

Bei zeitaufwendigen Problemen, die möglicherweise mehrere Stunden oder Tage an Berechnungszeit benötigen, kann daher solch eine Substituierung trotzdem sinnvoll sein. Es besteht die Möglichkeit, dass der Zeitaufwand für die Substituierung (gegebenenfalls ein paar Sekunden) vernachlässigbar ist, wenn dadurch erheblich mehr Zeit (Minuten, Stunden oder Tage) beim eigentlichen Lösungsvorgang eingespart werden kann.

6.1.6 Das *Sum*-Constraint

Das *Sum*-Constraint wurde bereits als Teil der Testreihe des *Scalar*-Constraints mit dem Eingabevektor $\vec{c} = (1, 1, ..., 1)^T$ betrachtet. Es erfolgt in diesem Abschnitt daher keine neue Generierung von Problemen, sondern lediglich eine separate Auswertung der Ergebnisse des *Scalar*-Constraints für die entsprechenden Eingaben mit $\vec{c} = (1, 1, ..., 1)^T$.

Aus Tabelle A.5 und im Diagramm in Abbildung B.5 im Anhang ist zu entnehmen, dass die allgemeinen Substituierungen aufgrund der hohen Transformationsdauern ungeeignet sind. Die Transformationsdauern der direkten Regularisierung, der *Regular*-basierten booleschen Skalarisierung und für die Erzeugung von booleschen *Count*-Constraints fallen im Gegensatz zum *Scalar*-Constraint viel geringer aus (zwischen 0,226s und 0,289s statt 1,91s bis 2,03s). Besonders von Bedeutung ist, dass die reine Lösungsdauer bei der *Regular*-basierten booleschen Skalarisierung im Durchschnitt von $0,188s$ auf $0,226s - 0,051s = 0,175s$ reduziert werden konnte. Das macht diese Substituierung für CSPs, die einen sehr zeitaufwendigen Lösungsprozess haben, besonders interessant, da die Substituierung nur einmal durchgeführt werden muss, die Propagation aber gegebenenfalls mehrere tausende Male ausgeführt wird und die reine Lösungsdauer somit bei realen Problemen mehr ins Gewicht fällt als die Substituierungsdauer.

Die Substituierungsdauer der direkten booleschen Skalarisierung ($B(\{c\}^D)$) ist für das *Sum*-Constraint und das *Scalar*-Constraint unverändert. Allerdings ist die reine Lösungsdauer beim *Sum*-Constraint etwa doppelt so hoch wie beim *Scalar*-Constraint. Damit ist die direkte boolesche Skalarisierung im Allgemeinen nicht für die Substituierung von *Sum*-Constraints geeignet. An dieser Stelle lässt sich allerdings festhalten, dass trotz der hohen durchschnittlichen Gesamtlösungsdauer (7,409s im Vergleich zum ursprünglichen *Sum*-Constraint mit 0,188s) in 24 von 100 Fällen eine Beschleunigung erzielt werden konnte. Es gilt das Gleiche wie beim *Scalar*-Constraint: können diese Fälle, in denen eine Beschleunigung eintritt, in kurzer Zeit ermittelt werden, so kann die direkte boolesche Skalarisierung ausgewählter *Sum*-Constraints durchaus sinnvoll sein.

Aufgrund dessen, dass die direkte boolesche Skalarisierung $B(\{c\}^D)$ beim *Sum*-Constraint deutlich langsamer zu einer ersten Lösung führt, als bei beliebigen *Scalar*-Constraints, wurde untersucht, ob die direkte boolesche Skalarisierung sich besser für *Scalar*-Constraints eignet, deren Vektor nicht nur 1-Werte enthält. Es wurden also nur die 300 Fälle betrachtet, die zur *Scalar*-Testreihe gehören aber nicht zur *Sum*-Testreihe. Es stellte sich heraus, dass die direkte boolesche Skalarisierung für diese Fälle im Durchschnitt nur 1,74 (statt 3,15) Sekunden benötigt hat. Damit ist die direkte boolesche Skalarisierung der beste Substituierungsansatz für *Scalar*-Constraints, die keinen Vektor mit nur 1-Werten als Eingabe erhalten. Es lässt sich somit die Empfehlung aussprechen, für *Sum*-Constraints die *Regular*-basierte und für alle anderen *Scalar*-Constraints die direkte boolesche Skalarisierung zu verwenden.

6.1.7 Das *Table*-Constraint

Die Propagationsdauer des $Table(\{x_1, ..., x_n\}, T)$-Constraints hängt von seinen Variablen, deren Domänen und den zulässigen Tupeln T ab, darauf basierend wurden die folgenden Parameter bestimmt:

- Die Anzahl Variablen $n \in \{2, 5, 10, 20, 50\}$
- Der maximale Domänenwert $k \in \{1, 5, 10, 20, 50\}$ der Variablen $x_1, ..., x_n$
- Die Anzahl der Tupel $|T| \in \{10, 100, 1.000, 10.000\}$
- Die Tupel $t_j = (v_1, ..., v_n)$ mit $v_i = Random(0, k),\ \forall i \in \{1, ..., n\}, \forall j \in \{1, ..., |T|\}$

Die Tupel werden bei der gegebenen Parameterbeschreibung völlig zufällig erzeugt. Es ist nicht davon auszugehen, dass solche zufällig über den Domänenraum verteilten Tupel in realistischen Problemen auftreten. Dort ist es nach eigenen Erfahrungen oftmals eher der Fall, dass die Tupel eine gewisse Nähe zueinander haben. Da dies für verschiedene Probleme allerdings sehr unterschiedlich aussehen kann, wurde an dieser Stelle auf spezifische Tupel verzichtet. Übersteigt die vorgegebene Tupelanzahl $|T|$ die Größe des Suchraumes $(k + 1)^n$, so wurde die tatsächliche Tupelanzahl auf die Größe des Suchraumes reduziert.

Da für die Substituierung des *Table*-Constraints nach Abschnitt 5.2.7 die allgemeinen Substituierungen verwendet werden, enthalten die Tabelle A.6 und das Diagramm in Abbildung B.6 im Anhang keine Daten zu direkten Transformationen ($B(\{c\}^D)$ bzw. $R(\{c\}^D)$). Aus der Tabelle A.6 ist entnehmbar, dass keine der Substituierungen zu einer Beschleunigung der Gesamtlösungsdauer führen konnte und in allen Fällen eine erhebliche Verlangsamung auftrat (die Gesamtlösungsdauer stieg auf das drei- bis 150-fache). Die Support-basierte boolesche Skalarisierung ($B(\{c\}^S)$, durchschnittliche Gesamtlösungsdauer 0,621s) ist, abgesehen vom ursprünglichen *Table*-Constraint (durchschnittliche Gesamtlösungsdauer 0,211s), am schnellsten. Dass die Support-basierte boolesche Skalarisierung für diese Testreihe deutlich schneller agiert als die konfliktbasierte, liegt an der geringen Anzahl der Tupel in Bezug auf die Suchraumgröße ($|T|\ /\ (k+1)^n$). Aufgrund des allgemeinen Substituierungsansatzes, in dem das zu substituierende Constraint zunächst gelöst werden muss, ist die Support-basierte boolesche Skalarisierung ($B(\{c\}^S)$ in jedem Fall langsamer als der ursprüngliche Ansatz ($\#Neg = 300$). Die geringe Verlangsamung um nur ca. Faktor zwei verdeutlicht allerdings, dass die einzelnen Support-basierten booleschen Skalarisierungen relativ gleichmäßig etwas langsamer sind als die ursprünglichen Constraints. Bei den anderen Substituierungen sind die Schwankungen hingegen deutlich stärker.

Da keine direkten Substituierungen existieren, kann keine generelle Empfehlung ausgesprochen werden, *Table*-Constraints zu substituieren. Die Regularisierung von *Table*-Constraints kann bei schwer zu lösenden CSPs allerdings trotzdem sinnvoll sein. Dies liegt daran, dass sich die reine Lösungsdauer bei der Regularisierung im Vergleich zum ursprünglichen *Table*-Constraint von durchschnittlich $0,211s$ auf $0,180s$ verringert hat und die reine Lösungszeit im Vergleich zur Substituierungszeit bei komplexen Problemen stärker ins Gewicht fällt.

Des Weiteren gilt, dass die Regularisierung und damit verbunden auch die *Regular*-basierte boolesche Skalarisierung und die Erzeugung von booleschen *Count*-Constraints noch besser agieren können, wenn die Tupel nicht zufällig im Suchraum verteilt sind. Sind die Tupel so gewählt, dass sie kompakt in einem levelbasierten DFA repräsentiert werden können, also mit wenigen Zuständen und Übergängen, so reduziert sich in der Regel auch die Propagationsdauer für solch einen Automaten und damit verbunden auch die Lösungsdauer des entsprechenden CSPs.

6.1.8 Das *Regular*-Constraint

Die Propagationsdauer des *Regular*($\{x_1, ..., x_n\}, M$)-Constraints ist, genauso wie die Propagationsdauer des *Table*($\{x_1, ..., x_n\}, T$)-Constraints, von der Anzahl der Variablen n, deren Domänen $D_1, ..., D_n$ und dem levelbasierten DFA M abhängig. Im Gegensatz zum *Table*-Constraint nehmen die Tupel T beim *Regular*-Constraint nur indirekt Einfluss auf die Propagationsdauer, da aus ihnen zunächst ein levelbasierter DFA M erstellt wird, dessen Aufbau die Propagationsdauer direkt beeinflusst. Es werden somit die gleichen Parameter wie beim *Table*-Constraint verwendet.

Aus Tabelle A.7 im Anhang kann entnommen werden, dass die allgemeinen Substituierungen nicht zu einer Verringerung der Gesamtlösungsdauer führen. Es ist allerdings auch zu erkennen, dass bei der Tabularisierung und der Support-basierten booleschen Skalarisierung nur eine geringfügige Verlangsamung von durchschnittlich 5,66% und 10,34% auftritt. Die konfliktbasierte boolesche Skalarisierung ($B(\{c\}^K)$) ist an dieser Stelle besonders ungeeignet, da sie nur für knapp die Hälfte der *Regular*-Constraints (für 132 von 300) eine Lösung finden konnte.

Die direkte boolesche Skalarisierung, die in diesem Fall der *Regular*-basierten booleschen Skalarisierung entspricht (siehe Abschnitt 5.2.8) und die Erstellung von booleschen *Count*-Constraints führen zwar nur in wenigen Fällen (16 bzw. 34 von 300) zu einer Verringerung der Gesamtlösungsdauer, allerdings kann dadurch die durchschnittlich benötigte Zeit in beiden Fällen mehr als gedrittelt werden. Diese Beobachtung deutet darauf hin, dass eine Substituierung von *Regular*-Constraints nicht in jedem Fall aber in speziellen Fällen besonders sinnvoll ist. Die Schwierigkeit besteht darin, in möglichst kurzer Zeit herauszufinden, welche *Regular*-Constraints dies betrifft.

Die beiden Diagramme i den Abbildungen B.6 und B.7 könnten den Eindruck erwecken, dass die Tabularisierung im Allgemeinen schneller ist als die Regularisierung, immerhin führt eine Tabularisierung von *Regular*-Constraints nur zu einer geringen Verlangsamung (siehe $T(\{c\})$ im Diagramm in Abbildung B.7), eine Regularisierung von *Table*-Constraints hingegen führt zu einer deutlich höheren Verlangsamung (siehe $R(\{c\})$ im Diagramm in Abbildung B.6). Das liegt insbesondere daran, dass die Tupel zufällig gewählt wurden und somit nur selten ein kompakter, kleiner, levelbasierter DFA existiert bzw. ermittelt werden kann. Zulässige Tupel von Constraints aus realen Problemen sind allerdings in der Regel nicht zufällig verteilt, sondern folgen einem Muster, welches sich oftmals kompakt als levelbasierter DFA darstellen lässt. Des Weiteren wurden bei der Evaluation von *Table*- und *Regular*-Constraints bisher nur solche Constraints betrachtet, die über maximal 10.000 Tupel verfügen. Bei deutlicher Erhöhung der Tupelanzahl wird eine Tabularisierung in der Regel erheblich ineffizienter, weswegen in [10] eine Tabularisierung nur für Constraints mit maximal 10.000 Tupel empfohlen wird. Eine Evaluation von realen Problemen wird im nachfolgenden Abschnitt 6.2 betrachtet und kann diese Empfehlung soweit unterstützen, dass bei sehr großen Tupelmengen die Tabularisierung nicht mehr anwendbar ist oder zu einer unzumutbaren Verlangsamung führt.

6.1.9 Fazit zur Substituierung einzelner globaler Constraints

Nachdem in den vorherigen Abschnitten der Einfluss der verschiedenen Substituierungen auf die Gesamtlösungsdauer evaluiert wurde, werden in diesem Abschnitt die wesentlichen Erkenntnisse der Versuchsreihe noch einmal zusammengefasst.

Die allgemeinen Substituierungen

Die allgemeinen Substituierungen $T(c), R(c), B(c^S)$ und $B(c^K)$ können aufgrund der Art, wie sie durchgeführt werden, bei dieser Testreihe nicht zu einer Beschleunigung führen. In der Regel entstehen durch sie erhebliche Verlangsamungen. Abgesehen vom *AllEqual*-Constraint ergeben sich diese Verlangsamungen allerdings hauptsächlich aus der sehr zeitaufwendigen Substituierung, der eigentliche Lösungsprozess wird in der Regel nur geringfügig verlangsamt. Das bedeutet, dass diese Substituierungen trotz der schlechten Testergebnisse in der Praxis sinnvoll sein können. Bei realen Problemen gilt unabhängig von der Substituierungsart immer, dass die Substituierung nur einmal durchgeführt werden muss und sich somit auch die Substituierungsdauer nur einmal negativ auf die Gesamtlösungsdauer auswirkt. Die reine Lösungsdauer (ohne Zeit für die Substituierung) hat sich hingegen in vielen Fällen deutlich verringert. Bestes Beispiel dafür ist das *Global Cardinality*-Constraint. Bei diesem wurden bei den allgemeinen Verfahren durchschnittlich $18s$ bis $27,1s$ für die Substituierungen aber nur durchschnittlich $0,1s$ bis $1,5s$ für die eigentliche Lösungsfindung benötigt. Das entspricht einer Beschleunigung der reinen Lösungszeit von $71,54\%$ bis $98,24\%$. Musste in den hier generierten CSPs, mit nur einem Constraint, nur wenige Male propagiert werden um eine Lösung zu finden, so muss dies in komplexeren CSPs viel häufiger geschehen. Aus diesem Grund schlägt sich die Reduzierung der reinen Lösungsdauer in komplexeren CSPs viel stärker nieder. Wird bei jedem Propagieren $0.01s$ gespart, so führt das bei 1.000 Propagationen schon zu einer Zeitersparnis von $10s$ und bei 100.000 Propagationen zu einer Zeitersparnis von $1000s \hat{=} 16,67min$.

Die direkten Sustituierungen

Die direkten Substituierungen $R(c^D)$, $B(c^D)$ und die aus den Regularisierungen abgeleiteten $B(c^{DFA})$ und $B(c^{Count})$ führten zwar auch nur bei dem *Global Cardinality*-Constraint und beim *Regular*-Constraint zu einer Verringerung der Gesamtlösungsdauer, allerdings wurde die Zeit für die Substituierung so sehr verringert, dass nun häufiger Fälle auftraten, in denen die Verlangsamung in einem akzeptablem Bereich liegt. Auch an dieser Stelle gilt die Argumentation, die schon bei den allgemeinen Verfahren aufgeführt wurde, dass sich der positive Effekt der Substituierungen in komplexeren CSPs verstärken kann. Des Weiteren ist auffällig, dass für jede Constraint-Art (*Count*, *Global Cardinality* etc.) mit jeder der vier direkten Substituierungen Fälle aufgetreten sind, die zu einer Beschleunigung führten. Am seltensten war dies beim *AllDifferent*- und beim *AllEqual*-Constraint und am meisten beim *Global Cardinality*-Constraint der Fall. Dass immer wieder einzelne Substituierungen besonders wirksam sind, kann unter anderem auf zwei Arten genutzt werden:

1. In einem Portfolio-Ansatz
2. Durch vorheriges Anwenden eines geeigneten Klassifikators

Bei einem Portfolio-Ansatz [79] werden verschiedene Modelle erzeugt, die das gleiche Problem beschreiben und echt oder quasi-parallel ausgeführt werden. Sobald ein System eine Lösung gefunden hat, können alle parallel laufenden Lösungsverfahren gestoppt und die Lösung zurückgegeben werden. Ein solches Verfahren benötigt weniger Zeit, aber dafür mehr Recheneinheiten (Prozessoren), was in der heutigen vernetzten Welt, wo auf mehrere Rechner zugegriffen werden kann und diese in der Regel auch mehrere Prozessoren haben, ein handhabbares Problem ist.

Das maschinelle Lernen und dessen Kombination mit der Constraint-Programmierung erfahren gerade einen Aufschwung [75, 138, 141]. Einen geeigneten Klassifikator für die bestmögliche Anwendung von Substituierungen zu finden, könnte ebenfalls ein Problem sein, dass durch maschinelles Lernen gelöst werden kann. Erste Versuche dahingehend wurden in der in Zusammenhang mit dieser Arbeit entstandenen Publikation [94] bereits untersucht. Diese Arbeit ist auf die Unterscheidung der Tabularisierung und der Regularisierung mittels Random Forests Classification [36] (Zufallswaldklassifikation) beschränkt, lässt aber positive Schlüsse auf eine Erweiterung um weitere Substituierungen zu, insofern die Trainingsdaten umfassend genug sind.

Existiert ein ausreichend schneller und präziser Klassifikator, so kann dieser vorab entscheiden, ob und wenn ja, welche Substituierung eingesetzt werden soll. Ein perfekter Klassifikator, also einer, der keine Zeit zum Entscheiden benötigt und immer richtig klassifiziert, verhält sich wie ein perfekter Portfolio-Ansatz, nur dass nur ein Prozess dafür notwendig ist.

Prinzipiell konnte nicht erwartet werden, durch die Substituierungen schneller Lösungen zu finden als durch das jeweilige globale Constraint. Tatsächlich musste davon ausgegangen werden, dass die ursprünglichen globalen Constraints schneller sind als die Substituierungen. Schließlich sind die globalen Constraints und deren Propagationsalgorithmen über Jahre, teilweise sogar über Jahrzehnte hinweg erforscht und entwickelt worden. Wäre eine Substituierung durch ein Constraint (Tabularisierung oder Regularisierung) im Durchschnitt schneller als das ursprüngliche Constraint, so sollte der ursprüngliche Propagator durch den des *Table*- bzw. *Regular*-Propagators ausgetauscht werden. Die in Abschnitt 4.2 beschriebenen Vorteile der Substituierung können erst eintreten, wenn mehrere Constraints durch ein einzelnes Constraint ersetzt werden. Um den gesteigerten Nutzen der Substituierungen bei komplexeren CSPs zu belegen, werden die Substituierungen im folgenden Abschnitt 6.2 für umfangreichere Anwendungsbeispiele genutzt.

Boolesche Skalarisierungen

Bei den booleschen Skalarisierungen war das eigentliche Ziel, FD-CSPs für pseudoboolesche lineare Solver wie zum Beispiel *PBSugar* [4] oder *SAT4J* [5] anwendbar zu machen. Bisher wurde aber lediglich der FD-Solver Choco zum Lösen der entstandenen booleschen *Scalar*-CSPs verwendet. Etwas überraschend konnte allerdings festgestellt werden, dass die boolesche Skalarisierung auch ohne Verwendung eines pseudo-booleschen linearen Solvers zur Beschleunigung des Lösungsvorganges führen kann. Nichts desto trotz ist davon auszugehen, dass bei vollständiger Substituierung des Ausgangs-CSPs die Verwendung eines solchen pseudo-booleschen linearen Solvers schneller zu einer ersten Lösung führt. Schlussendlich wurde an dieser Stelle aus den folgenden drei Gründen auf die Verwendung eines solchen Solvers verzichtet:

- Der Choco-Solver verfügt nicht direkt über einen pseudo-booleschen linearen oder SAT-Solver und ein fairer Laufzeitvergleich über verschiedene Solver hinweg ist nur schwierig umsetzbar.
- Nicht alle komplexen Probleme können vollständig in akzeptabler Zeit substituiert werden und somit ist die Verwendung eines solchen spezialisierten Solvers dann in der Folge nicht möglich.

- Es traten auch ohne Verwendung eines pseudo-booleschen linearen Solvers bereits nicht zu verachtende Beschleunigungen durch die boolesche Skalarisierung auf.

Ursache für die bereits auftretende Beschleunigung ohne Verwendung spezialisierter Solver kann unter anderem darin liegen, dass sich komplexe globale Constraints, wie zum Beispiel das *Count-*, das *Global Cardinality-* oder das *Sum*-Constraint, in einfache lineare Gleichungen über boolesche Variablen überführen lassen.

Wirksamkeit der Substituierungen bei einzelnen globalen Constraints

Die verschiedenen Substituierungen sind für die verschiedenen globalen Constraints unterschiedlich effektiv. Beim *Count-*, *Global Cardinality-*, *AllEqual-* und beim *Regular*-Constraint konnte zum Beispiel, abgesehen vom ursprünglichen CSP, die direkte boolesche Skalarisierung am schnellsten eine Lösung finden. Beim *Scalar*-Constraint war es die direkte Regularisierung und beim *Summen*-Constraint die levelbasierte boolesche Skalarisierung. Das *AllDifferent*-Constraint ist das einzige, für das keine Substituierung in akzeptabler Zeit eine erste Lösung findet. Eine Substituierung des *AllDifferent*-Constraints ist deswegen in der Regel nicht zu empfehlen. Eine Substituierung des *AllDifferent-* oder vergleichbarer Constraints kann aber trotzdem sinnvoll sein, wenn es zum Beispiel das einzige nicht boolesche (oder *Regular-* oder *Table-*) Constraint eines CSPs ist, um somit einen booleschen (oder *Regular-* oder *Table-*) Solver nutzbar zu machen. Des Weiteren kann bei der Substituierung auch durch Vereinigung der substituierten Constraints eine Beschleunigung erzielt werden. Im Allgemeinen (außer beim *Global Cardinality-*Constraint) zeichnet sich ab, dass die Substituierungen weniger zielführend sind, je größer der Suchraum des ursprünglichen Constraints ist.

Wie effektiv die einzelnen Substituierungen sind, kann sich schlussendlich erst in der Praxis bei echten Problemen zeigen. Diesbezüglich folgt im nächsten Abschnitt eine Evaluation des Einsatzes von Substituierungen bei verschiedenen typischen Praxisanwendungen.

6.2 Anwendungsbeispiele für die Substituierung von CSPs

In diesem Abschnitt werden konkrete Anwendungsbeispiele aus der Praxis und der *CSPLib* [83], einer Bibliothek für Constraint-Probleme, hinsichtlich ihrer Substituierbarkeit und den Einfluss solcher Substituierungen auf die Lösungsgeschwindigkeit untersucht. Wenn nicht anders beschrieben, ist mit der Anwendung eines Substituierungsverfahrens immer sowohl die Transformation als auch die Lösung des entsprechendes CSPs gemeint.

Im Gegensatz zu den primitiveren CSPs in Abschnitt 6.1 sind die folgenden CSPs deutlich komplexer und zeichnen sich insbesondere durch das Überlagern mehrerer

Constraints aus. Zwei Constraints c_1 und c_2 werden an dieser Stelle als überlagernd bezeichnet, wenn sie über mindestens eine *shared Variable* verfügen ($sharedVariables(c_1, c_2) = scope(c_1) \cap scope(c_2)$). Aufgrund der erhöhten Komplexität wurde auch das Zeitlimit für das Substituieren und Lösen von einer Minute auf fünf Minuten angehoben.

Da die CSPs teilweise über mehrere hundert Constraints verfügen und eine Auflistung aller zu substituierenden Constraints in der Folge zu umfangreich ist, werden neue, zusammenfassende Bezeichner für die verschiedenen Substituierungen eingeführt:

$$Substituierungsart^{Spezifikation}$$

Die *Substituierungsart* gibt dabei an, dass es sich entweder um die $Tabularisierung = T$, die $Regularisierung = R$ oder die $boolesche\ Skalarisierung = B$ handelt. Ist keine Spezifikation angegeben, so wird von den allgemeinen Substituierungsverfahren (siehe Abschnitt 5.1) ausgegangen. Der Bezeichner D steht für die direkte Substituierung (siehe Abschnitte 5.2 und 5.3). Im Falle der booleschen Skalarisierung sind zusätzlich noch die Spezifikationen *DFA* für die *Regular*-basierte boolesche Skalarisierung und *Count* für die Erzeugung von booleschen *Count*-Constraints möglich (siehe Abschnitte 5.2.8 und 5.4). Zusätzlich kann es vorkommen, dass nicht alle Constraints eines CSPs substituiert werden. Ist dies der Fall, so wird bei den einzelnen Anwendungsbeispielen explizit darauf hingewiesen, welche Constraints wie substituiert werden.

Des Weiteren können alle Ansätze, die eine Regularisierung beinhalten (R, R^D, B^{DFA} und B^{Count}), als zusätzliche Spezifikation das Schnittmengensymbol $\cap$ erhalten. Wie in Abschnitt 5.4.2 erläutert, erfolgt die *Regular*-basierte boolesche Skalariesierung B^{DFA} und das Erzeugen von booleschen *Count*-Constraints B^{Count} jeweils in zwei Schritten. Im ersten Schritt wird das ursprüngliche CSP P regularisiert, so dass ein CSP $P^{Regular}$ entsteht und dieses wird anschließend in ein boolesches *Scalar*-CSP transformiert. Somit kann die levelbasierte DFA-Schnittmengenbildung auf das regularisierte CSP $P^{Regular}$ angewendet werden. Im Anschluss daran können die so entstandenen DFAs zur gegebenenfalls weiteren Substituierung (B^{DFA} und B^{Count}) und Lösung (R und R^D) verwendet werden. Welche DFAs dabei jeweils vereint werden, ist zu jeder Problemstellung einzeln aufgeführt.

6.2.1 Schichtplanungsprobleme

An dieser Stelle soll das in der Motivation eingeführte Beispiel 1.1 und in Beispiel 2.5 als FD-CSP dargestellte Schichtplanungsproblem generalisiert und anschließend substituiert und gelöst werden. Um eine größere Bandbreite an konkreten Schichtplanungsproblemen lösen zu können, wurden die folgenden Verallgemeinerungen zu den Anforderungen in Beispiel 1.1 getroffen.

A1 Der Planungszeitraum umfasst w Wochen mit jeweils d Tagen für w Mitarbeiter. Vorher waren $w = 8$ und $d = 7$ fest gewählt.

A2 Es gibt s verschiedene Schichttypen und den zusätzlichen Typ „Freischicht“ = 0, wobei die weiteren Schichten $1, ..., s$ zeitlich nacheinander beginnen. Zuvor wurde von 3 Schichten: Früh = 1, Spät = 2 und Nacht = 3 ausgegangen. Die Verallgemeinerung lässt z.B. auch einen Schichtplan mit vier verschiedenen Schichten, die jeweils sechs Stunden lang sind, zu.

A3 Es gibt für jeden Wochentag und für jede Schicht eine untere und obere Schranke, wie viele Mitarbeiter benötigt werden. Vorher wurde eine konkrete Schichtstärkenmatrix angegeben, die für jeden Tag für jede Schicht eine exakte Vorgabe aufführte.

A4 Am vorletzten Tag jeder Woche soll die gleiche Schicht wie am darauffolgenden Tag sein. Das entspricht einer Verallgemeinerung von 7-Tage-Wochen auf d-Tage-Wochen.

A5 An mindestens s_{min} aufeinander folgenden Tagen soll für eine Person die gleiche Schicht festgelegt sein. Das entspricht einer Verallgemeinerung von 2 auf s_{min}.

A6 An nicht mehr als s_{max} aufeinander folgenden Tagen soll für eine Person die gleiche Schicht festgelegt sein. Das entspricht einer Verallgemeinerung von 4 auf s_{max}.

A7 Die Schichtfolge soll sich an das Prinzip der Vorwärtsrotation halten (keine Änderung).

A8 Innerhalb von zwei Wochen sollen immer mindestens zwei freie Tage liegen (keine Änderung).

Für die individuelle Anpassung an reale Probleme werden gegebenenfalls weitere Änderungen oder das Hinzufügen oder Entfernen von Constraints notwendig. Für die Verdeutlichung des Nutzens von Substituierungen sollen diese Verallgemeinerungen aber genügen.[1] Ein Schichtplanungsproblem der soeben beschriebenen Art lässt sich eindeutig durch *Schicht-w-d-s-M-*s_{min}-s_{max} mit $w, d, s, s_{min}, s_{max} \in \mathbb{N}, M \in \mathbb{N}^{(s+1)\times d}$ angeben. Für die folgende Testreihe wurden die Parameter mit folgenden Werten belegt:

- Die Anzahl Wochen $w \in \{4, 5, 6, 7, 8, 9, 10\}$
- Die Anzahl Tage pro Woche $d \in \{6, 7\}$
- Die Anzahl verschiedener Schichten $s \in \{3, 4\}$ zuzüglich Freischicht
- Die Schichtanforderungsmatrix $M \in \mathbb{N}^{(s+1)\times d}$ mit $M_{i,j} = \{\lfloor \frac{w}{s} \rfloor, \lceil \frac{w}{s} \rceil\}$ für $i \in \{0, 1, ..., s\}$, $j \in \{1, ..., d\}$

---

[1] Parallel zu dieser Arbeit wurde das Problem in die CSPlib, eine Bibliothek für Constraint-Probleme, aufgenommen ([150]).

- Die minimale Anzahl an gleichen Schichtbelegungen $s_{min} \in \{1, 2, 3\}$ für aufeinanderfolgende Tage

- Die maximale Anzahl an gleichen Schichtbelegungen $s_{max} \in \{3, 4\}$ für aufeinanderfolgende Tage

Damit ergeben sich $7 * 2 * 2 * 3 * 2 = 168$ verschiedene Instanzen des Schichtplanungsproblems, die mit dem CSP 15 beschrieben werden. Die Werte für die Matrix M sind dabei so gewählt, dass alle Schichten gleich verteilt sind. Das entspricht nicht allen realen Szenarien. In der Praxis kommt es vor, dass die Nachtschichten oder Schichten an Wochenenden mit weniger Personal besetzt sind. Da mit der hier dargestellten Berechnung von $M_{i,j}$ allerdings für alle Instanzen eine Anforderung gegeben werden kann, ohne diese explizit für alle 168 Fälle einzeln angeben zu müssen und es mit der Belegung nicht unnötig häufig zu Instanzen ohne Lösung kommt, wurde sich schlussendlich für diese Variante entschieden.

Das CSP 15 ergibt sich aus der direkten Übersetzung der verallgemeinerten Bedingungen eins bis acht in Variablen, Domänen und Constraints. Es wurde dabei darauf geachtet, wenn immer möglich, globale Constraints zu verwenden. Es sollte somit bereits vor den Substituierungen ein möglichst schnell zu lösendes CSP entstehen. Die Implikationen $(A \rightarrow B)$ wurden dabei intern durch $\neg A \vee B$ ersetzt. Die Funktion $mod(k)$ berechnet den Wert $(k \; mod \; (w * d)) + 1$ und verhindert, dass auf eine Variable x_i mit einem Index i größer als $w * d$ oder kleiner als 1 zugegriffen wird und dass auf die Variable x_{w*d} wieder die Variable x_1 folgt. Damit wird ein zyklisches Verhalten geschaffen, wodurch das in Abschnitt 1.1 aufgeführte Rotationsprinzip verwendet werden kann. Für alle verwendeten Constraints wurde in Kapitel 5 eine direkte Substituierung in *Regular*- und boolesche *Scalar*-Constraints eingeführt.

Da die allgemeinen Substituierungsverfahren für die hier aufgeführten *Global Cardinality*- und *Count*-Constraints aufgrund der immens hohen Tupelanzahl, die diese Constraints beschreiben, und den damit verbunden Zeit- und Ressourcenaufwand nicht anwendbar sind, wurde auf eine Verwendung dieser Verfahren generell verzichtet. Es wurde stattdessen für jede Parameterkombination *Schicht-w-d-s-M-*s_{min}-s_{max} das ursprüngliche CSP (Original), das durch direkte Regularisierung R^D, das durch *Regular*-basierte boolesche Skalarisierung B^{DFA}, das durch Erzeugung von booleschen *Count*-Constraints B^{Count} und das durch direkte boolesche Skarlarisierung B^D erzeugte CSP automatisch erstellt.

CSP 15: $P = (X, D, C)$ mit:

$$X = \{x_1, ..., x_{w*d}\} \mathrel{\hat{=}} X^{2d} \qquad (w * d \text{ Variablen, A1})$$

$$\text{mit } X^{2d}_{i,j} = X_{(i-1)*d+j} \text{ für } i \in \{1, ..., w\}, j \in \{1, ..., d\}$$

$$D = \{D_1, ..., D_{w*d} \mid D_1 = ... = D_{w*d} = \{0, 1, ..., s\}\} \qquad (s + 1 \text{ Schichttypen, A2})$$

$$C = \{GCC(\{X^{2d}_{1,i}, X^{2d}_{2,i}, ..., X^{2d}_{w,i}\}, \{M_{0,i}, M_{1,i}, ..., M_{s,i}\} \mid \forall i \in \{1, ..., d\}\} \cup \qquad (\text{Personalanforderungen, A3})$$

$$\{X^{2d}_{j,d-1} = X^{2d}_{j,d} \mid \forall j \in \{1, ..., w\}\} \cup \qquad (\text{Wochenendgleichheit, A4})$$

$$\{(x_i \neq x_{mod(i+1)}) \rightarrow AllEqual(\{x_i, x_{mod(i-1)}, ..., x_{mod(i-s_{min}+1)}\}) \mid \forall i \in \{1, ..., w * d\}\} \cup \qquad (\text{min. } s_{min} \text{ Wiederholungen, A5})$$

$$\{AllEqual(x_i, x_{mod(i+1)}, x_{mod(i+2)}, ..., x_{mod(i+s_{max}-1)}) \rightarrow (x_i \neq x_{mod(i+s_{max})}) \mid \forall i \in \{1, ..., w * d\}\} \cup \qquad (\text{max. } s_{max} \text{ Wiederholungen, A6})$$

$$\{(x_i \leq x_{mod(i+1)}) \vee (x_{mod(i+1)} = 0) \mid \forall i \in \{1, ..., w * d\}\} \cup \qquad (\text{Vorwärtsrotation, A7})$$

$$\{Count(\{x_i, x_{mod(i+1)}, ..., x_{mod(i+2*d)}\}, 0, c) \mid \forall i \in \{1, ..., w * d\}, c \text{ ist eine neue Variable mit } D_c = \{2, 3, ..., 2 * d\}\} \qquad (\text{min. 2 freie Tage, A8})$$

Für die Verfahren, die eine Regularisierung beinhalten, wurde jeweils ein zweites CSP erzeugt, bei dem nach der Regularisierung eine Vereinigung bestimmter Constraints durchgeführt wurde. Diese CSPs sind in der Folge mit $R^{D^\cap}$, $B^{DFA^\cap}$ und $B^{Count^\cap}$ bezeichnet. Als Vereinigungsstrategie wurde in diesem Fall die gleiche Variablenreihenfolge gewählt. Das bedeutet, es wurden alle levelbasierten DFAs vereint, die die Variablenreihenfolge $x_1, x_2, ..., x_{w*d}$ besitzen. Das entspricht allen Constraints außer denen, die über die Variable x_{w*d} hinaus laufen, um zu garantieren, dass auf die letzte Schichtbelegung wieder die erste folgen kann. Zusätzlich wurden alle levelbasierten DFAs von der Vereinigung ausgenommen, deren Variablen Lücken bezüglich der Variablenreihenfolge $x_1, x_2, ..., x_{w*d}$ aufweisen. In CSP 15 betrifft das die ursprünglichen *Global Cardinality*-Constraints (A3).

Es ergeben sich somit $d * (s + 1)$-viele levelbasierte DFAs für die *Global Cardinality*-Constraints, ein levelbasierter DFA für alle ursprünglichen Constraints, die dafür sorgen, dass die Bedingungen A4 bis A8 zwischen x_1 und x_{d*w} erfüllt sind und je ein levelbasierter DFA für jedes ursprüngliche Constraint, das über die Variable x_{d*w} hinaus läuft. Ein Beispiel für diese letzte Kategorie ist das Constraint $Count(\{x_i, x_{mod(i+1)}, ..., x_{mod(i+2*d)}\}, 0, c)$ für $i = w * d$ und $D_c = \{2, 3, ..., 2 * d\}$, welches für die Variablen $x_i, x_{mod(i+1)}, ..., x_{mod(i+2*d)}$ fordert, dass diese mindestens zweimal den Wert 0 enthalten (mindestens zwei Freischichten innerhalb von zwei Wochen). Die Variable x_{w*d} tritt in diesem Constraint vor der Variable x_1 auf, was gegen die Variablenordnung $x_1, x_2, ..., x_{d*w}$ verstößt.

Die *Global Cardinality*-Constraints zur Beschreibung der Personalanforderungen A3 wurden bewusst nicht in die Vereinigung einbezogen, da diese bei der Vereinigung mit den levelbasierten DFAs der anderen Constraints zu einem exorbitant großen levelbasierten DFA führen.

Die levelbasierten DFAs, die die geforderte Variablenordnung verletzen, erfüllen alle die Variablenordnung $x_{\lfloor w*d/2 \rfloor}, ..., x_{d*w}, x_1, ..., x_{\lfloor w*d/2 \rfloor - 1}$. Somit können alle levelbasierten DFAs, die jeweils ursprüngliche Constraints repräsentieren und gegen die erste Variablenordnung verstoßen, zu einem levelbasierten DFA vereint werden. Ein regularisiertes CSP 15 kann somit durch $d * (s + 1) \ + \ 1 \ + \ 1$ viele *Regular*-Constraints repräsentiert werden. Durch die Vereinigung treten die in Abschnitt 4.2 aufgeführten Vorteile auf. In der folgenden Testreihe ist die Zeit, die für die levelbasierte Schnittmengenbildung benötigt wurde, in der Zeit für die Substituierung T_{Sub} inkludiert.

Im Gegensatz zu den CSPs in Abschnitt 6.1, die nur über jeweils ein Constraint verfügen und zu denen ohne Verwendung von Transformation innerhalb des angegebenen Zeitlimits immer eine Lösung gefunden werden konnte, sind die CSPs jetzt sehr komplex und es ist nicht gewährleistet, dass für jedes CSP eine Lösung existiert oder gefunden werden kann. Aus diesem Grund können die substituierten CSPs nicht nur schneller oder langsamer, sondern auch gleich schnell zu einem Ergebnis oder einem Widerspruch gelangen bzw. nach Erreichen des festgesetzten Zeitlimits abbrechen. Die einzelnen Probleminstanzen werden den folgenden drei Kategorien zugeordnet:

- Probleme, die durch Substituierung schneller gelöst werden können (*Pos*): durch die entsprechende Substituierung konnte die Gesamtlösungsdauer um mindestens 5% verringert werden.
- Probleme, die durch Substituierung langsamer gelöst werden (*Neg*): durch die entsprechende Substituierung wurde die Gesamtlösungsdauer um mindestens 5% erhöht.
- Probleme, die durch Substituierung genauso schnell gelöst werden (*Eq*): durch die entsprechende Substituierung hat sich die Gesamtlösungsdauer um weniger als 5% verändert *oder* es hat weder das ursprüngliche CSP, noch das substituierte eine Lösung innerhalb des Zeitlimits gefunden.

Sollte zu einem CSP keine Lösung existieren, so trifft Gleichheit (*Eq*) auch dann zu, wenn das ursprüngliche CSP und das substituierte dies mit einem Zeitunterschied von weniger als 5% ermitteln.

In Tabelle 6.2 sind die durchschnittlichen Zeiten für das Finden einer ersten Lösung T_{Fst} und das Durchführen der Substituierungen T_{Sub} sowie die Anzahl der Fälle, in denen durch die Substituierung eine Beschleunigung #*Pos*, eine Verlangsamung #*Neg* (jeweils von mindestens 5%) oder keine wesentliche Veränderung #*Eq* auftrat und die Anzahl der Fälle, in denen kein Ergebnis gefunden werden konnte #*Fail*, sowie die Anzahl der Fälle, in denen die jeweilige Substituierung dazu führte, dass von allen Vorgehen am schnellsten eine Lösung gefunden werden konnte #*Best*, aufgeführt. Im Diagramm in Abbildung 6.3 sind die Zeiten für das Finden einer ersten Lösung T_{Fst} und das Durchführen der Substituierungen T_{Sub} (schraffiert) visualisiert. Es ist zu erkennen, dass die Substituierungsdauer mit maximal $1,121s$ bei der *Regular*-basierten booleschen

Scalar	Original	R^D	$R^{D^\cap}$	B^D	B^{DFA}	$B^{DFA^\cap}$	B^{Count}	$B^{Count^\cap}$
T_{Fst} in s	40,401	4,458	1,266	64,656	23,175	5,851	26,957	5,767
T_{Sub} in s	0	0,435	0,824	0,551	1,121	0,956	0,509	0,873
#*Pos*	-	109	104	11	37	76	37	70
#*Eq*	-	7	7	16	6	2	6	3
#*Neg*	-	52	57	141	125	90	125	95
#*Fail*	7	0	0	9	3	1	1	0
#*Best*	55	41	69	0	0	1	0	2

Tabelle 6.2: Zeitaufwände für das Substituieren und Lösen der 168 Schichtplanungsprobleme.

Skalarisierung im Verhältnis zur ursprünglichen Lösungsdauer (40,401s) mit 2,775% keinen signifikanten Einfluss hat. Auf die durchschnittliche Lösungszeit und die Anzahl der im Zeitlimit gefundenen Lösungen betrachtet, schneidet nur eine Substituierung (die direkte boolesche Skalarisierung) schlechter ab als das ursprüngliche CSP: es waren zwei CSPs weniger lösbar (#*Fail* = 9 statt 7) und es wurde durchschnittlich 60% mehr Zeit zum Lösen benötigt (64,656s statt 40,401s). Alle anderen Substituierungen führten im Durchschnitt zu einer Beschleunigung von 33,276% bis 96,866%.

Auffällig ist, dass zwar nur die direkte Regularisierung (ohne und mit levelbasierter DFA-Vereinigung) zu mehr Beschleunigungen (109 bzw. 104) als zu Verlangsamungen (52 bzw. 57) führt, aber trotzdem alle Substituierungen bis auf die direkte boolesche Skalarisierung zu einer durchschnittlichen Beschleunigung führen. Das bedeutet, dass die Beschleunigungen vor allem bei den Problemen auftreten, die besonders zeitaufwendig sind. Dieses Verhalten ist besonders positiv zu bewerten. Eine geringe prozentuale Verlangsamung bei schnell lösbaren Problemen (z.B. in wenigen Sekunden) führt in der Regel zu Problemen, die immer noch schnell gelöst werden können. Im Gegenzug dazu kann eine hohe prozentuale Beschleunigung bei schwer zu lösenden Problemen dazu führen, dass diese in einem vorgegebenen Zeitlimit oder mit gegebenen Ressourcenbegrenzungen eine Lösung finden, die vorher nicht ermittelbar war. Besonders die direkte Regularisierung mit levelbasierter DFA-Vereinigung scheint schwere Probleme besonders schnell lösen zu können. Mit Ihrer Hilfe konnten alle Probleminstanzen gelöst werden und die Lösungsdauer zum Finden einer ersten Lösung signifikant reduziert werden (Beschleunigung von 96,866%). Aber auch die auf der Regularisierung und levelbasierten DFA-Vereinigung basierenden booleschen Skalarisierungen ($B^{DFA\cap}$ und $B^{Count^\cap}$) führen zu einer erheblichen Beschleunigung ($95,518\%$ bis $95,726\%$). Diese Verfahren können besonders dann noch an Bedeutung gewinnen, wenn für die entstandenen CSPs jeweils noch spezialisierte Solver verwendet werden.

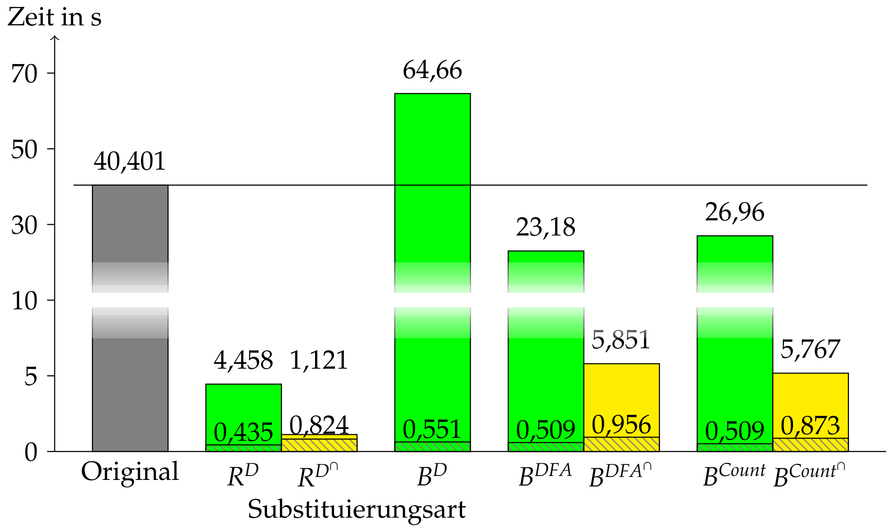


Abbildung 6.3: Die durchschnittlichen Substituierungs- und Lösungszeiten (für das Finden einer ersten Lösung) für die generierten Schichtplanungsprobleme[2].

[2] Es gilt die unterschiedlichen Skalierungen der Ordinate im Raum von 0 bis 10 und 10 bis 70 zu beachten.

Aus Tabelle 6.2 lassen sich zusätzlich die Anzahl der Fälle entnehmen, in denen der jeweilige Ansatz am schnellsten eine Lösung gefunden hat (#*Best*-Wert). Es ist zu erkennen, dass der ursprüngliche Ansatz (Original, 55) und die Regularisierung ohne (R^D, 41) und mit levelbasierte DFA-Vereinigung ($R^{D^\cap}$, 69) die mit Abstand meisten der 168 Probleme am schnellsten lösen. Da allerdings keines der Verfahren immer am besten agierte, wurde die durchschnittliche Zeit für das Finden einer ersten Lösung eines Portfolio-Ansatzes ermittelt, welcher angibt, wie viel Zeit durchschnittlich benötigt wird, wenn immer das schnellste der vorgestellten Vorgehen ausgewählt wird. Bei einem Portfolio-Ansatz werden in der Regel verschiedene Modelle oder gleiche Modelle mit verschiedenen Solvern oder Solver-Einstellungen unabhängig voneinander parallel gelöst. Das erste Verfahren, das die gewünschte Lösung oder die gewünschte Anzahl an Lösungen gefunden hat, beendet alle Suchvorgehen. Bei echt paralleler Ausführung entspricht dieses Vorgehen zeitlich dem des besten der parallel ausgeführten Verfahren. In diesem Fall führt ein Portfolio-Ansatz zu einer durchschnittlichen Zeit von 1,066 Sekunden zum Finden einer ersten Lösung, was keine signifikante Beschleunigung im Vergleich zur Regularisierung mit levelbasierter DFA-Vereinigung ist. Das bedeutet im Umkehrschluss, dass die Regularisierung mit levelbasierter DFA-Vereinigung in fast allen Fällen am besten oder nur wenig langsamer als die anderen Verfahren ist.

Da viele Schichtplanungsprobleme mit Hilfe der Substituierungen in weniger Zeit als im Vorfeld erwartet gelöst werden konnten, wurde eine zweite Testreihe mit großen Probleminstanzen erstellt. Dafür wurde der Anzahl-Wochen-Parameter w geändert, so dass sich dieser aus dem Wertebereich $11, 12, 13, 14, 15, 16$ ergibt. Der Suchraum wird mit

Schicht	Original	R^D	$R^{D^\cap}$	B^D	B^{DFA}	$B^{DFA^\cap}$	B^{Count}	$B^{Count^\cap}$
T_{Fst} in s	206,772	94,327	47,775	233,819	202,299	126,483	222,993	115,810
T_{Sub} in s	0	0,664	1,038	1,498	5,031	2,915	1,041	1,464
#*Pos*	-	95	124	13	23	39	13	83
#*Eq*	-	39	15	84	78	77	85	34
#*Neg*	-	8	3	45	41	26	44	25
#*Fail*	87	27	14	95	70	35	80	35
#*Best*[3]	14	43	70	1	1	4	0	9

Tabelle 6.3: Zeitaufwände für das Substituieren und Lösen der 168 großen Schichtplanunsprobleme.

jeder zusätzlichen Woche um den Faktor $(s+1)^d$ größer. Folglich bewegt er sich für die so beschriebenen Probleme zwischen $(3+1)^{11*6} = 5,445 * 10^{39}$ und $(4+1)^{16*7} = 1,926 * 10^{78}$ Schichtplan-Kombinationen.

In Tabelle 6.3 und im Diagramm in Abbildung 6.4 sind die durchschnittlichen Zeiten für das Finden einer ersten Lösung T_{Fst} und das Durchführen der Substituierungen T_{Sub} sowie die Anzahl Beschleunigungen #*Pos*, Verlangsamungen #*Neg* usw. für die Substituierung und Lösung der großen Schichtplanungsprobleme aufgeführt. Es ist zu erkennen, dass der ursprüngliche Ansatz für Probleme dieser Größe in den veranschlagten fünf Minuten nur für ca. die Hälfte (81 von 168) der Probleme eine Lösung oder einen Widerspruch finden kann. Lediglich die direkte boolesche Skalarisierung fand in acht weiteren Fällen keine Lösung. Alle anderen Verfahren konnten mehr Probleme lösen. Im Gegensatz zu den kleinen Problemen führt die boolesche Skalarisierung zu einer Verlangsamung von 7,845%. Auch der zuvor beste Ansatz, die direkte Regularisierung mit Vereinigung der levelbasierten DFAs, wirkt auf den ersten Blick nicht mehr so effektiv wie bei den kleinen Problemen (durchschnittliche Beschleunigung von nur noch 76,995% anstelle von 96,866%). Diese Verringerung der Beschleunigung stellt ein verzerrtes Bild dar, das dadurch entsteht, dass das Zeitlimit von fünf Minuten beim ursprünglichen CSP sehr häufig erreicht wird. In 87 von 168 Fällen müssten bei dem ursprünglichen Ansatz mehr als fünf Minuten in die Durchschnittsberechnung mit einbezogen werden. Bei der direkten Regularisierung mit anschließender levelbasierter DFA-Vereinigung ist dies hingegen nur 14 mal der Fall.

Das Diagramm in Abbildung 6.5 vermittelt einen besseren Eindruck wie viel besser oder schlechter die einzelnen Substituierungen agieren. Die Abszisse gibt jeweils die verwendete Substituierungsart und die Ordinate die Anzahl der Schichtplanungsproble-

---

[3] Die #*Best*-Werte ergeben in Summe nur 142 statt 144, da zu zwei Instanzen keine Vorgehensweise innerhalb des Zeitlimits eine Lösung fand oder feststellen konnte, dass keine Lösung existiert.

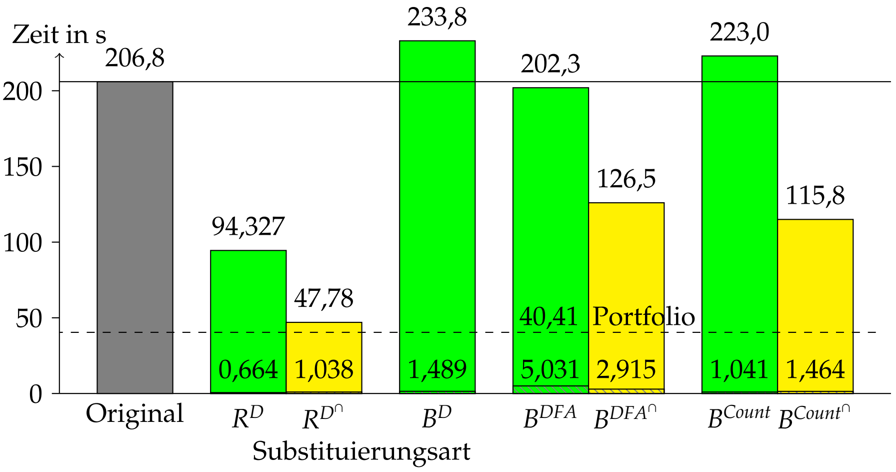


Abbildung 6.4: Die durchschnittlichen Substituierungs- und Lösungszeiten (für das Finden einer ersten Lösung) für die generierten Schichtplanungsprobleme.

me, in denen es aufgrund der Substituierung zu einer 2 (bzw. 5, 10, 50 oder 100)-fachen Beschleunigung oder Verlangsamung kam, an. Es ist zu erkennen, dass die Regularisierung ohne und mit levelbasierter DFA-Vereinigung nur in drei bzw. einem Fall zu einer Verlangsamung führt, die mindestens doppelt so viel Zeit in Anspruch nimmt wie das ursprüngliche Problem. Im Gegensatz dazu führt die Regularisierung in 81 Fällen zu einer Beschleunigung um mindestens Faktor 2 und in 22 von diesen Fällen sogar zu einer Beschleunigung um mindestens Faktor 100. Bei der Regularisierung mit levelbasierter DFA-Vereinigung erhöhen sich diese Werte sogar noch einmal auf 109 und 26.

Des Weiteren ist zu erkennen, dass die levelbasierte DFA-Vereinigung in allen Fällen zu mehr beschleunigten Fällen führt (Vergleich R^D mit $R^{D^\cap}$, B^{DFA} mit $B^{DFA^\cap}$ und B^{Count} mit $B^{Count^\cap}$). Die *Regular*-basierte boolesche Skalarisierung und das Erzeugen von booleschen *Count*-Constraints führen sowohl mit als auch ohne levelbasierter DFA-Vereinigung nicht zu Fällen, die mindestens 50 mal so schnell gelöst werden können wie das ursprüngliche CSP, allerdings zu solchen, die mindestens 100 mal so langsam sind. Das bedeutet, dass es gerade bei den Ansätzen der booleschen Skalarisierung von Bedeutung ist, diejenigen Probleminstanzen im Vorfeld der Substituierung zu erkennen, die zu einer Beschleunigung oder Verlangsamung führen. Bei den *Regular*-basierten Ansätzen spielt dieses Vorhersagen für das vorgestellte Schichtplanungsproblem hingegen nur eine untergeordnete Rolle, da bei ihr nie von einer signifikanten Verlangsamung (Beschleunigungsfaktor kleiner 0,2) auszugehen ist.

Zusammenfassend lässt sich sagen, dass sich die Regularisierung für die in diesem Abschnitt vorgestellte Variante der Schichtplanungsprobleme außerordentlich gut eignet. Aber auch die booleschen Skalarisierungsverfahren, die eine Regularisierung beinhalten

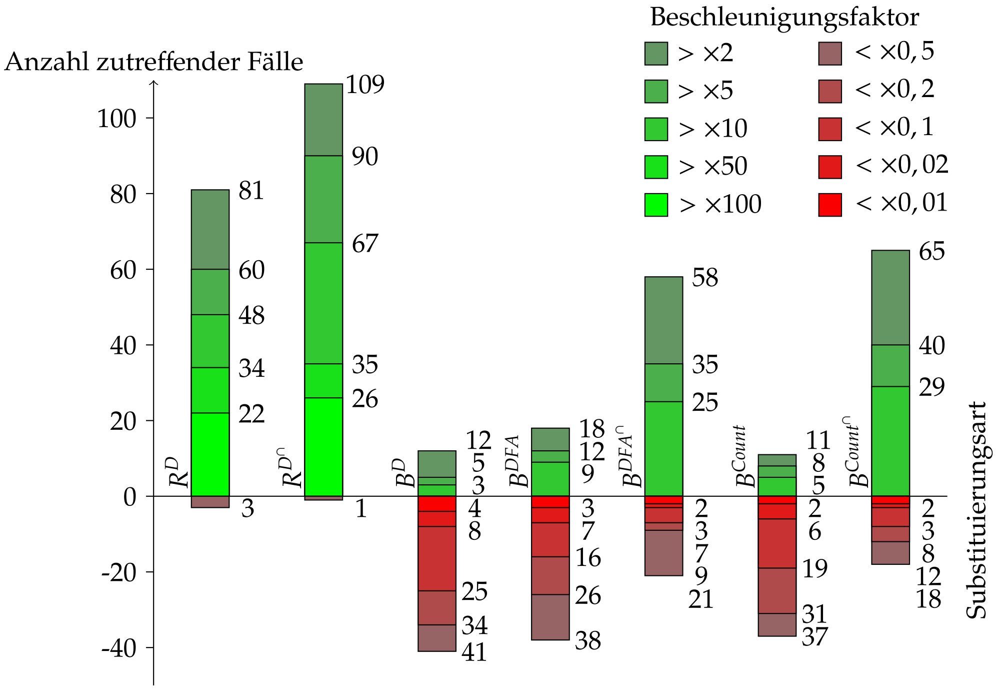


Abbildung 6.5: Eine Veranschaulichung der Anzahl an signifikanten Beschleunigungen und Verlangsamungen, die sich aus den verschiedenen Substituierungen für die generierten Schichtplanungsprobleme ergeben.

B^{DFA} und B^{Count}, führen sowohl bei kleinen Problemen ($w \in \{4, 5, ..., 10\}$) als auch bei großen Problemen ($w \in \{11, 12, ..., 16\}$), welche Gebrauch von der levelbasiertern DFA-Vereinigung machen, zu einer durchschnittlichen Beschleunigung des Lösungsvorganges.

Die Beschleunigung durch die Substituierungen ergibt sich vermutlich aus dem Vorhandensein von Implikationen und or-Constraints im ursprünglichen CSP (siehe A5 und A6 bzw. A7 in CSP 15). Diese sind für den Menschen zwar intuitiv verständlich, durch die Verschachtelung der Constraints aber nur langsam propagierbar. Dass die levelbasierte DFA-Vereinigung für dieses Problem zu einer so deutlichen Beschleunigung führt, liegt daran, dass die aus der Regularisierung entstandenen Automaten teilweise Wissen sehr redundant repräsentieren und sich die einzelnen Automaten mit Ausnahme der Regularisierungen der sich aufblähenden *Global Cardinality*-Constraints (A3) sehr schnell und kompakt vereinigen lassen.

6.2.2 Das Black Hole-Problem

Das Black Hole (auf deutsch: schwarzes Loch)-Problem [116] ist ein Kartenspiel für eine Person. Gespielt wird mit einem 52-Karten-Blatt (4 Farben, 13 Werte pro Farbe). Zu Spielbeginn wird das Pik-Ass in die Mitte des Tisches gelegt. Dieses steht symbolisch für das schwarze Loch. Die anderen 51 Karten werden gemischt und zufällig auf 17 Kartenfächer mit jeweils 3 Karten aufgeteilt. Es sind jederzeit alle Karten sichtbar. Der Spieler kann in jedem Zug die oberste Karte eines Kartenfächers nehmen und diese auf das schwarze Loch legen. Es dürfen allerdings nur solche Karten auf das schwarze Loch gelegt werden, die im Wert eins höher oder tiefer als die oberste Karte des schwarzen Lochs sind. Farben spielen dabei keine Rolle und jedes Ass zählt gleichzeitig im Wert als 1 und als 14, damit ist es möglich auf ein Ass sowohl eine Karte mit Wert 2 oder 13 (bei einem klassischen Kartenspiel der König) zu legen. Untere Karten eines jeden Dreier-Fächers dürfen erst verwendet werden, wenn die darüberliegenden Karten bereits dem schwarzen Loch zugefügt wurden. Gewonnen ist das Spiel, wenn alle 51 Karten der 17 Fächer dem schwarzen Loch zugefügt wurden.

Die praktische Relevanz des Black Hole-Problems ist auf den ersten Blick nicht direkt ersichtlich, allerdings ist es verwandt mit dem Problem des Rangierens von Wagons in Bahnhöfen. In beiden Fällen müssen der Reihe nach Bewegungen durchgeführt werden, um ein Gut (Karte beim Black Hole-Problem, Zug beim Wagonrangieren) an die gewünschte Zielposition zu bringen. Dabei herrschen starke zeitliche Abhängigkeiten zwischen den Gütern (ein im Weg stehender Wagon muss ebenso wie eine im Weg liegende Karte zunächst beseitigt werden).

Das Black Hole-Problem wurde von Gent et al. [61] für eine Reihe von Solvern entwickelt. Wir nutzen eine der einfachsten und deklarativsten Modellierungen von Dekker et al. [49] für das Problem. In dieser Variante werden 52 Variablen $X = \{x_0, ..., x_{51}\}$ mit Domänen $D_0 = ... = D_{51} = \{0, 1, 2, ..., 51\}$ angelegt, wobei die Werte 0 bis 51 dabei den

Karten $A\spadesuit$, $2\spadesuit$, ..., $K\spadesuit$, $A\clubsuit$, $2\clubsuit$,..., $K\clubsuit$, $A\heartsuit$, $2\heartsuit$, ..., $K\heartsuit$, $A\diamondsuit$, $2\diamondsuit$, ..., $K\diamondsuit$ entsprechen. Jede Variable x_i gibt also den Wert d an, den die i-te Karte des finalen Ablagestapels annimmt. Die partielle Belegung ϕ mit $\phi(x_0) = 0, \phi(x_1) = 14, \phi(x_2) = 2$ und $\phi(x_3) = 27$ repräsentiert zum Beispiel den Ablagestapel $0, 14, 2, 27$ also $A\spadesuit$ $2\clubsuit$ $3\spadesuit$ $2\heartsuit$.

Für die Modellierung der Constraints werden 52 weitere Variablen $Y = \{y_0, ..., y_{51}\}$ mit Domänen $D_0^y = ... = D_{51}^y = \{0, 1, 2, ..., 51\}$ verwendet, die invers zu X agieren. Das bedeutet, dass für jede Variable x_i mit Wert j eine Variable y_j mit Wert i existiert.

Die Werte (1 bis 51) der zu Beginn gegebenen 51 sichtbaren Karten der 17 Fächer können in einer eindimensionalen Tabelle $T = (t_1, ..., t_{51})$ abgelegt werden, so dass t_1 gleich der obersten, t_2 der mittleren und t_3 der unteren Karte des ersten Fächers entspricht. Die Werte für t_4 bis t_6 entsprechen den Werten der Karten des zweiten Fächers usw. Der Wert t_i der Tabelle T entspricht somit immer dem Wert der $(((i-1) \mathit{mod}\ 3) + 1)$-ten Karte des $(((i-1) \mathit{div}\ 3) + 1)$-ten Kartenfächers. Die Bedingung $(i =)\ y_{t_1} < y_{t_2}\ (= j)$, dass der Wert i von Variable y_{t_1} kleiner ist als der Wert j der Variablen y_{t_2}, drückt für die invers angebundenen Variablen x_i und x_j mit $x_i = t_1$ und $x_j = t_2$ aus, dass i kleiner j sein muss. Die Karte mit Wert t_1 muss also eher auf den Ablagestapel gelegt werden als die Karte mit Wert t_2, dazwischen können sich allerdings auch noch weitere Karten befinden. Basierend auf den bisherigen Beschreibungen beschreibt das folgende CSP 16 das Black Hole-Problem:

CSP 16: $P = (X, D, C)$ mit:

$$X = \{x_0, ..., x_{51}, y_0, ..., y_{51}\} \qquad \text{(2 $*$ 52 Variablen)}$$

$$D = \{D_0, ..., D_{51}, D_0^y, ..., D_{51}^y \mid D_0 = ... = D_{51} = D_0^y = ... = D_{51}^y = \{0, 1, ..., 51\}\} \qquad \text{(52 Werte \& Positionen)}$$

$$C = \{x_0 = 0\} \cup \qquad \text{(1. Karte $A\spadesuit$)}$$

$$\{\mathit{Inverse}(\{x_0, ..., x_{51}\}, \{y_0, ..., y_{51}\})\} \cup \qquad \text{(Zusammenhang X \& Y)}$$

$$\{(y_{t_i} < y_{t_{i+1}}) \mid \forall i \in \{0, 1, 3, 4, 6, 7, 9, 10, ..., 46, 47, 49, 50\}\} \cup \qquad \text{(Fächer-Ordnung)}$$

$$\{(|x_i - x_{i+1}|\ \mathit{mod}\ 13 \in \{1, 12\}) \mid \forall i \in \{0, ..., 50\}\} \qquad \text{(Ablage-Ordnung)}$$

Das bisher nicht erwähnte *Inverse*-Constraint bindet zwei geordnete Variablenmengen X und Y derart aneinander, dass die Bedingungen $x_i = j$ und $y_j = i$ für alle Variablen $x_i \in X$ und $y_j \in Y$ erfüllt sind [1]. Zusätzlich bedingt das *Inverse*-Constraint die paarweise Verschiedenheit der Werte der Variablen in X und die paarweise Verschiedenheit der Werte der Variablen in y. Dies entspricht zwei *AllDifferent*-Constraints über den Variablenmengen X bzw. Y.

Die Fächer-Ordnungs-Constraints garantieren, dass eine Karte a, die beim Start in einem Fächer über der Karte b liegt, im Ablagestapel unter der Karte b liegen muss. Die Ablage-Ordnungs-Constraints gewährleisten, dass aufeinanderfolgende Karten auf dem Ablagestapel sich im Wert um 1 unterscheiden, wobei die Farbe der Karte vernachlässigt wird und berücksichtigt ist, dass das Ass sowohl als kleiner zwei ($|x_i - x_{i+1}|\ mod\ 13 = 1$), als auch als größer König ($|x_i - x_{i+1}|\ mod\ 13 = 12$) betrachtet werden kann.

Für die Substituierungen ergeben sich die folgenden Schwierigkeiten:

- Für das *Inverse*-Constraint liegt bisher keine direkte Transformation vor.
- Das *Inverse*-Constraint beschreibt eine zu große Menge sowohl an gültigen als auch an ungültigen Tupeln, als dass diese mit den allgemeinen Verfahren substituiert werden können.
- Da das *Inverse*-Constraint nicht substituiert werden kann, ist das CSP nicht vollständig substituierbar, wodurch eine Verwendung von spezialisierten Solvern (z.B. SAT-, *Table*- oder *Regular*-Solver) erschwert oder ausgeschlossen wird.
- Für die Ablage-Ordnungs-Constraints existiert keine direkte Transformation. Hier kann allerdings die allgemeine Substituierung verwendet werden, da die Lösungsmenge jedes einzelnen Constraints sehr gering ist.

Da für das *Inverse*-Constraint keine direkte Substituierung vorliegt, wurde auf eine Substituierung dieses Constraints verzichtet. Hinzu kommt, dass das *Inverse*-Constraint zwei *AllDifferent*-Constraints impliziert und bei *AllDifferent*-Constraints ohnehin durch Substituierungen drastische Verlangsamungen auftreten (siehe Abschnitt 6.1.3).Da das *Inverse*-Constraint nicht substituiert wird, kann das CSP nicht komplett substituiert werden. Eine Verwendung eines booleschen oder reinen *Regular*-Solvers ist somit nicht möglich.

Die restlichen Constraints zur Beschreibung der Fächer- und der Ablage-Ordnung wurden jeweils substituiert. Die Fächer-Constraints können dabei als Summen-Constraints angesehen und, wie in Abschnitt 5.2.6 beschrieben, transformiert werden. Für die Ablage-Constraints existiert keine direkte Substituierung. Da die gültige Tupelmenge eines jeden Ablage-Constraints mit $8 * 52 = 416$ relativ gering ist, können an dieser Stelle im Gegensatz zum *Inverse*-Constraint die allgemeinen Verfahren aus Abschnitt 5.1 verwendet werden.

Für die 50 verschiedenen Instanzen des Black Hole-Problems wurden das ursprüngliche CSP (*Original*) mit den aus Tabularisierung (T), Regularisierung (R), konfliktbasierter (B^K), Support-basierter (B^S) und *Regular*-basierter boolescher Skalarisierung (B^{DFA}) entstehenden CSPs sowie das boolesche *Count*-CSP (B^{Count}) miteinander verglichen. Zusätzlich wurde für jedes CSP, welches durch Regularisierung erzeugt wurde, ein CSP erstellt, bei dem vor dem Lösen oder weiterem Transformieren eine levelbasierte DFA-Vereinigung durchführt wurde ($R^{\cap}, B^{DFA^{\cap}}, B^{Count^{\cap}}$). Als Vereinigungsstrategie wurde, wie bei den Schichtplanungsproblemen, die gleiche Variablenreihenfolge gewählt. Das bedeutet, es wurden alle levelbasierten DFAs vereint, die die Variablenreihenfolge $x_0, x_1, ..., x_{51}, y_0, y_1, ..., y_{51}$ ohne Lücken besitzen. Für die gegebene Problembeschreibung bedeutet das, dass alle DFAs aus Ablage-Constraints und dem ($x_0 = 0$)-Constraint zu

Black Hole	Original	T	R	$R^{\cap}$	B^S	B^K	B^{DFA}	$B^{DFA^{\cap}}$	B^{Count}	$B^{Count^{\cap}}$
T_{Fst} in s	481,63	98,29	118,46	165,19	573,29	565,40	213,40	218,78	212,21	416,17
T_{Sub} in s	0	0,247	0,34	0,708	1,968	1,197	1,055	1,876	0,345	1,008
#Pos	-	38	37	31	0	0	28	30	30	10
#Eq	-	6	6	12	38	38	13	12	12	29
#Neg	-	6	7	7	12	12	9	8	8	11
#Fail	38	6	5	11	47	47	13	8	12	7
#Best	0	21	9	2	0	0	6	4	5	1

Tabelle 6.4: Zeitaufwände für das Substituieren und Lösen der 50 Black Hole Instanzen.

einem levelbasierten DFA vereinigt werden. Die DFAs der Fächer-Constraints hingegen werden mit Ausnahme sehr günstiger Kartenverteilungen nicht vereint. Es ergeben sich somit ein *Inverse*-Constraint, in der Regel $17 * 2 = 34$ Fächer-Constraints und ein Ablage-Constraint.

Es wurden 50 zufällige Startkonfigurationen erzeugt und unter Anwendung der unterschiedlichen Transformationen gelöst. Als Zeitlimit wurden dafür zehn Minuten festgelegt. In Tabelle 6.4 und im Diagramm in Abbildung 6.6 sind die durchschnittlichen Zeiten der einzelnen Verfahren für das Finden einer ersten Lösung T_{Fst} und das Substituieren T_{Sub} sowie die Anzahl der Beschleunigungen *#Pos*, gleichbleibenden *#Eq*, Verlangsamungen *#Neg* und fehlgeschlagenen *#Fail* Fälle (inklusive Anzahl der Fälle, die das Zeitlimit von zehn Minuten überschritten) pro Substituierungsart aufgeführt. Des Weiteren wurde aufgeführt, für wie viele Probleme die jeweilige Substituierungsart am schnellsten zu einer Lösung führt *#Best*. Die Summe der *#Best* ergibt in diesem Fall nur 48 und nicht 50, da zu zwei Instanzen kein Vorgehen eine Lösung innerhalb des Zeitlimits von zehn Minuten finden konnte. Es ist zu erkennen, dass insgesamt die Tabularisierung am besten für diese Testreihe geeignet ist. Sie weist die geringste durchschnittliche Lösungszeit auf (98,29s) und die mit Abstand meisten Fälle, in denen die Lösung als erstes gefunden wurde (21). Der ursprüngliche Ansatz hingegen ist hier nie am schnellsten und scheitert in 38 von 50 Fällen ohne Lösung innerhalb des Zeitlimits. Lediglich die Support- und die konfliktbasierte boolesche Skalarisierung führen zu durchschnittlichen Verschlechterungen beim Lösungsvorgang (12 Verlangsamungen, Rest gleichbleibend). Abgesehen vom booleschen *Count*-CSP mit vorher durchgeführter levelbasierter DFA-Vereinigung führen alle anderen Verfahren im Durchschnitt mindestens zu einer 54-prozentigen Beschleunigung.

Neben der Tabularisierung schneidet auch die Regularisierung besonders gut ab. Sie ist in der Lage die meisten Probleme zu lösen (45 von 50) und ist durchschnittlich nur etwa 20% langsamer als die Tabularisierung. Interessant ist, dass bei dieser Problemstellung die verschiedenen Ansätze besonders schwankende Performances aufweisen. So führt,

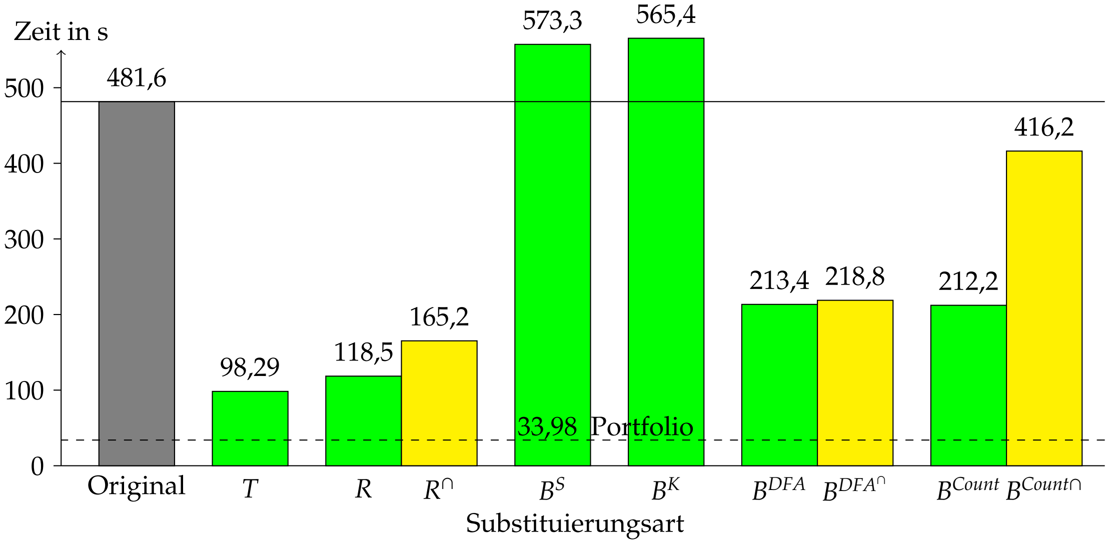


Abbildung 6.6: Die durchschnittlichen Substituierungs- und Lösungszeiten (für das Finden einer ersten Lösung) für die generierten Schichtplanungsprobleme.

bis auf die allgemeinen booleschen Skalarisierungsverfahren, jedes Verfahren mindestens einmal am schnellsten zu einer Lösung. Aus diesem Grund wurde im Diagramm in Abbildung 6.6 zusätzlich eine Linie (Portfolio) eingezeichnet, die angibt, wie viel Zeit durchschnittlich benötigt wird, wenn immer das schnellste der vorgestellten Vorgehen gewählt worden wäre. Es ist zu erkennen, dass durch die Auswahl des jeweils besten Verfahrens im Vergleich zur Tabularisierung nochmal 65,43% der Zeit eingespart werden kann. Dies unterstreicht, wie wichtig im Vorfeld ein geeigneter Klassifikator für die jeweilige Problemstellung ist. Einen solchen Klassifikator zu entwickeln ist eine der wichtigsten und erfolgversprechendsten Aufgaben für die Zukunft (in diesem Forschungsumfeld).

Im Gegensatz zu den Schichtplanungsproblemen in Abschnitt 6.2.1 führt die Anwendung der levelbasierten DFA-Vereinigung zu einer Verlangsamung (Vergleich von R mit $R^{\cap}$ und von B^{DFA} mit $B^{DFA^{\cap}}$ und B^{Count} mit $B^{Count^{\cap}}$). Dieses Verhalten ist gegebenenfalls darauf zurückzuführen, dass sich die Summe der Zustände ($15 * 51 + 2 = 767$) und Übergänge ($15 * 156 + 2 = 7958$) der einzelnen levelbasierten DFAs nicht stark genug von denen der vereinigten DFAs (587 Zustände und 4681 Übergänge) unterscheidet (Verringerung um lediglich $23,47\%$ und $41,18\%$). Zum Vergleich: bei den Schichtplanungsproblemen aus Abschnitt 6.2.1 wurde durch die levelbasierte DFA-Vereinigung die Anzahl der Zustände teilweise um bis zu $73,35\%$ und die Anzahl der Übergänge um bis zu $88,27\%$ reduziert. Diese, auf die geringere Kompaktheit der erzeugten levelbasierten DFAs zurückzuführende, Verlangsamung überträgt sich, wenn auch nicht proportional, auch auf die Lösungsgeschwindigkeiten der Substituierungen, die auf der Regularisierung aufbauen ($B^{DFA^{\cap}}$ und $B^{Count^{\cap}}$). Verlangsamt sich der Lösungsfindungsprozess bei der *Regular*-basierten booleschen Skalarisierung um lediglich 2,52%, so sind es bei der *Count*-basierten booleschen Skalarisierung erhebliche 96,11%.

War in Tabelle 6.4 und im Diagramm in Abbildung 6.6 noch von einer $79,59\%$-igen Beschleunigung durch die Tabularisierung und einer $75,4\%$-igen durch die Regularisierung auszugehen, so muss an dieser Stelle dazu gesagt werden, dass in 38 von 50 Fällen beim ursprünglichen Ansatz das Zeitlimit von 10 Minuten in die Rechnung einging, obwohl noch keine Lösung gefunden wurde. Somit ist der Wert für die durchschnittliche Lösungsdauer verfälscht und in Wahrheit deutlich höher. In Diagramm in Abbildung 6.7 ist die Anzahl der Probleme visualisiert, die durch die einzelnen Substituierungsverfahren zu einer signifikanten Beschleunigung oder Verlangsamung führen. Dies vermittelt einen realistischeren Eindruck der Effektivität der Substituierungen. Bei der Tabularisierung sind zum Beispiel lediglich fünf Probleme mindestens doppelt so zeitaufwendig wie beim ursprünglichen Verfahren. Dafür waren allerdings 36 Probleme mindestens doppelt so schnell und sogar 20 Probleme davon mindestens 100 mal so schnell lösbar wie beim ursprünglichen CSP. Auch bei den anderen Substituierungsverfahren, abgesehen von den allgemeinen booleschen Skalarisierungsverfahren (B^S und B^K) und dem booleschen *Count*-CSP mit Anwendung der levelbasierten DFA-Vereinigung, ist klar erkennbar, dass diese zu wesentlich mehr und signifikanteren Beschleunigungen als Verlangsamungen neigen.

Abschließend kann gesagt werden, dass die Regularisierung (R), aber auch die aus der Regularisierung hervorgehende *Regular*-basierte boolesche Skalarisierung (B^{DFA}), konkurrenzfähig zur Tabularisierung (T) sind. Es wurde zudem gezeigt, dass auch ohne Vorhandensein von direkten Substituierungen (für die Ablage-Constraints in CSP 16) durch Anwenden der Regularisierung und der *Regular*-basierten booleschen Skalarisierung eine deutliche Performancesteigerung im Vergleich zum ursprünglichen CSP möglich ist. Direkte Transformationen könnten allerdings, insbesondere im Falle der booleschen Skalarisierung, noch zu weiteren Beschleunigungen führen. Des Weiteren kristallisiert sich heraus, dass durch die sehr unterschiedlichen CSPs, die durch die verschiedenen Substituierungen entstehen, ein paralleler Portfolio-Ansatz sehr vielversprechend ist. Ein Portfolio-Ansatz konnte bei der angegeben Testreihe zum Black Hole-Problem eine Reduzierung der Gesamtlösungsdauer von $92,95\%$ zum ursprünglichen und von $65,43\%$ zum besten anderen Verfahren bewirken.

6.2.3 Das Knight Tour-Problem

Das Knight Tour-Problem (auf deutsch Springerproblem) ist ein Spiel für eine Person, das im Allgemeinen auf einem (8×8)-Schachbrett gespielt wird. Ziel des Spiels ist es, mit der Schachfigur Springer eine Route (Knight Tour) auf dem Schachfeld zu finden, so dass der Springer auf jedem Feld genau einmal steht. Ein Springer kann dabei von

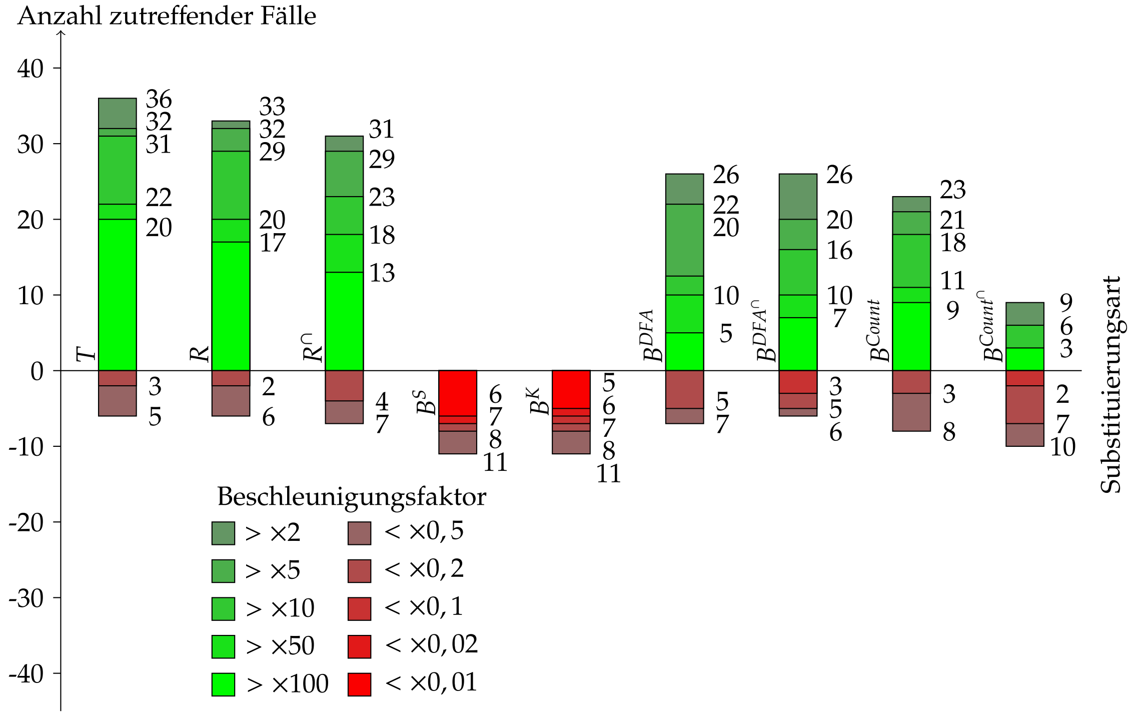


Abbildung 6.7: Eine Veranschaulichung der Anzahl an signifikanten Beschleunigungen und Verlangsamungen, die sich aus den verschiedenen Substituierungen für die generierten Black Hole-Probleme ergeben.

einer stehenden Position zur nächsten gelangen, indem er ein Feld zur Seite und zwei Felder nach vorne oder hinten, oder zwei Felder zur Seite und eines nach vorne oder hinten geht. Somit hat ein Springer, bei ausreichend viel Platz in alle Richtungen, acht Möglichkeiten sich in einem Zug zu bewegen. Das Problem kann, bei gleichbleibenden Bewegungsmöglichkeiten des Springers, auf ein $(n \times m)$-Feld verallgemeinert werden.

Das Knight Tour-Problem ist eine Spezialform des Traveling Salesman-Problems (Rundreiseproblems), welches für eine weite Bandbreite wirtschaftlicher Probleme anwendbar ist (z.B. beim Planen von Liefertouren oder Ermitteln kürzester Wege beim vollautomatischen Schweißen).

In der Vergangenheit wurde das Knight Tour-Problem bereits vielfältig untersucht und es existiert ein $O(m * n)$-Algorithmus zum Finden einer „Knight Tour" auf einem $(n \times m)$-Schachfeld [92]. Es geht an dieser Stelle nicht darum, schneller eine Lösung zu finden als es dieser oder vergleichbare Algorithmen tun. Viel mehr geht es darum zu zeigen, dass eine intuitive Modellierung des Problems als CSP mit Hilfe der in Kapitel 5 vorgestellten Substituierungsverfahren schneller eine Lösung finden kann als ohne diese Verfahren. Zum Lösen der beschriebenen $(n \times m)$-Probleme wurde das CSP 17 erstellt:

CSP 17: $P = (X, D, C)$ mit:

$$X = \{x_1, x_2, ..., x_{n*m}\} \qquad (n * m \text{ Variablen})$$

$$D = \{D_1, D_2, ..., D_{n*m} \mid D_1 = ... = D_{n*m} = \{1, 2, ..., n * m\}\} \qquad (n * m \text{ Positionen})$$

$$C = \{AllDifferent(\{x_1, x_2, ..., x_{n*m}\})\} \cup \qquad \text{(jedes Feld einmal besuchen)}$$

$$\{(dist(x_i \; mod \; m, x_{i+1} \; mod \; m, =, 1) \wedge dist(x_i \; div \; m, x_{i+1} \; div \; m, =, 2)) \vee$$
$$(dist(x_i \; mod \; m, x_{i+1} \; mod \; m, =, 2) \wedge dist(x_i \; div \; m, x_{i+1} \; div \; m, =, 1)) \mid$$
$$\forall i \in \{1, ..., n * m - 1\}\} \qquad \text{(Bewegungs-Constraint)}$$

Bei dieser Modellierung beschreibt jede Variable x_i die Position des Springers, die er nach i-Schritten eingenommen hat. Jede Variablenbelegung $\phi(x_i) = d$ für $i \in \{1, ..., n * m\}$ gibt dementsprechend an, dass das Feld an Position d an i-ter Stelle von dem Springer besucht wird. Die $(n \times m)$-Felder sind dabei mit den Werten $1, ..., n * m$ durchnummeriert, so dass das Feld an Position d dem in der (d/m)-ten Zeile und der $(d \; mod \; m)$-ten Spalte des $(n \times m)$-Feldes entspricht. Das *AllDifferent*-Constraint sorgt dafür, dass jedes Feld nur einmal besucht wird. Die Bewegungs-Constraints schränken die Bewegungen des Springers auf seine vorher beschriebenen Züge ein. Die Modellierung ist dabei sehr nah an der sprachlichen Beschreibung gehalten, so dass an dieser Stelle von einer intuitiven Modellierung gesprochen werden kann. Die *mod*-Operationen sorgen für einen Zeilenabstand von 1 bzw. 2 und die *div*-Operationen für einen Spaltenabstand von 1 bzw. 2. Vereinfacht ausgedrückt muss entweder der Zeilenabstand zweier aufeinanderfolgender Positionen gleich 1 und der Spaltenabstand gleich 2 sein oder umgekehrt. Für die folgende Testreihe wurden alle 36 Probleminstanzen $n \times m$ mit $n, m \in \{3, ..., 10\}$ gebildet. Es wurden dabei die gleichen Substituierungen wie bei dem Black Hole-Problem (Abschnitt 6.2.2) durchgeführt. Da beim *AllDifferent*-Constraint, wie in Abschnitt 6.1.3 beschrieben, alle

Knight Tour	Original	T	R	$R^{\cap}$	B^{S}	B^{K}	B^{DFA}	$B^{DFA^{\cap}}$	B^{Count}	$B^{Count^{\cap}}$
T_{Fst} in s	493	108	112	131	409	352	190	233	237	164
T_{Sub} in s	0	0,185	0,296	1,450	0,628	0,791	0,718	1,580	0,456	1,550
#*Pos*	-	31	31	30	12	15	27	23	24	28
#*Eq*	-	5	5	6	23	21	9	13	12	8
#*Neg*	-	0	0	0	1	0	0	0	0	0
#*Fail*	24	5	5	6	24	20	9	13	12	8
#*Best*[4]	0	24	1	1	0	0	4	1	1	3

Tabelle 6.5: Zeitaufwände für das Substituieren und Lösen der 36 Knight Tour Instanzen.

Substituierungen zu erheblichen Verlangsamungen beim Lösungsprozess führen, wurde auf eine Substituierung dessen verzichtet. Somit entstehen, wie zuvor bereits beim Black Hole-Problem, CSPs, die gegebenenfalls nicht direkt mit spezialisierten Solvern (*Regular*- oder pseudo-boolesche-Solver) gelöst werden können.

Bei der Regularisierung R, der *Regular*-basierten booleschen Skalarisierung B^{DFA} und dem Erzeugen von booleschen *Count*-Constraints B^{Count} wurden wieder zwei Varianten erstellt, einmal ohne und einmal mit Anwendung der levelbasierten DFA-Vereinigung. Bei Anwendung der Vereinigung wurden alle levelbasierten DFAs der Bewegungs-Constraints zu einem levelbasierten DFA vereinigt, so dass bei der Regularisierung mit levelbasierter DFA-Vereinigung ein CSP mit lediglich zwei Constraints entsteht: ein *AllDifferent*-Constraint und ein *Regular*-Constraint.

Tabelle 6.5 enthält die Daten zu der beschriebenen Testreihe. Die entsprechenden durchschnittlichen Lösungszeiten der verschiedenen Ansätze sind zusätzlich im Diagramm in Abbildung 6.8 veranschaulicht. Aus Tabelle 6.5 ist entnehmbar, dass lediglich eine Substituierung (B^{S}) einer einzelnen Probleminstanz zu einer verlangsamten Lösungsfindung führt. Alle anderen Substituierungen führen zu einer beschleunigten oder gleichbleibenden Lösungsfindung. Das spricht dafür, dass die Modellierung, egal wie intuitiv diese für den Menschen wirkt, für die Maschine nur sehr umständlich zu lösen ist. Die Tabularisierung, und dicht darauf folgend die Regularisierung, konnten das Knight Tour-Problem durchschnittlich am schnellsten lösen: mindestens 78,19% bzw. 77,28% schneller als das ursprüngliche CSP. Es gilt dabei zu beachten, dass das ursprüngliche CSP in zwei Drittel der Fälle (24 von 36) keine Lösung innerhalb des Zeitlimits von zehn Minuten gefunden hat. Für die Durchschnittsberechnung gingen an dieser Stelle jeweils zehn Minuten statt den tatsächlichen höheren Werten ein.

---

[4] Die #*Best*-Werte ergeben in Summe nur 35 statt 36, da keine der Substituierungen auf einem (9×9)-Feld innerhalb des Zeitlimits eine Lösung fand.

Etwas überraschend sind für dieses Problem sogar die allgemeinen booleschen Skalarisierungen B^S und B^K schneller als das ursprüngliche CSP. Das liegt daran, dass die einzelnen zu substituierenden Constraints eine relativ geringe Anzahl gültiger und ungültiger Tupel besitzen (der Suchraum der Bewegungs-Constraints umfasst lediglich $(n * m) * (n * m)$ viele Möglichkeiten).

Die levelbasierte DFA-Vereinigung führt bei dieser Testreihe, wie schon bei der Testreihe zum Black Hole-Problem (siehe Abschnitt 6.2.2) bei der Regularisierung $R^{\cap}$ und bei der *Regular*-basierten booleschen Skalarisierung $B^{DFA^{\cap}}$, zu einer Verlangsamung. Dies lässt sich damit begründen, dass die vereinigten levelbasierten DFAs nur geringfügig weniger Zustände und Übergänge benötigen als die nicht vereinten levelbasierten DFAs in Summe. Für ein (10×10)-Schachfeld werden zum Beispiel ohne levelbasierte DFA-Vereinigung in Summe 10.098 Zustände und 66.924 Übergänge benötigt, mit levelbasierter DFA-Vereinigung sind es immer noch 9.902 Zustände und 57.124 Übergänge. Das entspricht nur einer Verringerung von $1,94\%$ bzw. $14,64\%$. Interessanterweise ist der Effekt, der durch die levelbasierte DFA-Vereinigung auftritt, bei der Erzeugung boolescher *Count*-CSPs genau umgekehrt. Statt zu einer durchschnittlichen Verlangsamung kommt es dabei zu einer Beschleunigung von $30,80\%$. Warum es in diesem Fall zu einer Beschleunigung kommt, beim Black Hole-Problem aber zu einer drastischen Verlangsamung, ist zum jetzigen Stand nicht erklärbar.

Die Tabularisierung und die Regularisierung können beide die größte Anzahl an Instanzen lösen (jeweils 36-5=31). Die Tabularisierung stellt dabei in zwei Drittel der Fälle (24 von 36) den schnellsten Lösungsweg dar, die Regularisierung hingegen nur in einem Fall. Trotzdem sind die durchschnittlichen Lösungsgeschwindigkeiten der beiden Ansätze sehr nahe beieinander. Die hohe Anzahl an CSPs, in denen die Tabularisierung als erstes eine Lösung findet (in 66% der Fälle), lässt zunächst vermuten, dass ein Portfolio-Ansatz in diesem Fall nicht sehr gewinnbringend ist. In Abbildung 6.8 ist allerdings etwas Gegenteiliges zu erkennen. Der Portfolio-Ansatz ist dort durch eine gestrichelte Linie dargestellt und liegt noch einmal 62% unter der durchschnittlich benötigten Zeit für die Tabularisierung. Das bedeutet, dass in den wenigen Fällen, in denen die Tabularisierung nicht am schnellsten eine Lösung findet, mindestens eine der anderen Substituierungen signifikant schneller ist. Das macht einen Klassifikator, der im Vorfeld die am besten geeignete Substituierung auswählt, umso lohnenswerter. Zusätzlich zum schnelleren Finden der Lösungen sorgt der Portfolio-Ansatz dafür, dass in 35 von 36 Fällen eine Lösung gefunden werden konnte, also in vier weiteren Fällen im Vergleich zur Tabularisierung und Regularisierung.

Besonders gut ergänzen sich an dieser Stelle die Tabularisierung T und die *Regular*-basierte boolesche Skalarisierung B^{DFA}. Werden nur diese beiden Substituierungen in einem Portfolio-Ansatz verwendet, so können bereits 34 der 36 Probleminstanzen gelöst werden und das bei einer durchschnittlichen Dauer von nur 56,49s. Neben der Möglichkeit, einen Portfolio-Ansatz mit nur zwei benötigten Prozessen zu schaffen,

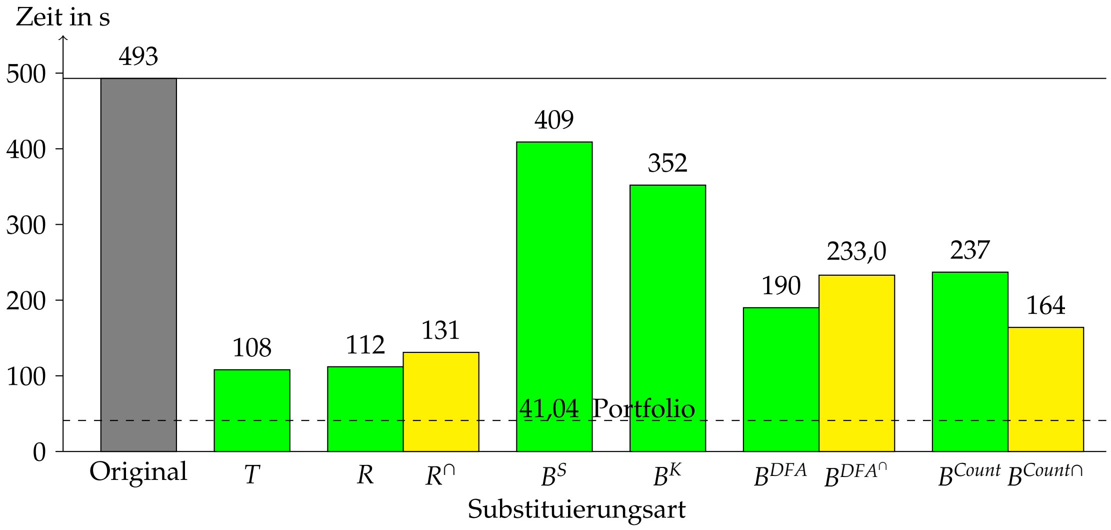


Abbildung 6.8: Die durchschnittlichen Substituierungs- und Lösungszeiten (für das Finden einer ersten Lösung) für die generierten Knight Tour-Probleme.

ergibt sich aus dieser Beobachtung auch die Möglichkeit einen Klassifikator für nur zwei (T und B^{DFA}) statt den zuvor zehn (ursprüngliches CSP plus den neun Substituierungen) betrachteten Fällen zu erzeugen. Der Vorteil eines solchen Klassifikators mit nur zwei Labels besteht darin, dass dieser wahrscheinlich eine höhere Präzision aufweist als einer mit zehn Labels. Durch eine höhere Präzision wird in mehr Fällen der beste Substituierungsansatz ausgesucht und somit die durchschnittliche Lösungsdauer gegebenenfalls mehr reduziert als bei einem ungenaueren Klassifikator mit mehreren Labels.

In Abbildung 6.9 ist die Anzahl der Fälle abgebildet, in denen die jeweilige Substituierung zu einer signifikanten Beschleunigung (um mindestens Faktor 2, 10, 100, 500 bzw. 1000) geführt hat. Es ist zu erkennen, dass alle Verfahren deutlich besser agieren als das ursprüngliche CSP. Jede Substituierung, außer den allgemeinen booleschen Skalarisierungen B^S und B^K, führte in wenigstens einem Fall mindestens 1000 mal so schnell zu einer ersten Lösung als das ursprüngliche CSP. Eine mindestens einmalige Steigerung der Lösungsgeschwindigkeit um mindestens Faktor 1000 kann auch zufällig durch eine besonders günstige Suchkonfiguration auftreten. Dass jedoch alle Substituierungen in mindestens 30% der Fälle mindestens doppelt so schnell sind wie das ursprüngliche CSP, stellt keinen Zufall dar, sondern ist eine deutliche durchschnittliche Beschleunigung. Werden die allgemeinen booleschen Skalarisierungen (B^S und B^K) außer acht gelassen, so beträgt die Anzahl der Fälle, die mindestens zu einer Beschleunigung um Faktor zwei führen, über 60%. Auf der anderen Seite gibt es über alle Substituierungen hinweg betrachtet keinen Fall, der zu einer signifikanten Verlangsamung um einen Faktor von mindestens zwei geführt hat. Damit ist eine Substituierung für das gegebene CSP 17, egal mit welchem der vorgestellten Substituierungsverfahren, in der Regel zielführend.

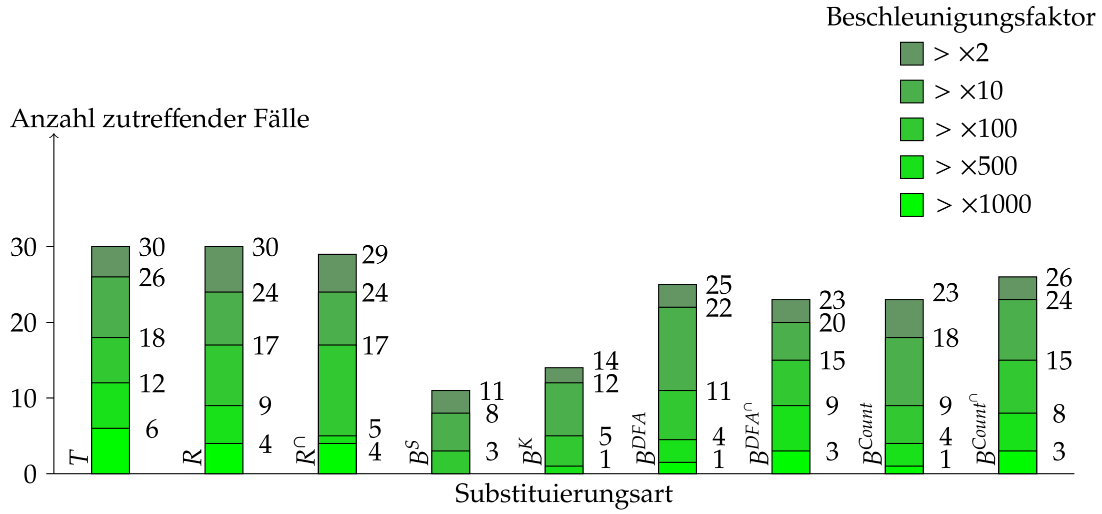


Abbildung 6.9: Anzahl signifikanter Beschleunigungen, die sich aus den verschiedenen Substituierungen für die generierten Knight Tour-Probleme ergeben.

6.2.4 Das Warehouse Location-Problem

In diesem Abschnitt wird eine Testreihe zum Warehouse Location-Problem [69] erstellt und untersucht. Das Ziel des Warehouse Location-Problems ist es die günstigste Zugehörigkeit von m Läden zu n Lagerhäusern zu finden, wobei die Kapazitäten $c_1, ..., c_n$ der Lagerhäuser gegebenenfalls beschränkt sind. Jeder Laden s_i mit $i \in \{1, ..., m\}$ muss einem offenen Lagerhaus w_j mit $j \in \{1, ..., n\}$ zugeordnet sein. Jedes Lagerhaus w_j verursacht Fixkosten f_{c_j}, wenn es geöffnet ist. Ein Lagerhaus w_j gilt als geöffnet, insofern es mindestens einen Laden s_i gibt, der diesem zugeordnet ist. Jedes geöffnete Lagerhaus w_j verursacht zusätzlich Versorgungskosten $M_{i,j}$ für jeden Laden s_i, den es versorgt.

Die Modellierung als CSOP

Bei dem Warehouse Location Problem handelt es sich um ein Constraint Satisfaction Optimization-Problem (CSOP). Das CSOP 1 gibt eine Modellierung des Warehouse Location-Problems an. Es werden dabei verschiedene Variablen mit ihren zugehörigen Domänen benötigt:

- Die Belegung einer Variablen x_i^W für $i \in \{1, ..., m\}$ mit einem Wert $d \in D_i^W = \{1, ..., n\}$ drückt aus, welchem Lagerhaus d der Laden i zugeordnet ist.
- Die Variablen $x_i^O, i \in \{1, ..., n\}$ drücken aus, ob das jeweilige Lagerhaus i geöffnet ($x_i^O = 1$) oder geschlossen ($x_i^O = 0$) ist.

- Die Belegung einer Variablen x_i^C mit $i \in \{1, ..., m\}$ mit einem Wert $d_j \in D_i^C = \{M_{i,1}, ..., M_{i,n}\}$ gibt an, welche Kosten d_j entstehen, wenn der Laden i vom Lagerhaus j versorgt wird.

  Für die Modellierung als CSOP werden alle Kosten $M_{i,j}$, die für die Versorgung eines Ladens i durch Lagerhaus j entstehen können, in einer Matrix $M^{m \times n}$ zusammengefasst. Die Liste M_i entspricht dabei der i-ten Zeile der Matrix M, beinhaltet also alle möglichen Kosten, die für einen Laden i infrage kommen.

- Die Belegung einer Hilfsvariablen y_i mit $i \in \{1, ..., n\}$ mit Wert d gibt an, dass das Lagerhaus i genau d viele Läden versorgt.

- Die Optimierungsvariable x_{costs} gibt die finalen Kosten des CSOPs an. Ziel ist es eine Belegung mit möglichst kleinen Wert für x_{costs} zu finden.

CSOP 1: $P = (X, D, C, f)$ mit:

$$
\begin{array}{rll}
X = & \{x_1^W, x_2^W, ..., x_m^W\} \cup & \text{(}m\text{ Läden)} \\
 & \{x_1^O, x_2^O, ..., x_n^O\} \cup & \text{(}n\text{ „Open"-Variablen)} \\
 & \{x_1^C, x_2^C, ..., x_m^C\} \cup & \text{(}m\text{ Kosten-Variablen)} \\
 & \{y_1, y_2, ..., y_n\} \cup & \text{(}n\text{ Hilfsvariablen)} \\
 & \{x_{costs}\} & \text{(Gesamtkosten / Optimierungsvariable)} \\
D = & \{D_1^W, D_2^W, ..., D_m^W \mid D_1 = ... = D_m = \{1, 2, ..., n\}\} \cup & \text{(}n\text{ mögliche Lager)} \\
 & \{D_1^O, D_2^O, ..., D_n^O \mid D_1 = ... = D_n = \{0, 1\}\} \cup & \text{(Lager geöffnet)} \\
 & \{D_1^C = \{M_{1,1}, M_{1,2}, ..., M_{1,n}\},\ D_2^C = \{M_{2,1}, M_{2,2}, ..., M_{2,n}\},\ ..., & \\
 & \ D_m^C = \{M_{m,1}, M_{m,2}, ..., M_{m,n}\}\} \cup & \text{(Kosten pro Laden)} \\
 & \{D(y_1), D(y_2), ..., D(y_n) \mid D(y_1) = ... = D(y_n) = \{0, 1, ..., c_i\}\} \cup & \text{(Kapazität)} \\
 & \{D(x_{costs}) = \{0, ..., f_{c_1} + ... + f_{c_n} + m * max(M_{i,j} \mid \forall\, M_{i,j} \in M)\}\} & \text{(Gesamtkosten)} \\
C = & \{Scalar(\{x_1^C, ..., x_m^C, x_1^O, ..., x_n^O\}, (1^m, f_{c_1}, f_{c_2}, ..., f_{c_n})^T, =, x_{costs})\} \cup & \\
 & & \text{(Kosten-Constraint)} \\
 & \{Element(x_i^C, x_i^W, M_i) \mid \forall i \in \{1, ..., m\} \cup & \text{(Ladenkosten-Constraints)} \\
 & \{Count(\{x_1^W, ..., x_m^W\}, y_i, i) \mid \forall i \in \{1, ..., n\} \cup & \text{(Kapazitäts-Constraints)} \\
 & \{(x_i^O = 0 \land y_i = 0) \lor (x_i^O = 1 \land y_i > 0) \mid \forall i \in \{1, ..., n\}\} & \text{(Open-Constraints)} \\
f = & min(x_{costs}) & \text{(Optimierungsfunktion)}
\end{array}
$$

Mittels der folgenden vier Constraint-Beschreibungen lässt sich das Warehouse Location-Problem modellieren:

- Das Kosten-Constraint rechnet alle entstandenen Kosten zusammen. Dabei repräsentieren die x_i^C Variablen genau die Kosten, die für die Versorgung des jeweils i-ten Ladens entstehen. Diese spiegeln die entsprechenden Kosten exakt wieder, so dass der Gewichtsvektor, mit denen die Variablen x_i^C multipliziert werden, nur Einsen enthält. Die zweite Menge an Variablen, die bei der Berechnung berücksichtigt werden muss, ist die der „Open"-Variablen. Diese sind entweder 1 (geöffnet) oder 0 (geschlossen). Ist das entsprechende Lagerhaus geschlossen, so fallen auch keine Kosten an, anderenfalls muss der Wert mit den tatsächlichen Fixkosten (f_{c_i}) multipliziert werden. Die Summe der aufgeführten Werte spiegelt dann die Gesamtkosten x_{cost} wieder.

- Die Ladenkosten-Constraints sorgen dafür, dass der jeweils i-ten Kosten-Variable x_i^C der Kostenwert $M_{i,j}$ zugeordnet wird, der durch die Belegung der i-ten Laden-Variable x_i^W mit Wert (bzw. Lagerhaus) j vorgegeben wird.
- Die Kapazitäts-Constraints sorgen dafür, dass genau y_i viele Variablen $x_1^W, ..., x_m^W$ den Wert i annehmen. Das bedeutet, es werden genau y_i viele Läden dem Lagerhaus w_i zugeordnet, wobei der maximale Wert für y_i gleich der maximalen Kapazität c_i des Lagerhauses entspricht. Liegen keine maximalen Kapazitäten vor, so wird für c_i an dessen Stelle gleich die Anzahl der Läden m angenommen.
- Die „Open"-Constraints sorgen dafür, dass die Variable x_i^O genau dann den Wert 1 annimmt, wenn das i-te Lagerhaus mindestens einen Laden unterstützt, y_i also größer oder gleich eins ist, und umgekehrt.

Substituierungen des CSOPs

Hinsichtlich der Substituierbarkeit von CSOP 1 ergeben sich die folgenden Schwierigkeiten. Für die Tabularisierung sind keine direkten Substituierungen bekannt, weswegen immer der allgemeine Ansatz gewählt werden muss. Dabei stellt die riesige Menge an gültigen Tupeln bei den Kosten- und Kapazitäts-Constraints ein Hindernis für die Tabularisierung dar. Eine Substituierung dieser Constraints erfordert jeweils das Aufzählen mehr gültiger Tupel als das gesamte CSOP Lösungen hat. Somit würde die Tabularisierung für diese Constraints nicht zu einer Beschleunigung des Lösungsverfahrens führen. Bei den Open-Constraints sieht dies anders aus. Für jedes der n-Constraints existieren genau $c_i + 1$ viele gültige Tupel. Eine Tabularisierung der *Element*-Constraints ist in CSOP 1 aufgrund der geringen Anzahl an Tupeln, die von diesem Constraint beschrieben werden, zwar möglich, führt aber zu keiner Beschleunigung, weswegen nachfolgend darauf verzichtet wurde. Für das CSOP 1 ist die Tabularisierung demzufolge nur für die Open-Constraints sinnvoll. Den CSOPs, die durch Tabularisierung der Open-Constraints entstehen, wird in der Folge der Bezeichner T zugeordnet.

Die boolesche Skalarisierung und die Regularisierung erlauben für das Problem aufgrund der vorhandenen direkten Substituierungen mehr Transformationen. Die Transformation der Ladenkosten-, Kapazitäts- und Open-Constraints ist in beiden Fällen möglich. Die Substituierung des Kosten-Constraints hingegen ist in beiden Fällen aus unterschiedlichen Gründen nicht durchführbar. Bei der Regularisierung entsteht schlicht ein zu großer levelbasierter DFA, der den Arbeitsspeicher der verwendeten Hardware übersteigt. Mit der direkten booleschen Skalarisierung des *Scalar*-Constraints (siehe Abschnitt 5.2.5) kann ein FD *Scalar*-Constraint in ein boolesches *Scalar*-Constraint überführt werden. Das Problem bei dieser Variante stellt allerdings die Repräsentation der Gesamtkosten x_{costs} durch boolesche Variablen dar. Da die Domäne von x_{costs} mehrere Millionen Werte umfasst, müssten auch allein dafür mehrere Millionen boolesche Variablen erzeugt werden, was in diesem Fall ebenfalls die gegebenen Hardware-Kapazitäten übersteigt. Bei den Laden-Constraints besteht ein weiteres Problem. Bisher wurde weder eine direkte Regularisierung noch eine direkte boolesche Skalarisierung für das *Element*-Constraint aufgeführt. Auf eine solche Angabe wird an dieser Stelle allerdings verzichtet. Da das gegebene *Element*-Constraint ohnehin nur m gültige Tupel umfasst, können dafür auch die allgemeinen Substituierungen gewählt werden.

Sowohl bei der Regularisierung als auch bei der *Regular*-basierten booleschen Skalarisierung und dem Erzeugen von booleschen *Count*-Constraints besteht die Möglichkeit levelbasierte DFA-Vereinigungen anzuwenden. Bei der Vereinigung der Automaten mindestens zweier Kosten-, Ladenkosten- oder Kapazitäts-Constraints entsteht jeweils ein levelbasierter DFA, der um ein Vielfaches mehr Knoten und Übergänge hat als die Ausgangsautomaten in der Summe. Dies kann erstens schnell zu Ressourcenengpässen und zweitens zu Verlangsamungen des Lösungsprozesses führen. Aus diesen Gründen wurde auf eine Vereinigung der zu diesen Constraints gehörenden levelbasierten DFAs verzichtet. Anders sieht es allerdings bei der Vereinigung der Automaten des jeweils *i*-ten Kapazitäts-Constraints mit dem des *i*-ten Open-Constraints aus. Eine Vereinigung dieser Automaten führt zu einem annähernd gleich großen Automaten und könnte demzufolge eine Beschleunigung nach sich ziehen.

Basierend auf den bisherigen Beschreibungen wurden die folgenden Substituierungen für das CSOP 1 erzeugt:

- $T^1 = T(Open)$: nur die Open-Constraints wurden mittels allgemeiner Tabularisierung substituiert
- $R^1 = R(Open^D)$: nur die Open-Constraints wurden mittels direkter Regularisierung substituiert
- $R^2 = R(Open^D, Kapazität^D)$: nur die Open- und die Kapazitäts-Constraints wurden mittels direkter Regularisierung substituiert
- $R^3 = R^{\cap}(Open^D, Kapazität^D)$: nur die Open- und die Kapazitäts-Constraints wurden mittels direkter Regularisierung substituiert und die Automaten des jeweils *i*-ten Open-Constraints wurden mit dem des jeweils *i*-ten Kapazitäts-Constraints vereint
- $R^4 = R(Open, Kapazität, Ladenkosten)^D$: die Open-, Kapazitäts- und Ladenkosten-Constraints wurden mittels direkter Regularisierung substituiert
- $B^1 = B(Open^S)$: nur die Open-Constraints wurden mittels Support-basierter boolescher Skalarisierung substituiert

- $B^2 = B(Open^K)$: nur die Open-Constraints wurden mittels konfliktbasierter boolescher Skalarisierung substituiert
- $B^3 = B(Open^{DFA})$: nur die Open-Constraints wurden mittels *Regular*-basierter boolescher Skalarisierung substituiert
- $B^4 = B(Open^{Count})$: nur die Open-Constraints wurden in boolesche *Count*-Constraints überführt
- $B^5 = B(Open^S, Kapazität^D)$: die Open-Constraints wurden mittels Support-basierter boolescher Skalarisierung und die Kapazitäts-Constraints mittels direkter boolescher Skalarisierung substituiert
- $B^6 = B(Open^K, Kapazität^D)$: die Open-Constraints wurden mittels konfliktbasierter boolescher Skalarisierung und die Kapazitäts-Constraints mittels direkter boolescher Skalarisierung substituiert
- $B^7 = B(Open^{DFA}, Kapazität^{DFA})$: die Open- und die Kapazitäts-Constraints wurden mittels *Regular*-basierter boolescher Skalarisierung substituiert
- $B^8 = B(Open^{Count}, Kapazität^{Count})$: die Open- und die Kapazitäts-Constraints wurden in boolesche *Count*-Constraints überführt
- $B^9 = B^{\cap}(Open^{DFA}, Kapazität^{DFA})$: die Open- und die Kapazitäts-Constraints wurden mittels *Regular*-basierter boolescher Skalarisierung substituiert, nach dem Zwischenschritt der Regulariserung wurden die Automaten des jeweils i-ten Open-Constraint mit dem des jeweils i-ten Kapazitäts-Constraints vereint
- $B^{10} = B^{\cap}(Open^{Count}, Kapazität^{Count})$: die Open- und die Kapazitäts-Constraints wurden in boolesche *Count*-Constraints überführt, nach dem Zwischenschritt der Regulariserung wurden die Automaten des jeweils i-ten Open-Constraint mit dem des jeweils i-ten Kapazitäts-Constraints vereint

Für die folgende Versuchsreihe wurden 37 Instanzen der Benchmark-Problemmengen VII, X und XIII aus [27] und das Einführungsbeispiel aus der CSPlib [69] erzeugt. Die Bezeichner der Instanzen zuzüglich der Anzahl verwendeter Lagerhäuser und Läden lässt sich der Tabelle C.1 aus Anhang C entnehmen. Da in [27] leider keine Kapazitäten für die Lagerhäuser gegeben sind, wird eine erste Testreihe ohne Kapazitäten ausgewertet (also mit $c_i = m$) und nachfolgend eine zweite mit $c_i = \lceil m/n \rceil$ geschaffen. Die beiden Testreihen unterscheiden sich dabei nur in ihrer Kapazität, alle anderen Eigenschaften, Substituierungen und Solver-Einstellungen sind gleich gesetzt worden. Die Berechnung der Kapazitäten c_i ist so gewählt, dass immer eine Lösung existiert und alle Lagerhäuser gleichmäßig ausgelastet sind. Ob dieses Szenario realistisch ist, kann an dieser Stelle nicht beurteilt werden. Durch die gleichmäßige Verteilung ergibt es sich, dass alle Lagerhäuser zwangsläufig geöffnet sind. Für die Optimierungsprobleme wurde ein Zeitlimit von zehn Minuten gesetzt.

Warehouse Location-Probleme ohne Kapazitäten

In der zweigeteilten Tabelle 6.6 sind die Ergebnisse des beschriebenen Testdurchlaufs für WLP-Probleme ohne Kapazitäten angegeben. Die Tabelle hat dabei den gleichen Aufbau wie die Tabellen aus den vorherigen Abschnitten 6.2.1 bis 6.2.3. Anders als bei den vorherigen Problemen handelt es sich bei dem WLP-Problem allerdings um ein Optimierungsproblem. Das verändert die Art und Weise, wie solche Probleme verglichen werden. In diesem Fall wurde ein durchschnittlicher Beschleunigungsfaktor a angegeben. Ist der Faktor positiv mit Wert p, so kann durch die Substituierung p-mal so schnell eine bessere oder gleichwertige Lösung gefunden werden. Ist der Faktor negativ mit Wert $-p$, so wird durch die Substituierung p-mal soviel Zeit benötigt, um eine bessere oder gleichwertige Lösung zu finden. Da das gesetzte Zeitlimit von 10 Minuten häufiger erreicht wurde, kann es sein, dass die verschiedenen Ansätze zu unterschiedlichen Lösungen mit einem unterschiedlichen Kostenwert gelangen. Um solche Verfahren miteinander vergleichen zu können, wurde von dem Verfahren, dass die schlechtere Lösung ermittelt hat, die dafür benötigte Zeit in die Berechnung einbezogen. Von dem Verfahren, dass eine bessere Lösung ermitteln konnte, wurde die Zeit für das Ermitteln der ersten Lösung, die mindestens so gut ist wie bei dem anderen Verfahren, in die Berechnung einbezogen. Konnte aufgrund des Zeitlimits von zehn Minuten ein Verfahren keine Lösung finden, so wurde dies mit zehn Minuten bewertet.

Hat der ursprüngliche Ansatz zum Beispiel nacheinander die folgenden Lösungen nach der angegebenen Zeit ermittelt: 230 in 4s, 212 in 5 min, 211 in 10 min und das andere Verfahren, mit dem verglichen werden soll, die Lösungen 220 in 2 min, 214 in 3 min, 212 in 4 min, 208 in 6 min, 205 in 8 min und 204 in 10 min, so hat das schlechtere Verfahren als besten Wert 211 in 10 Minuten ermittelt. Das zweite Verfahren hat nach 6 Minuten mit 208 den ersten besseren Wert ermittelt. Somit ergibt sich an dieser Stelle ein Beschleunigungsfaktor von $\frac{10}{6} = 1,67$. Für die andere Richtung ergibt sich ein Beschleunigungsfaktor von $-1,67$. Dadurch, dass jeweils zwei Verfahren anhand der besten Lösung des langsameren Verfahrens verglichen werden, können die Werte nicht einfach transitiv miteinander verglichen werden. Die Tabelle 6.6 enthält daher nur Beschleunigungswerte bezüglich des ursprünglichen Verfahrens und nicht zwischen zwei Substituierungsverfahren. Dafür wird auf das Diagramm in Abbildung C.1 in Anhang C verwiesen, welches alle Verfahren untereinander vergleicht. Wird zum Beispiel das ursprüngliche Verfahren mit der Regularisierung der Open-Constraints R^1 verglichen, so ist ein Beschleunigungsfaktor von $1,55$ erkennbar. Beim Vergleich des ursprünglichen Ansatzes mit dem Erzeugen und Lösen von booleschen *Count*-Constraints der Open- und Kapazitäts-Constraints unter Verwendung von levelbasierter DFA-Vereinigung B^{10} ist eine geringe Verlangsamung von $1,07$ erkennbar. Dadurch lässt sich vermuten, dass ein Vergleich von R^1 mit B^{10} eine Verlangsamung um einen Faktor von $1,5 * 1,07$, also ca. $1,7$ nach sich zieht. Tatsächlich ist es allerdings nur eine Verlangsamung von $1,11$ (siehe Tabelle C.1 Zeile R^1, Spalte B^{10}).

WLP	Original	T^1	R^1	R^2	R^3	R^4
Beschleunigungsfaktor a	-	$-1,46$	$1,55$	$-1,82$	$-1,53$	$-1,89$
T_{Sub} in s	0	0,02	0,62	2,95	0,02	3,23
#*Pos*	-	10	19	0	6	0
#*Eq*	-	13	15	15	12	12
#*Neg*	-	15	4	23	20	26
#*Fail*	12	12	12	12	12	12
#*Best*	3	8	15	0	0	0
#*Best**	20	14	26	14	17	14
#*Sol*	0-104	0-104	0-104	0-104	0-104	0-104

WLP	B^1	B^2	B^3	B^4	B^5	B^6	B^7	B^8	B^9	B^{10}
a	$-3,08$	$-3,08$	$-2,5$	$-3,08$	$-3,08$	$-2,08$	$-3,08$	$-3,08$	$-1,12$	$-1,07$
T_{Sub} in s	0,17	0,13	0,71	0,66	0,28	0,24	55,18	7,07	11,37	9,63
#*Pos*	0	0	0	0	0	12	0	0	15	15
#*Eq*	12	12	12	12	12	0	12	12	0	0
#*Neg*	26	26	26	26	26	26	26	26	23	23
#*Fail*	25	37	24	37	12	13	12	37	0	0
#*Best*	0	0	0	0	0	12	0	0	0	0
#*Best**	1	1	8	1	1	13	1	1	14	14
#*Sol*	0,81-104	0	0-74	0	0-104	0,81-134	0-104	0	0-134	0-134

Tabelle 6.6: Zeitaufwände für das Substituieren und Lösen der 38 Warehouse Location-Probleme ohne Kapazitätsgrenzen der Lagerhäuser.

Zwei zusätzliche Werte in Tabelle 6.6 sind einerseits die *#Best**-Werte, die angeben, in wie vielen Fällen eine optimale Lösung gefunden wurde. Im Vergleich dazu gibt *#Best* an, wie oft das jeweilige Verfahren am besten von allen Verfahren war. Das bedeutet, wie oft das jeweilige Verfahren als erstes eine optimale Lösung oder aber eine Lösung mit dem geringsten Kostenwert gefunden hat. Die *#Sol*-Werte geben an, welche Probleminstanzen gelöst werden konnten. Die Angabe $0, 81-134$ gibt zum Beispiel an, dass für die Instanzen 0, 81, 82, 83, 84, 91, 92, 93, 94, 101, 102, 103, 104, 111, 112, 113, 114, 121, 122, 123, 124, 131, 132, 133 und 134 mindestens eine Lösung gefunden werden konnte.

Aus Tabelle 6.6 lässt sich entnehmen, dass das ursprüngliche Verfahren bereits sehr gut ist und nur die Regularisierung der Open-Constraints (R^1) durchschnittlich zu einer Beschleunigung führt (Faktor von $1,55$). Alle anderen Verfahren, bis auf die *Regular*-basierte boolesche Skalarisierung (B^9) und das Erzeugen und Lösen boolescher *Count*-Constraints der Open- und Kapazitäts-Constraints (B^{10}), führen durchschnittlich zu einer Verlangsamung um Faktor 1,47 bis 3,08. Die Beschleunigung bei der Regularisierung der Open-Constraints (R^1) verdeutlicht eindrucksvoll, dass die Substituierung ausgewählter Constraints zu einer signifikanten Beschleunigung führen kann. An dieser Stelle wird auf den großen zeitlichen Unterschied zwischen der Regularisierung R^1 und der Tabularisierung T^1 der Open-Constraints hingewiesen, der zwar festgestellt, nicht aber begründet werden kann. Die erzeugten *Table*-Constraints in T^1 sorgen bei jedem Propagieren für die gleichen Werteausschlüsse wie beim Propagieren des äquivalenten *Regular*-Constraints in T^1 bei ansonsten identischem CSOP. Diese erheblichen zeitlichen Unterschiede lassen sich nur damit begründen, dass die möglichen unterschiedlichen Reihenfolgen der Werteausschlüsse innerhalb einer Propagation zwischen T^1 und R^1 möglicherweise im späteren Verlauf der Suche diese beeinflussen.

Werden die Regularisierungsansätze R^1, R^2, R^3 und R^4 miteinander verglichen, so fällt auf, dass R^1 mit Abstand am schnellsten ist und R^3 nichtsdestotrotz schneller als R^2 und R^4 ist. Bei genauerer Betrachtung der ursprünglichen Constraints, die substituiert werden, ist dieses Verhalten auch nicht verwunderlich. Die Substituierung der logischen Meta-Constraints (der Open-Constraints, R^1) führt zu einer Beschleunigung, da unter anderem das Konsistenzniveau der Open-Constraints damit erhöht werden kann (für eine Beschreibung der Vorteile der Regularisierung siehe Abschnitt 4.2). Die Regularisierung der *Count*-Constraints führt hingegen zu einer Verlangsamung. Dieses Verhalten bei der Substituierung von *Count*-Constraints mit großem Suchraum (n^m) konnte bereits aus der Testreihe in Abschnitt 6.1.1 abgeleitet werden. Eine levelbasierte Vereinigung der levelbasierten DFAs der ursprünglichen Open- und Kapazitäts-Constraints R^3 führt zwar zu einer Beschleunigung (um Faktor $1,55$) im Vergleich zu der nicht vereinten Variante (siehe Diagramm in Abbildung C.1 Zeile R^2 Spalte R^3), kann aber die Verlangsamung, die durch das Regularisieren der *Count*-Constraints entsteht, nicht wettmachen (siehe Diagramm in Abbildung C.1 Zeile R^1 Spalte R^3). Das *Element*-Constraint und Substituierungen dessen wurden bisher nicht speziell betrachtet, allerdings lässt sich aus Abschnitt 6.1 ableiten, dass eine Regularisierung von globalen Constraints in der Regel zu durchschnittlichen Verlangsamungen führt, so wie es hier auch der Fall ist.

Werden die Beschleunigungen bei den booleschen Skalarisierungen B^1 bis B^{10} in Tabelle 6.6 betrachtet, so wirken diese Substituierungen durchweg nachteilig, allerdings muss hier differenziert werden. Die Verfahren $B^2 = B(Open^K)$, $B^4 = B(Open^{Count})$ und $B^8 = B(Open^{Count}, Kapazität^{Count})$ führten lediglich bei dem kleinen Einführungsbeispiel „0.txt“ der CSPlib zu einer Lösung, für alle anderen Probleminstanzen konnte nach Durchführung dieser Substituierungen keine Lösung innerhalb des Zeitlimits gefunden werden. Die drei aufgeführten Verfahren sind somit für das Warehouse Location-Problem ungeeignet. Bei B^4 und B^8 werden jeweils boolesche *Count*-Constraints erzeugt, welche offenbar für diese Problemreihe besonders ungünstig sind, zumindest, wenn nach dem Zwischenschritt der Regularisierung die zugrunde liegenden levelbasierten DFAs nicht vereinigt werden. Werden diese allerdings vereinigt, wie bei B^{10}, so entstehen die nach den regularisierten und den ursprünglichen CSPs schnellsten CSPs (der durchschnittliche Verlangsamungsfaktor ist lediglich $1,07$, siehe Tabelle 6.6). Der interessante Punkt an dieser Substituierung B^{10} und der *Regular*-basierten booleschen Skalarisierung B^9 ist, dass damit für alle Warehouse Location-Probleme eine Lösung gefunden werden konnte ($\#Fail = 0$). Das macht diese Verfahren besonders dann attraktiv, wenn nach einer guten, aber nicht zwangsläufig nach einer besten Lösung gesucht wird. In immerhin 14 von 38 Fällen fanden die beiden Verfahren eine beste Lösung ($\#Best^* = 14$), allerdings waren sie dabei nie das schnellste Verfahren ($\#Best = 0$).

Die Substituierungen B^1, B^5 und B^7 finden zwar für 13 bzw. 24 der Probleme eine Lösung, benötigen dabei im Schnitt aber so viel länger, dass die Beschleunigung genauso negativ ausfällt, wie bei B^2, B^4 und B^6. Die Verfahren sind dabei niemals am schnellsten und finden jeweils auch nur zum Einführungsbeispiel eine optimale Lösung. Die beiden booleschen Skalarisierungen B^3 und B^6 liefern besonders interessante Ergebnisse. Zwar findet B^3 nur in 12 von 38 Fällen eine Lösung, allerdings sind 8 dieser 12 Lösungen am besten oder konnten am schnellsten gefunden werden im Vergleich zum ursprünglichen Problem. Am interessantesten ist die boolesche Skalarisierung B^6. Diese findet zwar auch nur zu 25 Problemen eine Lösung, allerdings findet sie in 12 dieser Fälle die beste Lösung. Hervorzuheben ist dabei, dass für die betreffenden 12 CSPs sonst nur die *Regular*-basierte boolesche Skalarisierung (B^9) und das Erzeugen und Lösen boolescher *Count*-Constraints der Open- und Kapazitäts-Constraints mit levelbasierter DFA-Vereinigung (B^{10}) ebenfalls eine Lösung finden. Das Vorgehen B^6 findet also Lösungen zu vermeintlich schweren Instanzen (Label 81-134) mit mehr Läden und Lagerhäusern, nicht aber zu den einfacheren Instanzen (Label 41-74). Das Lösen der schweren Instanzen macht dieses Vorgehen zu einer perfekten Ergänzung zu dem ursprünglichen CSP oder der Substituierung R^1, die jeweils sehr gute Lösungen für die anderen Instanzen finden. Während sich die Beschleunigungen und Verlangsamungen bei der Regularisierung sehr gut erklären und nachvollziehen lassen, sind diese bei den verschiedenen booleschen Skalarisierungen für das Warehosue Location-Problem noch völlig ungeklärt. Im folgenden Abschnitt 6.2.5 werden Schlussfolgerungen zu den einzelnen Substituierungsverfahren über alle Anwendungsbeispiele (siehe Abschnitt 6.2.1 bis 6.2.4) hinweg gezogen.

Auch wenn zu den booleschen Skalarisierungen keine fundierten Aussagen getroffen werden können, wann diese zu Beschleunigungen führen oder nicht, treten durch diese immer wieder Häufungen auf, die zu Beschleunigungen führen. Teilweise kann erkannt werden, dass bestimmte boolesche Substituierungen für gebündelte Probleminstanzen besonders geeignet sind, wie die Substituierung B^6 für schwere Warehouse Location-Probleme. Des Weiteren existieren Substituierungen, wie B^9 und B^{10}, die generell für alle Instanzen (mindestens) eine erste Lösung finden.

Der Portfolio Ansatz

Obwohl der Nutzen der booleschen Skalarisierung im Einzelnen bisher nur schwer vorhersagbar ist, lässt sich durch den Portfolio-Ansatz bereits ein erheblicher Nutzen aus dieser und den anderen Substituierungen ziehen. Um zu verdeutlichen, wie groß der Effekt von Substituierungen bei einem Portfolio-Ansatz sein kann, wurden an dieser Stelle drei Portfolio-Ansätze P^1, P^2 und P^3 durchgeführt. Bei der ersten Variante P^1 wurden alle 16, durch Transformationen erzeugte, CSPs parallel, ohne Austausch von Werten, gelöst. Bei der zweiten Variante P^2 wurden die vier erfolgversprechendsten Substituierungen ausgewählt und ohne Austausch von Werten unabhängig voneinander betrachtet. Für die dritte Variante P^3 wurden ebenfalls die vier erfolgversprechendsten Substituierungen ausgewählt und deren Zielfunktionswert während der Lösungssuche miteinander geteilt. Im Falle von P^2 und P^3 wurden nur 4 Varianten betrachtet, da die genutzte Hardware (siehe Abschnitt 1.4) über vier Prozessoren verfügt und so die unterschiedlichen CSPs echt parallel gelöst werden können. Am erfolgversprechendsten wurden im Vorfeld die folgenden vier Substituierungen eingeschätzt: *Original*, R^1, B^6 und B^{10}. Diese weisen möglichst große Unterschiede in der Modellierung auf (einmal der ursprüngliche Ansatz, einmal regularisiert und zwei unterschiedliche boolesche Skalarisierungen), weswegen vermutet wird, dass die Lösungsfindung in diesen Verfahren sehr unterschiedlich abläuft und somit gegebenenfalls sehr stark davon profitiert werden kann, dass die verschiedenen Substituierungen verschiedene Bereiche des Suchraums abdecken (P^1, P^2 und P^3) bzw. der jeweils beste Zielfunktionswert untereinander ausgetauscht wird (P^3).

Die verschiedenen Portfolio-Ansätze führten noch einmal zu einer deutlichen Beschleunigung im Vergleich zum ursprünglichen Ansatz. Das unabhängige, parallele Lösen P^1 erzielte bereits einen Beschleunigungsfaktor von 16,04. Für die echt parallele Durchführung dieses Ansatzes werden allerdings auch 16 Prozessoren benötigt, was schlussendlich in einem linearen Speedup mündet (Beschleunigung ca. gleich der Anzahl der Prozesse $16,04 \approx 16$). Werden statt allen 16 Substituierungen nur die vier benannten (*Original*, R^1, B^6 und B^{10}) verwendet (P^2), so steigt der Speedup im Verhältnis zur Anzahl der Prozessoren ca. um das vierfache ($15,71/4 \approx 4$). Was einem superlinearen Speedup entspricht ($15,71 > 4$). Werden nur diese vier Substituierungen verwendet und der Zielfunktionswert zwischen den CSPs ausgetauscht (P^3), so kommt es interessanterweise nur zu einer Beschleunigung um einen Faktor von 6,20. Was immer noch einem superlinearen Speedup ($6,2 > 4$) entspricht, aber geringer ist als bei P^2. Die Verlangsamung von P^2 zu P^3 ist überraschend, lässt sich aber wie folgt begründen:

1. Der Austausch von Informationen in parallelen Systemen sorgt in der Regel für Verlangsamungen, weil für den Moment, indem die Werte angepasst werden, diese nicht für andere Berechnungen genutzt werden können. Es entsteht dadurch ein aktives Warten. Dieser Effekt sollte an dieser Stelle allerdings nicht so stark zum Tragen kommen, da lediglich Informationen über eine Ganzzahlvariable ausgetauscht werden.

2. Eine Veränderung der Domänen von Variablen kann bei dynamischer Variablen- und Werteauswahlheuristik zu einer veränderten Auswahlreihenfolge führen. Dadurch kann es zu massiven Änderungen innerhalb des Suchbaums kommen, was die benötigte Dauer für das Finden einer ersten Lösung signifikant beeinflussen kann (siehe Abschnitt 2.4.1).

Warehouse Location-Probleme mit Kapazitäten

Für diese Testreihe wurden die gleichen Probleme, wie im Abschnitt zuvor beschrieben, verwendet. Es wurden lediglich die Kapazitäten $c_i = \lceil m/n \rceil$ für jedes Lagerhaus w_i in jeder Probleminstanz mit m Läden und n Lagerhäusern hinzugefügt. In Tabelle 6.7 sind die Ergebnisse der Testreihe zusammengefasst. Die Tabelle hat dabei den identischen Aufbau wie die Tabelle 6.6 zur Testreihe ohne Kapazitätsgrenzen. Auffällig ist, dass bei der Testreihe ohne Kapazitäten durchschnittlich zu 21 der 38 Probleme (innerhalb von 10 Minuten) eine Lösung gefunden werden konnte, wohingegen das bei der Testreihe mit Kapazitäten nur noch zu durchschnittlich knapp 12 der 38 Probleme der Fall war (obwohl in beiden Fällen alle WLP-Probleme Lösungen haben). Die Testreihe mit Kapazitäten scheint somit deutlich schwerer zu lösen zu sein, als die ohne. Ein signifikanter Unterschied zwischen beiden Testreihen ist, dass der ursprüngliche Ansatz bei der Variante ohne Kapazitäten vergleichsweise gut abschneidet (zweitbester gemessen am Beschleunigungsfaktor), dieser aber bei der Variante mit Kapazitäten nahezu unbrauchbar ist. Lediglich zum sehr kleinen Einführungsbeispiel der CSPlib kann mit dem ursprünglichen Verfahren eine Lösung gefunden werden. In allen anderen Fällen kann innerhalb des Zeitlimits von zehn Minuten kein Ergebnis gefunden werden. Gleiches gilt für die vier booleschen Skalarisierungen B^1 bis B^4.

Analog zum vorherigen Abschnitt wurde neben der Tabelle 6.7 auch das Diagramm in Abbildung C.2 zum vollständigen direkten Vergleich aller einzelner Varianten erstellt. In diesem lässt sich erkennen, dass es keinen Ansatz gibt, der hier immer am schnellsten agiert (es existiert keine Spalte mit nur positiven Werten). Zusammen betrachtet mit der Tabelle 6.7 lässt sich aber erkennen, dass vor allem die boolesche Skalarisierung

WLP	Original	T^1	R^1	R^2	R^3	R^4
Beschleunigungsfaktor a	-	$1,48$	$1,48$	$1,48$	$1,48$	$1,48$
T_{Sub} in s	0	0,01	0,11	0,43	0,02	0,61
#*Pos*	-	12	12	12	12	12
#*Eq*	-	25	25	25	25	25
#*Neg*	-	1	1	1	1	1
#*Fail*	37	25	25	25	25	25
#*Best*	0	2	10	0	0	0
#*Best**	1	13	13	13	13	13
#*Sol*	0	0, 111-134	0, 111-134	0, 111-134	0, 111-134	0, 111-134

WLP	B^1	B^2	B^3	B^4	B^5	B^6	B^7	B^8	B^9	B^{10}
a	1	1	1	1	$1,54$	$1,54$	$2,19$	$1,96$	$1,32$	$3,00$
T_{Sub} in s	0,05	0,04	0,11	0,17	0,11	0,10	3,85	1,13	2,32	0,78
#*Pos*	1	0	0	1	14	14	26	25	14	26
#*Eq*	37	37	38	37	24	24	12	12	24	12
#*Neg*	0	1	0	0	0	0	0	1	0	0
#*Fail*	37	37	37	37	24	24	12	12	24	12
#*Best*	0	0	0	0	4	3	0	7	0	12
#*Best**	1	1	1	1	4	4	1	8	1	13
#*Sol*	0	0	0	0	0-74	0-74	0-74, 111-134	0-74, 111-134	0-74	0-104

Tabelle 6.7: Zeitaufwände für das Substituieren und Lösen der 38 Warehouse Location-Probleme mit Kapazitätsgrenzen der Lagerhäuser.

B^{10}, gefolgt von B^7 und B^8, besonders effektiv ist. Diese finden die meisten Lösungen (scheitern am wenigsten: #*Fail* = 12) und haben den besten durchschnittlichen Beschleunigungsfaktor ($a = 3,00$ bzw. $2,19$ bzw. $1,96$). Auch in dieser Testreihe konnte zu jeder Probleminstanz mindestens ein Vorgehen eine Lösung finden. Der boolesche Skalarisierungsansatz B^{10}, der auch schon bei der Testreihe ohne Kapazitäten gut abschnitt (dritter Platz gemäß Beschleunigungsfaktor), schneidet diesmal im Vergleich zum ursprünglichen Ansatz am besten ab (Beschleunigungsfaktor 3,00) und findet von allen Ansätzen die meisten Lösungen am schnellsten (#*Best* = 12). In dieser Kategorie folgen die Regularisierung R^1 mit 10 und die booleschen Skalarisierungen B^8, B^5 und B^6 mit jeweils 7, 4 bzw. 3 schnellsten Lösungen.

Der Portfolio-Ansatz

Analog zu den Portfolio-Ansätzen in der Testreihe ohne Kapazitäten wurden diese für die Testreihe mit Kapazitäten genauso angewendet. Abgesehen von dem ursprünglichen Verfahren, das hier im Portfolio jeweils erhalten sein soll, sind die Ansätze R^1, B^{10}, B^7 und B^8 am schnellsten. Es wurde an dieser Stelle allerdings darauf verzichtet, B^7 oder B^8 ins Portfolio aufzunehmen, da dann mit R^1, B^{10} und B^7 oder B^8 drei Verfahren enthalten wären, die eine Regularisierung beinhalten. An dieser Stelle wurde es als wichtiger bewertet möglichst unterschiedliche Ansätze zu verwenden. Aus diesem Grund wurde sich für B^6 entschieden, welches das beste Verfahren ist, das keine Regularisierung enthält.

Der Ansatz P^1 führt wieder alle Substituierungen parallel aus, ohne dass Informationen zwischen den parallel gelösten CSPs ausgetauscht werden. Der Ansatz P^2 führt die vier Substituierungen *Original*, R^1, B^6 und B^{10} ohne Austausch von Werten parallel aus und der Ansatz P^3 führt die vier Substituierungen ebenfalls parallel aus, wobei der jeweils beste Zielfunktionswert den anderen Verfahren mitgeteilt wird. Es ergeben sich dabei noch signifikantere Beschleunigungen als bei der Testreihe ohne Kapazitäten. Der Ansatz P^1 erreicht dabei einen Beschleunigungsfaktor von $407,71$. Der Ansatz P^2 ist nur unwesentlich langsamer mit einem Beschleunigungsfaktor von $405,20$. Lediglich der dritte Ansatz P^3 fällt mit einem Beschleunigungsfaktor von $297,03$ etwas ab. Die Verlangsamung von P^2 zu P^3 lässt sich wie bei der Testreihe ohne Kapazitäten erklären. In allen drei Fällen konnte somit ein deutlicher superlinearer Speedup erreicht werden.

6.2.5 Fazit zur Anwendung der Substituierungen auf realistischen Anwendungsbeispielen

In diesem Abschnitt werden die gewonnenen Erkenntnisse aus den verschiedenen Anwendungsbeispielen noch einmal zusammengefasst, es werden weiterführende Betrachtungen zur Regularisierung beleuchtet und anschließend ein Fazit zu den neu entwickelten Substituierungen gezogen. Es ist zu beobachten, dass aufgrund von Ressourcenengpässen nicht alle Substituierungen für alle CSPs anwendbar sind.

Hinzu kommt, dass auch wenn die entsprechenden Substituierungen durchgeführt werden können, diese nicht immer zu einer Beschleunigung führen. Nachfolgend werden die einzelnen Substituierungen (Tabularisierung, Regulariserung und boolesche Skalarisierung) differenziert betrachtet und analysiert, wann die jeweilige Substituierung zu einer Zeitersparnis bei der Suche nach einer ersten Lösung führt.

Die Tabularisierung

Die Tabularisierung, wie sie in [10] vorgestellt und in dieser Arbeit verwendet wird, eignet sich in der Regel nur für die Substituierung von Constraints oder Constraint-Mengen, die nur wenige gültige (oder ungültige) Tupel besitzen. In [10] werden diesbezüglich 10.000 Tupel angegeben, die tatsächliche Grenze kann aber von Problem zu Problem stark variieren. Unabhängig von der konkreten Tupelobergrenze fällt diese allerdings zu gering aus, als dass die Tabularisierung in realistischen CSPs als Substituierung für die meisten globalen Constraints infrage kommt. Bei den gegebenen Anwendungsbeispielen ließ sich die Tabularisierung somit auch nur für solche Constraints anwenden, die wenige gültige Tupel hatten. In diesen Fällen waren das ausnahmslos logische Meta-Constraints: die Ablage- und die Fächer-Ordnungs-Constraints beim Black Hole Problem (Abschnitt 6.2.2), die Bewegungs-Constraints beim Knight Tour-Problem (Abschnitt 6.2.3) und die Open-Constraints beim Warehouse Location-Problem (Abschnitt 6.2.4). Wenn die Tabularisierung anwendbar ist, führt sie zu sehr guten Ergebnissen. So war sie die durchschnittlich schnellste Substituierung beim Black Hole- und beim Knight Tour-Problem.

Die Regularisierung

Im Vergleich zur Tabularisierung bietet die Regularisierung neben der allgemeinen Transformation (siehe Abschnitt 5.1.1) auch die Möglichkeit der direkten Transformation von ausgewählten Constraints in *Regular*-Constraints (siehe Abschnitt 5.2). Diese direkten Transformationen erlauben den Einsatz der Regularisierung für globale Constraints (siehe Abschnitte 6.2.1 bis 6.2.4). Somit ergibt sich bei der Regularisierung ein viel größerer Anwendungsbereich als bei der Tabularisierung.

Bei den hier dargestellten Anwendungsfällen ist die Propagation der einzelnen Constraints bei der Tabularisierung, wenn diese eingesetzt werden kann, in der Regel geringfügig schneller als bei der Regularisierung. Aus der langsameren Propagation des *Regular*-Constraints folgt allerdings in der Regel auch nur ein geringer zeitlicher Mehraufwand für das Finden einer ersten Lösung ($17,1\%$ bei den Black Hole-Problemen, $3,6\%$ bei den Knight Tour-Problemen, 0% beim Warehouse Location-Problem mit Kapazitäten). Lediglich bei dem Warehouse Location-Problem ohne Kapazitäten ist die Regularisierung im Durchschnitt mit $47,1\%$ signifikant schneller als die Tabularisierung. Allerdings wurde bereits in Abschnitt 6.2.4 erläutert, dass dieses Verhalten untypisch und bisher nicht erklärbar ist. Zusammenfassend lässt sich sagen, dass sich die Tabularisierung und die Regularisierung, in Bezug auf das Finden einer ersten Lösung für die vorgestellten Anwendungsbeispiele, in der gleichen Größenordnung bewegen.

Die Transformationsdauer für die Tabularisierung und die Regularisierung wurde bei den einzelnen Anwendungsbeispielen jeweils mit aufgeführt, machte aber nur einen unwesentlichen Anteil an der Gesamtlösungsdauer der CSPs aus. Das liegt bei den Problemen in den Abschnitten 6.2.2 bis 6.2.4 hauptsächlich daran, dass die Anzahl der gültigen Tupel der einzelnen Constraints mit maximal 1326 sehr gering ist (siehe Tabelle 6.8). Dementsprechend sind die benötigten Datenstrukturen für die Tabularisierung (die Supports) und die Regularisierung (die levelbasierten DFAs) ebenfalls sehr klein und können schnell erzeugt werden. Bei den Schichtplanungsproblemen in Abschnitt 6.2.1 ist dies teilweise anders. Das Constraint $A8$, welches für zwei aufeinanderfolgende Wochen fordert, dass mindestens zwei Tagen davon eine Freischicht zugeordnet ist, hat für $s = 3$ Schichten und $d = 7$ Tage pro Woche zum Beispiel knapp 193 Millionen gültige Tupel. Das lässt sich mit einem *Table*-Constraint nicht kompakt repräsentieren. Auch der allgemeine Regularisierungsansatz würde dabei schnell die technischen Grenzen der Hardware erreichen. Allerdings kann mittels der direkten Substituierungen aus Abschnitt 5.2 die Regularisierung mit geringem Zeitaufwand trotzdem durchgeführt werden. Dies unterstreicht auf der einen Seite die Bedeutung direkter Transformationen und auf der anderen Seite veranschaulicht es, dass die Regularisierung häufiger eingesetzt werden kann als die Tabularisierung.

Eine hohe Tupelanzahl sorgt, neben dem viel größeren Aufwand für die Substituierung eines Constraints, auch dafür, dass sich die Lösungsgeschwindigkeit eines tabularisierten CSPs gegenüber dem ursprünglichen oder einem kompakten regularisierten CSP drastisch erhöht. Es stellt sich die Frage, ob es neben der Anzahl gültiger Tupel, die ein Constraint repräsentiert, noch weitere Einflussfaktoren gibt, die eine Tabularisierung oder Regularisierung begünstigen. Diese Problemstellung wurde in der in Zusammenhang mit dieser Arbeit entstanden Publikation [94] diskutiert und wird in Abschnitt 6.2.5 (Weitergehende Betrachtungen zur Regularisierung I) erläutert.

Die Regularisierung mit levelbasierter DFA-Vereinigung

Eine weitere Option, die die Regularisierung einer Menge von Constraints ermöglicht, ist durch die levelbasierte DFA-Vereinigung gegeben. Dabei werden die durch Substituierung der ursprünglichen Constraints entstehenden und über *sharedVariables* verfügenden levelbasierten DFAs mittels levelbasierter DFA-Vereinigung zu einem einzelnen DFA vereint. Dadurch ist es möglich, die ursprüngliche Constraint-Menge C durch ein *Regular*-Constraint zu substituieren, was zu den in 4.2 aufgeführten Vorteilen führen kann. Wie in den Abschnitten 6.2.1 bis 6.2.4 erkennbar ist, führt die DFA-Vereinigung allerdings nicht in allen Fällen zu Beschleunigungen. Aus diesem Grund ist es notwendig vorher eine Abschätzung zu treffen, wann sie potentiell beschleunigend ist und wann nicht. Zu diesem Zweck werden im Folgenden drei Kenngrößen benannt und miteinander verglichen. Berechnungsgrundlage für diese Kenngrößen sind die Automatenmenge $M = \{M_1, ..., M_n\}$, der zugehörigen *Regular*-Constraints $C^R = \{c_1^R, ..., c_n^R\}$, und der durch levelbasierte DFA-Vereinigung entstehende Automat $M^\cap$. Mit $M_i.Q$ bzw. $M^\cap.Q$ sei die Zustandsmenge Q des DFAs M_i bzw. des vereinten DFAs $M^\cap$ und mit $M_i.\delta$ bzw. $M^\cap.\delta$ die Menge der Übergänge von M_i bzw. $M^\cap$ bezeichnet.

1. Die prozentuale Reduktion der benötigten Zustände

$$R^s = 100 - 100 * |M^\cap.Q| / \sum_{M_i \in M} |M_i.Q| \tag{6.1}$$

2. Die prozentuale Reduktion der benötigten Übergänge

$$R^t = 100 - 100 * |M^\cap.\delta| / \sum_{M_i \in M} |M_i.\delta| \tag{6.2}$$

3. Die prozentuale Reduktion der Variablenüberlagerungen:

$$R^v = 100 - 100 * |X^\cap| \; / \sum_{c_i \in C} |scope(c_i)| \tag{6.3}$$

Bei den prozentualen Reduktionen wird ermittelt, um wie viel Prozent der vereinigte DFA $M^\cap$ weniger Zustände, Übergänge bzw. beteiligte Variablen hat als die Ausgangs-DFAs $M_1, ..., M_n$, welche die ursprünglichen Constraints C beschreiben, in Summe. Die prozentuale Reduktion der Variablenüberlagerungen gibt an, wie stark die Vernetzung im Constraint-Netz abnimmt. Es ist ein Maß dafür, wie viele Überlappungen der Constraints vermieden werden können, also *sharedVariables* reduziert werden können.

Das Diagramm in Abbildung 6.10 stellt die prozentualen Reduktionen der Anzahl an Zuständen R^s, Übergängen R^t und Variablenüberlagerungen R^v sowie deren Durchschnitt für die verschiedenen Anwendungsbeispiele aus den Abschnitten 6.2.1 bis 6.2.4 dar. Jeder Strich, in einer der vier Farben, repräsentiert eine Reduktion (der Zustände, Übergänge, *sharedVariables* oder des Durchschnitts dieser drei) einer Anwendungsinstanz[5]. Es ist zu erkennen, dass die Reduktion der Zustände R^s und Übergänge R^t bei den Schichtplanungsproblemen und den Warehouse Location-Problemen ohne Kapazitäten jeweils deutlich höher ist als bei den anderen Problemen (jeweils mindestens 57% gegenüber sonst höchstens 42%). Die Reduktion bei den Variablenüberlagerungen R^v ist nur bei den Schichtplanungsproblemen sehr hoch (über 80% im Vergleich zu sonst maximal 50%). Generell gilt, je größer die Reduktionswerte sind, desto vielversprechender ist

---

[5] Besonders beim Black Hole- und beim WLP-Problem kommt es zu Reduktionen, die, bis auf das Einführungsbeispiel beim WLP, einen sehr ähnlichen Wert aufweisen. Bei den WLP-Problemen ohne und mit Kapazitäten stellen die obersten Striche (außer bei R^v von WLP ohne Kapazitäten der unterste Strich) jeweils einen einzelnen Ausreißer dar, der das kleine Einführungsbeispiel aus der CSPlib repräsentiert. Alle anderen Werte sind durch die anderen Striche dargestellt.

die Anwendung der levelbasierten DFA-Vereinigung. Wird der Durchschnitt der verschiedenen Reduktionswerte ermittelt (die roten Markierungen), so ist erkennbar, dass dieser bei der Schichtplanung mit $70,1\%$ bis $86,4\%$ deutlich höher liegt als bei den WLP-Problemen ohne Kapazitäten ($50,7$ bis $62,8\%$) und den anderen Problemen (maximal $30,1\%$). Zu beobachten ist, dass bei durchschnittlichen Reduktionswerten $(R^s + R^t + R^v)/3$ von über 50% eine durchschnittliche Beschleunigung (Schichtplanungsprobleme und WLP ohne Kapazitäten) und sonst keine Veränderung (WLP mit Kapazitäten) bzw. eine Verlangsamung (Black Hole- und Knight Tour-Problem) auftritt. Anhand der gegebenen Datenmenge lässt sich also ein Grenzwert festlegen (zum Beispiel von 50%), ab dem die levelbasierte DFA-Vereinigung in den Anwendungsbeispielen im Durchschnitt zu Beschleunigungen führt. Dieser Grenzwert ist nicht für alle Probleme korrekt, kann aber ein erstes Indiz sein. Nicht für alle Schichtplanungsprobleme (siehe Abschnitt 6.2.1) und Warehouse Location-Probleme ohne Kapazitäten (siehe Abschnitt 6.2.4) führt die levelbasierte DFA-Vereinigung zu einer Beschleunigung und nicht für alle anderen Anwendungsbeispiele aus den Abschnitten 6.2.2 bis 6.2.4 trat eine Verlangsamung auf. Durchschnittlich betrachtet ist die Vereinigung bei den Schichtplanungsproblemen und den Warehouse Location-Problemen ohne Kapazitäten aber empfehlenswert und bei den anderen hier vorgestellten Problemen nicht. Die angesprochenen Unterschiede innerhalb einer Anwendungsklasse (Schichtplanung, Black Hole, Knight Tour, WLP) verdeutlichen, dass eine präzisere Abschätzung als die hier angegebene 50%-Regel für weitere Anwendungsbeispiele sinnvoll ist. Es empfiehlt sich zum Beispiel ein maschinelles Lernverfahren, ähnlich dem zur Klassifikation der Regularisierung und Tabularisierung (Vergleich mit [94] und den nachfolgenden Abschnitten), um die verschiedenen Reduktionen und möglicherweise weitere Parameter gezielter in Verbindung zu setzen.

Weitergehende Betrachtungen zur Regularisierung I: Ein auf maschinellem Lernen basierender Klassifikator für das *Table-Regular*-Problem

Im Zusammenhang mit dieser Arbeit wurden erste Untersuchungen dazu durchgeführt und in [94] veröffentlicht, wann welche Constraint-Substituierung (*Table*- oder *Regular*-Constraint) schneller zu einer Lösung führt. Dabei wurde ein maschineller Lernansatz entwickelt, der nach Eingabe allgemeiner (Anzahl beteiligter Variablen, Größe der Domänen, etc.) und Constraint-spezifischer Parameter (für *Table*: Anzahl an 1-en und 0-en in den zugehörigen Supports und für *Regular*: Anzahl an Zuständen und Übergängen im levelbasierten DFA, die Breite des DFAs oder dessen Verzweigungsgrad) prognostizieren kann, ob eine Substitution durch ein *Table*- oder ein *Regular*-Constraint zur schnelleren Lösungsfindung beiträgt. Dabei wurden ähnlich zu Abschnitt 6.1 allerdings nur CSPs mit exakt einem *Table*- oder *Regular*-Constraint betrachtet. Für dieses Szenario konnte mit einer Präzision von über 98% und einem Recall-Wert von über 91% der relevanten Probleme die richtige Vorhersage getroffen werden[6].

---

[6] Als relevant galten dabei nur Constraints, bei denen die *Regular*-Form und die *Table*-Form eine zeitliche Unterscheidung um mindestens Faktor 1,5 aufwiesen.

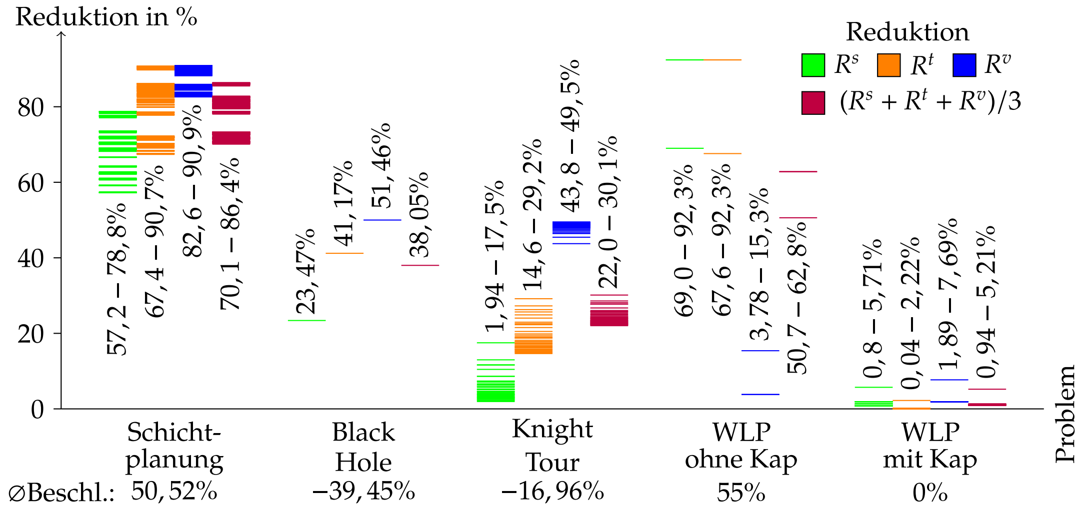


Abbildung 6.10: Die prozentualen Reduktionen der Anzahl an Zuständen, Übergängen und Variablenüberlagerungen für die verschiedenen Anwendungsbeispiele aus den Abschnitten 6.2.1 bis 6.2.4.

Es wird davon ausgegangen, dass der Ansatz des maschinellen Lernens weiterverfolgt werden kann, um diesen erstens für beliebige CSPs und zweitens für weitere Constraints, in die umgewandelt werden soll (zum Beispiel boolesche *Scalar*-Constraints), anwendbar zu gestalten. In [94] wurde ein Zufallswaldklassifikator (Random Forests Classificator) mittels der Python Bibliothek scikit-learn [121] erstellt und verwendet. Mit diesem Klassifikator konnten die Parameter ermittelt werden, die den größten Einfluss auf die Klassifikation haben. In diesem Fall sind das die berechneten *Regular*-spezifischen Parameter *CiT* (compactness in transitions, Gleichung 6.4) und *CiW* (compactness in width, Gleichung 6.5). Beide Parameter sind ein Maß für die Kompaktheit eines levelbasierten DFAs. Bei der Kompaktheit in den Übergängen *CiT* wird die Anzahl gültiger Tupel $|T|$, die der DFA beschreibt, mit der Anzahl seiner Übergänge $|M.\delta|$ ins Verhältnis gesetzt. Bei der Kompaktheit in der Breite wird die Anzahl der gültigen Tupel $|T|$ mit der Breite des DFAs $width(M)$ ins Verhältnis gesetzt. Die Breite eines levelbasierten DFAs entspricht dabei der maximalen Anzahl an Zuständen, die der DFA in einem Level i enthält ($max(|Q_i|), \forall i \in \{0, ..., n\}$).

1. Kompaktheit bezüglich der Übergänge (compactness in transitions)

$$CiT = |T|/|M.\delta|, \tag{6.4}$$

2. Kompaktheit bezüglich der Breite (compactness in width)

$$CiW = |T|/width(M). \tag{6.5}$$

Problem	Constraints	CiT	CiW	$\lvert T\rvert$	Prognose
Black Hole	Fächer-Ordnung	$0,96$	$25,98 - 26,5$	1326	*Table*-Constraint
Black Hole	Ablage-Ordnung	$2,66$	$31,92$	415	*Table*-Constraint
Knight Tour	Bewegungs-Constraint	$0,67 - 0,85$	$2 - 5,76$	$16 - 576$	*Table*-Constraint
WLP	Open-Constraint	$0,6 - 0,96$	$1,5 - 25,5$	$3 - 51$	*Table*-Constraint

Tabelle 6.8: Übersicht der verschiedenen Parameter der zu substituierenden Constraints zur Einschätzung, ob eine Tabularisierung oder eine Regularisierung zielführender ist.

Für beide Kompaktheitsmaße gilt, je größer deren Werte sind, desto besser ist das *Regular*-Constraint zur Beschreibung des ursprünglichen Constraints geeignet. Entsprechend ist das *Table*-Constraint umso besser geeignet, je kleiner die Werte der Kompaktheitsmaße sind. Ein weiteres Kriterium ist die Anzahl der gültigen Tupel T, die durch das jeweilige Constraint repräsentiert werden. Ist diese sehr gering, so ist das *Table*-Constraint zu bevorzugen. In Tabelle 6.8 sind die Werte für die drei Parameter Kompaktheit in den Übergängen CiT, Kompaktheit in der Breite CiW und Anzahl der gültigen Tupel T für die in den Abschnitten 6.2.1 bis 6.2.4 zu tabularisierenden und regularisierenden Constraints aufgeführt. In allen Fällen ist die Anzahl der gültigen Tupel (unter 1500) und der jeweilige CiT Wert (unter 3) sehr gering. Lediglich die CiW-Werte für das Black Hole- und Teile des Warehouse Location-Problems sind hoch genug um eine Regularisierung in Betracht zu ziehen. Es gilt dabei aber zu beachten, dass es keine festen, unabhängigen Grenzwerte für die einzelnen Parameter gibt, sondern diese im Random Forrest-Klassifikator voneinander abhängig sind. Deswegen kann für die Werte in Tabelle 6.8 kein fester Wert für CiT, CiW und T angegeben werden, ab wann sie regularisiert oder tabularisiert werden sollen. Der Klassifikator aus [94] empfiehlt für die angegebenen Fälle jeweils die Tabularisierung.

In den Abschnitten 6.2.1 bis 6.2.4 ist, bis auf beim Warehouse Location-Problem, die Tabularisierung schneller als die Regulariserung (Vergleich von T_{Fst} von T und R beim Black Hole- und Knight Tour-Problem bzw. T^1 und R^1 vom Warehouse Location-Problem). Das entspricht, bis auf beim Warehouse Location Problem, der Prognose des vorgestellten Klassifikators. Das Warehouse Location-Problem stellt dabei, wie in Abschnitt 6.2.4 erläutert, eine Ausnahme dar. Auch bei diesem ist das Propagieren des *Table*-Constraints schneller als das des *Regular*-Constraints, allerdings differieren die Suchen (die Variablenauswahlheuristiken wählen unterschiedliche Variablen) aus einem bisher nicht bekannten Grund ab einem fortgeschrittenen Punkt, wodurch die Regularisierung schneller zu einer ersten Lösung führt.

Weitergehende Betrachtungen zur Regularisierung II: Ein Meta-CSOP zum Erkennen von Teil-CSPs, die besonders für die levelbasierte DFA-Vereinigung geeignet sind

Im Zusammenhang mit dieser Arbeit entstand die Veröffentlichung [97], in der beschrieben wird, wie zu einem CSP ein für die Regularisierung geeignetes Teil-CSP ermittelt werden kann. Zu einem bestehenden CSP $P = (X, D, C)$ mit $X = \{x_1, ..., x_n\}$, $D = \{D_1, ..., D_n\}$ und $C = \{c_1, ..., c_m\}$ wird dafür ein Meta-CSOP $P_{opt} = (X', D', C', f)$ mit $X' = \{x'_j \mid \forall j \in \{1, ..., m\}\} \cup x_{opt}$, $D = \{D_j = \{0, ..., k\} \mid \forall j \in \{1, ..., m\}\} \cup D_{opt} = \{0, ..., up\}$, $C = \{cspOverlapSplit(...)\}$ und $f = max(x_{opt})$ erzeugt. Das neu erzeugte CSOP besitzt eine Variable $x'_j \in X'$ für jedes ursprüngliche Constraint $c_j \in C$ mit der Domäne $D_j = \{0, ..., k\}$, wobei k der Anzahl der maximal zu ermittelnden Teil-CSPs entspricht. Jede Wertebelegung ϕ mit $\phi(x'_j) = v_j$ entspricht dabei einer Zuordnung des Constraints $c_j \in C$ zu Teil-CSP v_j, wobei für v_j gleich 0 gilt, dass das Constraint c_j keiner Teilmenge zugeordnet wird. Ziel ist die Maximierung der Variable x_{opt}, die angibt, wie verbunden die einzelnen Teil-CSPs sind. Der maximal annehmbare Wert up ist dabei ein im Vorfeld nach oben abgeschätzter, nicht überschreitbarer, fester Wert in $\mathbb{N}$.

Auf das neu entwickelte *cspOverlapSplit*-Constraint soll an dieser Stelle nicht detailliert eingegangen werden, stattdessen wird auf das Papier [97] verwiesen. Vereinfacht ausgedrückt ordnet dieses Constraint den Variablen X' Werte $0, ..., k$ zu, so dass die Teil-CSPs P_i mit $i \in \{1, ..., k\}$, genau die Constraints $c_j \in C$ enthalten, deren korrespondierende Variablen $x'_j \in X'$ den Wert i zugeordnet bekommen haben. Dabei darf die Gesamtgröße jedes Teil-CSPs eine gegebene Grenze nicht überschreiten bei gleichzeitiger Maximierung des Verbundenheitswertes x_{opt}. Diese Herangehensweise ermöglicht es in *Regular*-CSPs Bereiche zu finden, die erstens klein genug sind, damit eine levelbasierte DFA-Vereinigung durchgeführt werden kann und zweitens über sehr viele *sharedVariables* verfügen. Mit dieser Erkennung von zu vereinigenden *Regular*-Constraints wurde ein weiterer Schritt in die vollautomatische Regularisierung getätigt. Mittels einer Anpassung der Größenschätzfunktion an die levelbasierten DFAs und deren Vereinigungsfunktion und einer detaillierteren Berechnung für die Bestimmung des Verbundenheitswertes x_{opt} kann dieses Vorgehen in Zukunft noch weiter verbessert werden.

Weitergehende Betrachtungen zur Regularisierung III: Ein Meta-CSOP zum Finden von Teil-CSPs, die besonders für die Parallelisierung geeignet sind

Analog zu dem im vorherigen Abschnitt beschriebenen Vorgehen kann ein Meta-CSOP entwickelt werden, das versucht, Teil-CSPs zu finden, die eine gegebene Größe nicht überschreiten und die nur über eine minimale Anzahl an *sharedVariables* verfügen. Die so entstehenden Teil-CSPs erlauben eine Zerlegung eines CSPs in Teile mit wenigen Berührungspunkten. Es ist somit möglich die Teil-CSPs in einem ersten Schritt unabhängig voneinander zu lösen und die Lösungen in einem zweiten Schritt zusammenzuführen (Problem Splitting). Für weitere Informationen wird auf die in Zusammenhang mit dieser Arbeit entstandene Publikation [96] verwiesen.

Die boolesche Skalarisierung

Bisher stellte sich die Vorhersage der Wirksamkeit der booleschen Skalarisierung als schwieriges Unterfangen heraus. Im Folgenden wird diskutiert, ob abgeschätzt werden kann, wann eine boolesche Skalarisierung sinnvoll ist. Dazu werden analog zum Vorgehen bei der levelbasierten DFA-Vereinigung die Erhöhungen der Anzahl der erzeugten *Scalar*- bzw. *Count*-Constraints und booleschen Variablen im Vergleich zur Anzahl der ursprünglichen Constraints und Variablen betrachtet. Für jede einzelne Probleminstanz der verschiedenen Anwendungsbeispiele aus den Abschnitten 6.2.1 bis 6.2.4 wurde jeweils die Anzahl der Variablen und Constraints der ursprünglichen CSPs und auch die der durch boolesche Skalarisierung erzeugten CSPs ermittelt, miteinander verglichen und die Vergrößerungsfaktoren V^c und V^v bezüglich der Anzahl der Constraints bzw. der Anzahl der Variablen gegenüber denen der ursprünglichen CSPs berechnet (siehe Gleichung 6.6 und 6.7). Sie sind inverse Maße für die Kompaktheit der Substituierung und geben an, wie viel mehr Variablen und Contraints durch die jeweilige Substituierung benötigt werden. Bei der Berechnung der beiden Maße bezeichnet C die Menge der ursprünglichen sowie C^S die Menge der durch boolesche Skalarisierung erzeugten *Scalar*-Constraints. Analog bezeichnet X die Menge der ursprünglichen Variablen und X^B die Menge der durch boolesche Skalarisierung erzeugten booleschen Variablen.

1. Vergrößerungsfaktor bezüglich der Anzahl an Constraints

$$V^c = |C^S|/|C| \tag{6.6}$$

2. Vergrößerungsfaktor bezüglich der Anzahl an Variablen

$$V^v = |X^B|/|X| \tag{6.7}$$

Bei der Regularisierung unterstützte die betrachtete Datenbasis die Hypothese, dass eine höhere Kompaktheit der erzeugten levelbasierten DFAs i.d.R. mit einer schnelleren Lösungsfindung einhergeht (Betrachtung der Werte *CiT* und *CiW*).

Gleiches gilt für die Verwendung der levelbasierten DFA-Vereinigung, welche umso gewinnbringender ist, je kompakter der resultierende DFA im Vergleich zum Ausgangs-DFA ist (R^s-, R^t- und R^v-Werte). Es lässt sich vermuten, dass sich auch mit Hilfe der booleschen Skalarisierung umso schneller eine Lösung finden lässt, je kompakter das erzeugte boolesche *Scalar*-CSPs ist, also je kleiner die V^c- und V^v-Werte sind. Somit lassen sich die zwei folgenden Hypothesen ableiten:

H1 Je kompakter die CSPs sind, die durch eine boolesche Skalarisierungsart (allgemein, direkt, *Regular*-basiert, *Count*-basiert, mit und ohne DFA-Vereinigung) erzeugt werden, umso größer ist deren Beschleunigungsfaktor a.

H2 Je kompakter ein CSP innerhalb einer booleschen Skalarisierungsart ist, umso größer ist sein Beschleunigungsfaktor a.

Zur Überprüfung der Hypothesen werden die Diagramme in Anhang D herangezogen. Sie stellen für alle Anwendungsbeispiele der Abschnitte 6.2.1 bis 6.2.4 die Beschleunigungsfaktoren a der jeweiligen booleschen Skalarisierungen ins Verhältnis zu den Vergrößerungsfaktoren V^c und V^v. Die Ordinate repräsentiert dabei jeweils den Beschleunigungsfaktor a und die Abszisse den jeweiligen Vergrößerungsfaktor V^v bzw. V^c. Angelegt wurden die Diagramme jeweils für große Schichtplanungs-, Black Hole-, Knight Tour- und Warehouse Location-Probleme mit und ohne Kapazitäten. Kleine Schichtplanungsprobleme wurden, da sie schnell gelöst werden können und in dieser Arbeit das Lösen großer Probleme im Vordergrund steht, vernachlässigt. Innerhalb jedes Diagrammes wurden für die verschiedenen booleschen Skalarisierungen verschiedene Farben verwendet.

Zur Unterstützung der ersten Hypothese müssten die Mittelpunkte der Punktewolken der verschiedenen booleschen Skalarisierungen eine Kurve von links oben (niedriger Vergrößerungsfaktor aber hoher Beschleunigungsfaktor) nach rechts unten (hoher Vergrößerungsfaktor aber niedriger Beschleunigungsfaktor) bilden. Zur Bestätigung der zweiten Hypothese müssten die einzelnen Punktewolken jeder booleschen Skalarisierung (einer Farbe) eine Kurve von links oben nach rechts unten bilden.

Betrachtungen zur Hypothese H1

Das Diagramm in Abbildung D.4 entspricht weitgehend der Erwartungshaltung zur ersten Hypothese. Die Punktewolken der verschiedenen Verfahren sind bis auf B^{DFA} von links nach rechts fallend. Um die Hypothese H1 exakt zu erfüllen, müsste die *Regular*-basierte boolesche Skalarisierung B^{DFA} im Durchschnitt schneller eine erste Lösung finden als das Erzeugen und Lösen boolescher *Count*-Constraints B^{Count} (oder der Vergrößerungsfaktor V^C von B^{DFA} müsste größer sein als der von B^{Count}). Diese Abweichung von den tatsächlichen Testergebnissen ist an dieser Stelle weniger relevant, da sowohl die Beschleunigungsfaktoren als auch die Vergrößerungsfaktoren der beiden Verfahren sehr dicht beieinander liegen. Das Diagramm in Abbildung D.3 bezüglich der Anzahl an Variablen sieht ähnlich aus, bis auf den Unterschied, dass die konfliktbasierte boolesche Skalarisierung B^K deutlich kompakter bezüglich ihrer Variablen ist als ihr Beschleunigungsfaktor vermuten lässt. Das liegt daran, dass bei der konfliktbasierten booleschen Skalarisierung, abgesehen von den booleschen Variablen zur Repräsentation der Variablenbelegungen, keine weiteren booleschen Variablen für die Erzeugung der *Scalar*-Constraints benötigt werden. Bei allen anderen booleschen Skalarisierungen werden für das Anlegen der Constraints zusätzliche boolesche Variablen benötigt. Da dies bei der konfliktbasierten booleschen Skalarisierung nicht der Fall ist, sollte dieses Vorgehen bei Betrachtung der Hypothese H1 nicht berücksichtigt werden.

Demgegenüber ist bezüglich der Hypothese H1 in den weiteren Diagrammen D.1 und D.2 sowie D.5 bis D.10 keine solche Tendenz erkennbar. Teilweise widersprechen die Darstellungen der ersten Hypothese sogar. Aus dem Diagramm in Abbildung

D.2 ist zu erkennen, dass die direkte boolesche Skalarisierung B^{D} kompakter ist als die *Regular*-basierte boolesche Skalarisierung mit DFA-Vereinigung $B^{DFA^{\cap}}$ und die wiederum kompakter ist als das Erzeugen boolescher *Count*-Constraints mit DFA-Vereinigung $B^{Count^{\cap}}$. Die Tabelle 6.3 zeigt aber genau entgegengesetzt dazu, dass $B^{Count^{\cap}}$ im Durchschnitt schneller eine Lösung findet als $B^{DFA^{\cap}}$, was wiederum schneller eine Lösung findet als B^{D}. Die erste Hypothese ist somit nicht oder nur unter bestimmten, noch nicht bekannten Voraussetzungen erfüllt.

Betrachtungen zur Hypothese H2

Auch zur Unterstützung der Hypothese H2 lassen sich Anwendungsbeispiele finden, die sie unterstützen. Werden die Diagramme aus den Abbildungen D.5 und D.6 betrachtet, so fällt auf, dass jede einzelne Punktewolke einen annähernd gleichen Verlauf hat. Vom kleinem Vergrößerungsfaktor V^{v} bzw. V^{c} zum großen (von links nach rechts) betrachtet starten die Beschleunigungswerte a jeweils pro Verfahren mit kleinen Werten, werden dann größer und fallen dann mit zunehmenden Vergrößerungsfaktor wieder ab. Wird die anfängliche Steigung vernachlässigt, so entspricht der Verlauf in etwa dem erwarteten für Hypothese H2. Die ersten Werte (also für kleine Vergrößerungsfaktoren) können dabei vernachlässigt werden, da sie sich auf kleine Instanzen des Knight Tour-Problems beziehen, die leicht zu lösen sind und der Fokus dieser Dissertation auf großen bzw. schwer zu lösenden CSPs liegt.

Unterstützen die Daten der Diagramme aus den Abbildungen D.3 und D.4 die erste Hypothese, so widersprechen sie der zweiten, die aussagt, dass ein geringerer Vergrößerungswert innerhalb einer Substituierungsart einen höheren Beschleunigungsfaktor a nach sich zieht und umgekehrt. In den beiden Diagrammen ist der Vergrößerungswert für jede Substituierungsart nahezu identisch, weswegen nicht erklärt werden kann, warum einzelne CSPs stark beschleunigt (z.B. Beschleunigungsfaktor a von über 2000 bei B^{DFA}) oder verlangsamt werden (Beschleunigungsfaktor a kleiner -10 bei B^{DFA}).

Zusammenfassende Betrachtungen zu den Hypothesen H1 und H2

Für spezielle Umgebungen wie das Anwendungsgebiet des Black Hole-Problems konnte zwar die erste Hypothese und für das Anwendungsgebiet des Knight Tour-Problems die zweite Hypothese grundsätzlich bestätigt werden, zusammengenommen widerlegen die Diagramme in den Abbildungen D.1 bis D.10 allerdings sowohl die erste als auch die zweite Hypothese oder schränken sie auf spezielle Bereiche ein. Daraus lässt sich schlussfolgern, dass die hier vorgestellte einfache Methodik nicht ausreicht, um präzise Vorhersagen über den Beschleunigungsfaktor konkreter boolescher Skalarisierungen zu treffen. Ein Vorgehen für die präzisere Abschätzung des Beschleunigungsfaktors kann dabei analog zu [94] ein maschineller Lernansatz sein. Hierzu sollte neben der Kompaktheit der booleschen *Scalar*-CSPs auch die Größe der Constraints (Anzahl beteiligter Variablen), die bei den booleschen *Scalar*-Constraints verwendeten Relationen

$(<, \leq, = \geq, >, \neq)$ und Skalare, die Art der Substituierung (*Regular*-, konflikt- oder Support-basiert bzw. direkt oder das Erzeugen von booleschen *Count*-Constraints) oder die Art des Constraints, das substituiert wird, bei der Berechnung untersucht werden.

Weitere Beobachtungen

Im Folgenden werden weitere Beobachtungen aus den Abbildungen D.1 bis D.10 genannt und ausgewertet.

- Die direkte boolesche Skalarisierung B^D benötigt die wenigsten Variablen und Constraints, ist also am kompaktesten. Ein Zusammenhang zwischen Kompaktheit und Beschleunigungsfaktor konnte aber nicht hergestellt werden.
- Das Erzeugen boolescher *Count*-Constraints benötigt immer mehr Variablen und Constraints als die *Regular*-basierte boolesche Skalarisierung. Die Ursache dafür ist, dass für die Erzeugung *Count*-basierter boolescher CSPs die *Scalar*-Constraints zur Beschreibung der Knoten Constraints C^{Node} (siehe Gleichung 5.18 und 5.30) zwei *Count*-Constraints (statt einem *Scalar*-Constraint) und eine zusätzliche Variable (*occ*) benötigt werden. Mit Ausnahme des Black Hole-Problems, bei dem der Unterschied zwischen der *Regular*-basierten booleschen Skalarisierung und dem Erzeugen und Lösen boolescher *Count*-Constraints sehr gering ist (ca. $0,5\%$), war dementsprechend die *Regular*-basierte boolesche Skalarisierung im Durchschnitt schneller als das Erzeugen und Lösen boolescher *Count*-Constraints.
- Die levelbasierte DFA-Vereinigung führte teilweise zu einer höheren und teilweise zu einer geringeren Kompaktheit. Ebenso führt sie teilweise zu Beschleunigungen und teilweise zu Verlangsamungen. Ein direkter Zusammenhang zwischen Kompaktheit und Beschleunigung konnte allerdings nicht festgestellt werden.
- Die Ausreißer in Abbildung D.1 und D.2 mit einem Vergrößerungsfaktor von eins entstehen, wenn das Erzeugen der levelbasierten DFAs zu einem leeren DFA, also einen ohne Lösungspfade, führt. Diese führen direkt zu einem Widerspruch und benötigen somit keine zusätzlichen booleschen Variablen oder weitere Constraints.

Der Portfolio-Ansatz

Aus den Abschnitten 6.2.1 bis 6.2.4 geht hervor, dass ein Portfolio-Ansatz stets schneller eine erste bzw. bessere Lösung finden konnte als die einzelnen Ansätze für sich. In den Tabellen 6.9 und 6.10 sind die Beschleunigungen der verschiedenen Portfolio-Ansätze für die Anwendungsbeispiele aus den Abschnitten 6.2.1 bis 6.2.4 dargestellt. Die Aufteilung auf zwei Tabellen ist dem geschuldet, dass bei den Problemen aus den Abschnitten 6.2.1 bis 6.2.3 nach einer ersten Lösung (Angabe in Sekunden), bei den Warehouse Location-Problemen (siehe Abschnitt 6.2.4) allerdings nach einer besten Lösung (Angabe mittels eines Beschleunigungsfaktors) gesucht wird.

Problem	$t_{Fst}^{\text{Portfolio}}$ in s	$t_{Fst}^{Original}$ in s	$\varnothing t_{Fst}$ in s	t_{Fst}^{Best} in s
Schichtplanung (klein)	1,066	40,40 (37,90)	21,57 (20,23)	1,266 (1,19)
Schichtplanung (groß)	40,41	206,77 (5,12)	156,28 (3,87)	47,78 $(1,18)$
Black Hole	33,98	481,63 (14,17)	306,28 (9,01)	98,29 (2,89)
Knight Tour	41	493 (12,01)	243 (5,92)	108 (2,63)

Tabelle 6.9: Die durchschnittlichen Zeitaufwände für das Finden einer ersten Lösung mit den Substituierungsverfahren und einem Portfolio-Ansatz.

Problem	P^1	P^2	P^3	*Original*	$\varnothing$	$\varnothing_4$	*Best*
WLP ohne Kapazitäten	16,04	15,71	6,2	1,0	-2,09	-1,15	1,55
WLP mit Kapazitäten	407,71	405,2	297,3	1,0	1,56	1,76	3,0

Tabelle 6.10: Die durchschnittlichen Beschleunigungsfaktoren a im Vergleich zum ursprünglichen Verfahren mit den Substituierungsverfahren und drei Portfolio-Ansätzen.

In Tabelle 6.9 sind jeweils die benötigten Zeiten für das Finden einer ersten Lösung mit dem Portfolio-Ansatz ($t_{Fst}^{\text{Portfolio}}$), dem ursprünglichen Ansatz ($t_{Fst}^{Original}$), dem Durchschnitt aller Ansätze ($\varnothing t_{Fst}$) und dem besten einzelnen Ansatz (t_{Fst}^{Best}) für die Anwendungsbeispiele aus den Abschnitten 6.2.1 bis 6.2.3 angegeben. Zusätzlich wurde für alle Ansätze (außer dem Portfolio-Ansatz) in Klammern der Beschleunigungsfaktor des Portfolio-Ansatzes gegenüber dem jeweiligen Ansatz aufgeführt. Das ermöglicht unter anderem einen Vergleich mit den Werten aus Tabelle 6.10. Bei dem Portfolio-Ansatz wurden jeweils alle in den Abschnitten 6.2.1 bis 6.2.3 aufgeführten Substituierungsarten unabhängig voneinander angewendet. Es ist zu erkennen, dass der Portfolio-Ansatz immer mindestens $1,18$ mal so schnell wie das beste einzelne Substituierungsverfahren, $3,87$ mal so schnell wie der Durchschnitt aller vorgestellten Verfahren und $5,12$ mal so schnell wie der ursprüngliche Ansatz war. Teilweise konnte sogar eine Beschleunigung um fast das 38-fache (der Portfolio-Ansatz gegenüber dem ursprünglichen Ansatz bei kleinen Schichtplanungsproblemen) erzielt werden.

In der Tabelle 6.10 sind die Beschleunigungsfaktoren der drei zuvor (auf Seite 208) vorgestellten Portfolio-Ansätze (P^1 = alle 16 Verfahren ohne geteilte Werte, P^2 = die vier erfolgversprechendsten Verfahren ohne geteilte Werte und P^3 = die vier erfolgversprechendsten Verfahren mit geteiltem Optimierungswert), des ursprünglichen CSOPs (*Original*), des Durchschnitts aller Substituierungsverfahren ($\varnothing$), der vier erfolgversprechendsten ($\varnothing_4$) und des besten Verfahrens (*Best*) dargestellt. Es ist deutlich erkennbar, dass die Portfolio-Ansätze signifikant besser sind als die Einzelverfahren. Besonders die beiden Verfahren ohne Austausch von Daten P^1 und P^2 überzeugen mit einem $15,71/1,55 = 10,14$ bis $407,71/3,0 = 135,90$ mal so hohen Beschleunigungsfaktor gegenüber dem besten eigenständigen Verfahren.

Nach den Erkenntnissen dieser Arbeit empfiehlt es sich, wann immer möglich, einen Portfolio-Ansatz zu verwenden. Dabei erscheint die Verwendung möglichst verschiedener Modelle (und Solver-Einstellungen) besonders vielversprechend zu sein, wodurch das Entwickeln weiterer Substituierungen von besonderem Wert sein kann.

Fazit

Abschließend lässt sich festhalten, dass Substituierungen sowohl einzeln als auch in der Verwendung eines Portfolio-Ansatzes zu einer erheblichen Beschleunigung des Lösungsvorganges von CSPs führen können. Dies konnte besonders bei der Substituierung von logischen Meta-Constraints und Constraint-Mengen mit hohem Vernetzungsgrad (also vielen $sharedVariables$) demonstriert werden. Die Substituierung einzelner globaler Constraints hingegen führte nur in sehr wenigen Fällen zu einer beschleunigten Lösungssuche.

Es wurde gezeigt, dass die Regularisierung und die verschiedenen booleschen Skalarisierungen konkurrenzfähig zur Tabularisierung und für einzelne Anwendungen sogar besser geeignet sein können (siehe das Warehouse Location-Problen in Abschnitt 6.2.4). Zusätzlich sind die vorgestellten Verfahren aufgrund der direkten Transformationen auch für große Constraints, insbesondere auch globale Constraints, geeignet. Das erhöht ihre Anwendbarkeit für reale Probleme gegenüber der Tabularisierung enorm. Es wurden erste Untersuchungen durchgeführt, die abschätzen, wann die Tabularisierung, die Regularisierung oder die boolesche Skalarisierung am besten auf ein CSP angewendet werden können. Zusätzlich wurde ein einfaches Threshold-Verfahren angegeben, das prognostiziert, wann die levelbasierte DFA-Vereinigung zu einer Zeitersparnis führt. In der Folge dieser Arbeit müssen die einzelnen Klassifikatoren weiter entwickelt und präzisiert werden. Verfahren des maschinellen Lernens bieten sich dabei für eine weitere Umsetzung an.

7 Zusammenfassung, Fazit und Ausblick

Ziel dieser Arbeit war die Remodellierung und Regularisierung von Finite Domain Constraint Satisfaction-Problemen mit der Absicht, die Dauer zum Finden einer ersten oder einer besten Lösung zu reduzieren. Im Folgenden werden das Vorgehen zum Erreichen dieses Zieles und dessen Ergebnisse zusammengefasst, ein abschließendes Fazit gezogen und ein Ausblick über zukünftige Arbeiten gegeben.

7.1 Zusammenfassung

Für die Erforschung neuer automatisierter Transformationsverfahren, welche den Lösungsprozess von Finite Domain Constraint Satisfaction-Problemen beschleunigen, wurden zunächst in Kapitel 2 die Grundlagen der Constraint-Programmierung erläutert. Nach einer Einführung der wesentlichen Begriffe erfolgte eine Klassifizierung von Constraint-Problemen nach ihren Domänen (boolesche, Finite Domain, reell-wertige) sowie eine kurze Vorstellung verschiedener Solver und der Funktionsweise der für diese Arbeit essenziellen FD-Solver. Abschließend wurde im Grundlagenkapitel das Konzept der globalen Constraints erläutert, bevor ausgewählte globale Constraints detaillierter beschrieben wurden.

Kapitel 3 beschäftigte sich mit zu dieser Arbeit verwandten Arbeiten und ging dabei auf drei bestehende Arten von Remodellierungen genauer ein: die binären Transformationen, die Umwandlung in SAT-Probleme und die Tabularisierung (Teilforschungsfrage TF1). Für die ersten beiden Arten wurden verschiedene Ausprägungen vorgestellt und verglichen und sie wurden von der in dieser Arbeit vorgestellten Regularisierung abgegrenzt. Da das Vorgehen bei der Tabularisierung dem der Regularisierung sehr ähnlich ist, wurden die einzelnen Schritte der beiden Vorgehen detailliert betrachtet. Abschließend wurde eine Übersicht der Abhängigkeiten der existierenden Verfahren gegeben und erläutert, wie die neu entwickelten Vorgehen (die Regularisierung und die boolesche Skalarisierung) die existierenden erweitern können.

In Kapitel 4 wurde das neu entwickelte Konzept der Finite Domain-vollständigen Constraints eingeführt und gezeigt, dass sowohl das *Table-* als auch das *Regular*-Constraint Vertreter dieser Klasse sind (Teilforschungsfrage TF2). Anhand von Beispielen wurde der Nutzen der Umwandlung von Constraints in *Table-* und *Regular*-Constraints veranschaulicht. Im zweiten Teil dieses Kapitels wurden die wesentlichen Vorteile der

Substituierungen mit Finite Domain-vollständigen Constraints erläutert (Teilforschungsfrage TF3). Das charakteristische Merkmal FD-vollständiger Constraints ist es dabei, dass sie aufgrund ihrer Struktur in der Lage sind, jedes andere FD-Constraint zu repräsentieren. Im Zuge der Diskussion spezialisierter Solver wurde das selbst entwickelte Konzept der Finite Domain-vollständigen CSPs vorgestellt und weiter in direkt und indirekt konvertierte CSPs differenziert. Direkt konvertierte CSPs können dabei erzeugt werden, ohne dass die ursprünglichen Variablen oder Domänen angepasst werden müssen. Es genügt also eine Substituierung der Constraints. Indirekt konvertierte CSPs erfordern hingegen auch eine Anpassung der Variablen und Domänen. Als Vertreter für direkt konvertierte FD-vollständige CSPs wurden *Table*- und *Regular*-CSPs und als Vertreter für indirekt konvertierte FD-vollständige CSPs SAT-Probleme, binäre CSPs und boolesche *Scalar*-CSPs vorgestellt.

Kapitel 5 umfasst die in Zusammenhang mit dieser Arbeit neu entworfenen Substituierungen. Das beinhaltet sowohl allgemeine Substituierungen, mit denen jedes Constraint und jedes CSP transformiert werden kann, als auch spezielle, besonders effektive Substituierungen von globalen Constraints in *Regular*- und boolesche *Scalar*-Constraints. Für die boolesche Skalarisierung wurden dabei zwei unterschiedliche allgemeine Substituierungen entwickelt, die sich in ihrer Herangehensweise dahingehend unterscheiden, dass sie entweder alle gültigen oder ungültigen Tupel durch *Scalar*-Constraints über booleschen Variablen beschreiben (Teilforschungsfrage TF4). Zusätzlich zu neuen Transformationen globaler Constraints wurden Möglichkeiten der Substituierung logischer Meta-Constraints entwickelt und deren Korrektheit bewiesen (Teilforschungsfrage TF5). Des Weiteren wurden die bis dahin vorgestellten Möglichkeiten der booleschen Skalarisierung um zwei weitere (das Erstellen *Count*-basierter CSPs und *Regular*-basierter CSPs) erweitert und erläutert, dass für diese noch spezialisierte, sehr leistungsstarke Solver entwickelt werden können.

Kapitel 6 beschäftigte sich mit der Evaluation der zuvor vorgestellten Substituierungsverfahren, die in zwei Schritten durchgeführt wurde. Zunächst wurden die Substituierungen auf spezielle CSPs, welche jeweils nur ein globales Constraint beinhalten, angewendet und die Dauer für das Finden einer ersten Lösung der transformierten CSPs mit der des ursprünglichen verglichen. Dabei konnte festgestellt werden, dass bei der Substituierung globaler Constraints, bis auf wenige Ausnahmen wie zum Beispiel beim *Global Cardinality*-Constraint, die ursprünglichen Constraints im Durchschnitt schneller zu einer ersten Lösung führten als die durch Substituierung erzeugten. Dieses Ergebnis ist nicht überraschend, schließlich basiert die Propagation globaler Constraints auf jahrzehntelanger Forschung und Spezialisierung, die diese so effektiv gestalten. Es konnte in diesem Abschnitt allerdings bereits gezeigt werden, dass die durch allgemeine Substituierungen auftretenden hohen Transformationsdauern mittels direkter Substituierungen soweit reduziert werden konnten, dass diese in der Regel nur einen unwesentlichen Anteil an der Dauer zum Finden einer ersten Lösung ausmachen. Im zweiten Teil der Evaluation wurden reale Anwendungsprobleme betrachtet. Dafür wurde für jede Problemstellung jeweils eine aus der Literatur bekannte Standardmodellierung vorgegeben und verschiedene Probleminstanzen dazu erzeugt, die anschließend mithilfe der verschiedenen Substituierungen gelöst wurden. Die benötigten Dauern zum Finden einer ersten bzw. einer besten Lösung wurden anschließend miteinander verglichen und ausgewertet. Abschließend wurden die Ergebnisse der Anwendungsbeispiele ausgewertet und festgestellt, dass die Regularisierung und teilweise auch die boolesche Skalarisierung konkurrenzfähig zur Tabularisierung sind (Teilforschungsfrage TF6). Da sich nicht jedes Verfahren für jede Probleminstanz als gleich geeignet darstellte, wurden in der Folge Abschätzungen getroffen, wann welches Verfahren besonders geeignet ist (Teilforschungsfrage TF8). Insbesondere wurde dabei auch bereits oberflächlich überprüft, wann die Vereinigung mehrerer levelbasierter DFAs zielführend ist (Teilforschungsfrage TF7). Mittels Portfolio-Ansätzen konnten schließlich alle Verfahren zusammengeführt werden und deren großer, gemeinsamer Nutzen hervorgehoben werden.

7.2 Fazit

Alles in allem betrachtet kann die Zielstellung, bestehende Remodellierungsverfahren zu erweitern und neue zu entwickeln, mit dem Ziel, die Dauer zum Finden einer ersten oder einer besten Lösung zu reduzieren, als erreicht angesehen werden (Hauptforschungsfrage HF). Auf der einen Seite konnten die Regularisierung weiterentwickelt und auf der anderen Seite verschiedene neue Formen der booleschen Skalarisierung (Support-, konflikt-, *Regular*-basierte und direkte boolesche Skalarisierung bzw. das Erstellen boolescher *Count*-Constraints) erforscht und entwickelt werden. Das bei Remodellierungen gängige Problem der zu hohen Transformationsdauern konnte durch das Schaffen direkter Transformationen ausgewählter globaler Constraints großteils minimiert werden, so dass die Transformationsdauern bei den präsentierten Anwendungsbeispielen keinen signifikanten Einfluss an der Gesamtlösungsdauer hatten.

Es konnte eindrucksvoll gezeigt werden, dass mit Hilfe der neu entwickelten Substituierungen nun zur Tabularisierung konkurrenzfähige Remodellierungsmöglichkeiten geschaffen wurden, die verschiedene Vorteile gegenüber der Tabularisierung aufweisen. So können die Regularisierung und die booleschen Skalarisierungen aufgrund der direkten Transformationsmöglichkeit für eine Vielzahl von globalen Constraints angewendet werden, für die die Tabularisierung nicht in Frage kommt. Die Regularisierung realisiert dabei häufig die Umwandlung eines beliebigen Constraints in genau ein *Regular*-Constraint. Durch weiterführende Maßnahmen wie der Minimierung oder levelbasierten DFA-Vereinigung der zugrundeliegenden levelbasierten DFAs ergeben sich weitere Optionen, mit denen der an die Regularisierung folgende Lösungsvorgang weiter beschleunigt werden kann. Erste Vorgehen zur Verwendung der levelbasierten DFA-Vereinigung wurden dazu in Abschnitt 6.2.5 gezeigt.

Die booleschen Skalarisierungen, die in einem Zwischenschritt eine Regularisierung durchführen (*Regular*-basierte boolesche Skalarisierung und das Erzeugen boolescher *Count*-Constraints), können die zuvor genannten Optimierungen der levelbasierten DFAs ebenfalls durchführen, um so kompaktere und schneller zu lösende boolesche *Scalar*-Constraints zu erzeugen. Die Regulariserung ermöglicht oftmals die Transformation eines globalen Constraints in ein *Regular*-Constraint. Abhängig von der Größe des zugrunde liegenden levelbasierten DFAs entsteht dabei allerdings gegebenenfalls eine sehr große Datenstruktur. Durch die direkte Transformation in boolesche *Scalar*-Constraints wird das Wissen der ursprünglichen globalen Constraints hingegen teilweise sehr kompakt in einem *Scalar*-Constraint über boolesche Variablen repräsentiert. Abhängig davon, wie viele boolesche Variablen für die Repräsentation der ursprünglichen Variablen notwendig sind, können die benötigten Datenstrukturen somit in der Regel sehr klein gehalten werden.

Zusammenfassend kann gesagt werden, dass alle vorgestellten Substituierungen ihre Daseinsberechtigung haben und zur beschleunigten Lösungssuche beitragen können. Bei allen Testreihen zu den vorgestellten Anwendungsbeispielen waren sowohl die Regularisierung als auch verschiedene Formen der booleschen Skalarisierung im Durchschnitt schneller als die ursprünglichen Verfahren. Besonders vorteilhaft erscheinen die einzelnen Substituierungen, wenn durch diese das zu lösende CSP sehr kompakt repräsentiert werden kann. Werkzeuge zur Erzeugung kompakter CSPs können neben der eigentlichen Transformation des CSPs auch die Minimierung und Vereinigung der zugrundeliegenden Datenstrukturen sein. Die entscheidende Frage ist, wann welche Substituierung bevorzugt einzusetzen ist. Diesbezüglich konnten erste Schritte zur Erstellung eines Klassifikators gegangen werden. Eine Möglichkeit, diesen bisher unvollständigen Klassifikator zu umgehen, stellt die Verwendung eines Portfolio-Ansatzes dar. Mit Hilfe von Portfolio-Ansätzen konnten für alle Anwendungsbeispiele erhebliche durchschnittliche Beschleunigungen bei der Suche nach einer ersten bzw. besten Lösung erreicht werden.

Festgehalten werden muss auch, dass die präsentierten Substituierungen bereits zu Beschleunigungen führen, obwohl eine gezielte Anpassung von Solver-Einstellungen bzw. eine Verwendung von auf die erstellten Constraints zugeschnittenen Solvern bisher nicht erfolgt ist. Wird dies in der Zukunft noch berücksichtigt, so ist davon auszugehen, dass die einzelnen Substituierungen noch vorteilhafter eingesetzt werden können.

7.3 Ausblick

Abschließend wird ein Ausblick gegeben, welche Forschungsarbeiten sich in Anschluss an die Ergebnisse dieser Arbeit ergeben. Dieser Ausblick unterteilt sich in drei Abschnitte: weitere Substituierungen, spezialisierte Solver und Solver-Einstellungen sowie weitere Optimierungen, die in der Folge beschrieben werden.

Weitere Substituierungen

Nachdem gezeigt werden konnte, dass die vorgestellten Substituierungen (die allgemeine und direkte Regularisierung, die Support-, konflikt-, *Regular*-basierte und direkte boolesche Skalarisierung sowie das Erzeugen von booleschen *Count*-Constraints) zu Beschleunigungen bei der Lösungssuche führen, stellt sich die Frage, welche weiteren Substituierungsverfahren erforscht und entwickelt werden können, die zu einer Beschleunigung beim Lösen eines CSPs führen. Ohne zu wissen, wie solche zukünftigen Verfahren aussehen werden, lassen sich diese wie folgt kategorisieren:

- **Substituierungen, die auf anderen Constraints basieren:** Neben der Umwandlung in *Regular-*, *Table-*, *Scalar-* oder *Count*-Constraints sollten weitere Constraint-Arten und Gruppen von Constraint-Arten untersucht werden, in die Ausgangs-Constraints überführt werden können.

- **Boolesche Skalarisierungen, die auf anderen Herleitungen basieren:** Bisher wurden bei den allgemeinen booleschen Skalarisierungen entweder alle gültigen oder ungültigen Tupel aufgelistet und für jedes dieser Tupel ein *Scalar*-Constraint über booleschen Variablen erzeugt. Für die Zukunft scheint es vielversprechend zu sein, die Repräsentation mehrerer gültiger oder ungültiger Tupel durch ein Constraint auszudrücken, oder die beiden bisherigen (und zukünftige) Verfahren miteinander zu kombinieren.

  Die speziellen Verfahren basieren bisher auf direkten Substituierungen oder auf der Regularisierung. Möglicherweise lassen sich weitere zweistufige Substituierungen über das *Regular*-Constraint oder andere Constraints finden. Des Weiteren sind mehrstufige Substituierungen über verschiedene Zwischenschritte denkbar.

- **Weitere direkte Substituierungen:** Bisher wurden zu verschiedenen globalen Constraints direkte Substituierungen erstellt und deren Korrektheit bewiesen. Die Entwicklung weiterer direkter Transformationen für Constraints, für die solche bisher noch nicht existieren oder aber effektivere Substituierungen für bereits vorhandene, stellt einen essentiellen Bestandteil zukünftiger Forschungen dar.

Spezialisierte Solver und Solver-Einstellungen

Neben weiteren Substituierungsmöglichkeiten wird sehr viel Potential darin gesehen, spezielle Solver für die erzeugten CSPs zu erforschen, wie es z.B. bei SAT-Solvern oder pseudo-booleschen SAT-Solvern schon geschehen ist. Durch die Verwendung von Solvern, die genau auf die jeweilige Constraint-Domäne zugeschnitten sind, wird sich eine deutliche Beschleunigung beim Lösen der entsprechenden CSPs erhofft. Da oftmals nicht das gesamte CSP, sondern lediglich Teile davon substituiert werden, erscheint es ratsam, zwei Kategorien von Solvern zu entwickeln: Solver, die nur Constraints einer bestimmten Art lösen können, und Solver, die eine bestimmte Art von Constraints besonders gut, aber auch beliebige FD-Constraints lösen können.

Unabhängig von der soeben beschriebenen Art des Solvers sind bereits jetzt die folgenden Solver denkbar:

- **Ein *Regular*-Solver:** Ein solcher Solver soll die den *Regular*-Constraints zugrundeliegenden levelbasierten DFAs möglichst geschickt zu einem einzelnen levelbasierten DFA verschmelzen können. Jeder Pfad vom Start- zum Zielzustand innerhalb des vereinten DFAs repräsentiert dann eine Lösung des ursprünglichen CPSs, welches mit Erstellung des DFAs vollständig gelöst wäre. Zur optimalen Suche nach einer Lösung muss der Solver so angepasst werden, dass er nicht den gesamten vereinten levelbasierten DFA erzeugt, sondern nur einen ersten Lösungspfad davon.

- **Ein boolescher *Scalar*-Solver:** Ein solcher Solver sollte auf optimierten Verfahren zur Vereinfachung und Lösung von FD-Ungleichungen basieren. Es ist davon auszugehen, dass durch die Verwendung ausschließlich boolescher Variablen eine schnellere Lösungsfindung möglich ist als bei der Lösungssuche in beliebigen FD-Ungleichungssystemen.

- **Ein spezieller boolescher Gleichungs-Solver:** Dieser Solver muss darauf spezialisiert sein, lineare Gleichungssysteme zu lösen, wobei die Variablen alle boolesche Domänen und die Koeffizienten alle den Wert 1 oder -1 haben, und als Resultate nur die Werte $-1, 0$ und 1 in Frage kommen (siehe Abschnitt 5.4.2).

- **Ein boolescher *Count*-Solver:** Ein solcher Solver muss in der Lage sein, CSPs mit booleschen Variablen und ausschließlich *Count*-Constraints über diesen zu lösen. Es wird an dieser Stelle vermutet, dass ein Algorithmus ähnlich dem, wie er zum Propagieren des *Count*-Constraints im Choco-Solver verwendet wird [129], auf die Auswertung mehrerer *Count*-Constraints gleichzeitig erweitert werden kann.

Für weitere potentielle Substituierungsverfahren, wie sie im vorherigen Abschnitt beschrieben wurden, sind weitere spezialisierte Solver denkbar und notwendig.

Ein anderer Aspekt, der bisher kaum diskutiert wurde, ist die Möglichkeit, die Einstellungen der verwendeten Solver an die jeweils substituierten CSPs anzupassen. Abhängig davon, welche Constraints besonders häufig im CSP vorkommen oder den Suchraum besonders stark beschneiden oder von denen am häufigsten der Propagator aufgerufen wird, können unterschiedliche Solver-Einstellungen günstig oder ungünstig ausfallen. Zu solchen Solver-Einstellungen gehören unter anderem die verwendete Suchstrategie (Variablen- und Wertauswahlheuristik), die Propagationsreihenfolge der einzelnen und aller Constraints sowie ob nogood-Lernen oder bestimmte Backtracking-/Backjumping-Varianten zum Einsatz kommen sollen.

Weitere Optimierungen

Das Bearbeiten des Forschungsthemas hat ergeben, dass auch andere hilfreiche Optimierungsansätze weiter verfolgt werden sollten.

- **Minimierung von Gleichungs-, Ungleichungs- und *Count*-Systemen:** Durch die boolesche Skalarisierung entstehen entweder Gleichungs-, Ungleichungs- oder *Count*-Systeme über booleschen Variablen. Eine Minimierung bzw. Lösung dieser vor dem Start der Backtracking-behafteten Suche kann den Suchraum und gegebenenfalls auch die Constraints signifikant verkleinern, wodurch sowohl eine schnellere Propagation als auch Suche möglich ist.
- **Vereinigung von Constraints:** Sowohl bei der Regularisierung als auch bei dem Erzeugen boolescher *Scalar*-CSPs, die die Regularisierung als Zwischenschritt enthalten, hatte die Vereinigung von Constraints einen Einfluss auf die Lösungsgeschwindigkeit. Dementsprechend sollte weiter untersucht werden, wann die Vereinigung sich beschleunigend auswirkt und ob Vereinigungen für weitere gleiche oder unterschiedliche Constraint-Arten möglich sind.
- **Klassifikatoren für die Auswahl geeigneter Substituierungen:** Es konnte gezeigt werden, dass verschiedene Substituierungen (Regularisierung, Tabularisierung, verschiedene Arten der booleschen Skalarisierung) zur beschleunigten Lösung von Constraint-Problemen führen können. Für die Zukunft scheint es unumgänglich, einen Klassifikator zu erforschen, der prognostiziert, wann welche Substituierung zu einer bzw. zur größten Beschleunigung führt. Mit der Entwicklung weiterer Transformationsverfahren kommt einem solchen Klassifikator eine immer größere Bedeutung zu. Mögliche Ansätze für die Entwicklung eines Klassifikators bieten dabei unter anderem die verschiedenen Methoden des maschinellen Lernens.

Abschließend lässt sich sagen, dass die Forschung und Entwicklung innerhalb der Constraint-Programmierung in den letzten Jahrzehnten stetig voran ging und noch viele Forschungsthemen für weitere Jahrzehnte offen hält.

Anhang

A Die Evaluation einzelner globaler Constraints - Tabellen

Die im Folgenden dargestellten Tabellen fassen die Ergebnisse der in Abschnitt 6.1 vorgestellten Testreihen zusammen. Zu jedem Constraint-Typ T (*Count*, *Global Cardinality* usw.) aus dem Abschnitt 5.2 wurden dabei einzelne Probleminstanzen $P = (\{x_1, ..., x_n\}, \{D_1, ..., D_n\}, \{c^T\})$ mit genau einem globalem Constraint c^T vom Typ T mit $scope(c^T) = \{x_1, ..., x_n\}$ generiert. Als Zeitlimit für das Substituieren und Finden einer ersten Lösung wurde eine Minute gesetzt. Benötigte die Substituierung mehr Zeit oder scheiterte sie an technischen Engpässen (zu wenig Arbeitsspeicher), so wurde bei der Berechnung des Durchschnittswertes, sowohl bei der Zeit für die Substituierung, als auch für das Finden einer ersten Lösung, eine Minute angenommen.

Es sind jeweils die durchschnittlichen Zeiten für das Finden einer ersten Lösung T_{Fst} und das Durchführen der Substituierungen T_{Sub} sowie die Anzahl der Fälle, in denen durch die Substituierung eine Beschleunigung *#Pos* oder eine Verlangsamung *#Neg* auftrat und die Anzahl der Fälle, in denen kein Ergebnis gefunden werden konnte *#Fail*, aufgeführt. Im Verlauf der Auswertungen zeigte sich, dass die allgemeinen Substituierungen wesentlich zeitaufwendiger sind als die direkten Transformationen, weswegen für die beiden Verfahren $B(\{c\}^{DFA})$ und $B(\{c\}^{Count})$, die eine Regularisierung als Zwischenschritt benötigen, lediglich die direkte und nicht die allgemeine Regularisierung verwendet wurde.

Für alle nachfolgenden Tabellen gilt die Legende:

T_{Fst} = Zeit für das Finden einer ersten Lösung
T_{Sub} = Zeit für das Durchführen der Substituierung
#Pos = Anzahl Probleme, die schneller gelöst werden konnten
#Neg = Anzahl Probleme, die langsamer gelöst wurden
#Fail = Anzahl Probleme, die nicht im Zeitlimit (1 min) gelöst werden konnten

Global Cardinality

GCC	Original	$T(\{c\})$	$R(\{c\})$	$R(\{c\}^D)$	$B(\{c\}^S)$	$B(\{c\}^K)$	$B(\{c\}^D)$	$B(\{c\}^{DFA})$	$B(\{c\}^{Count})$
T_{Fst} in s	5,67	18,1	19,3	5,74	19	28,6	2,87	4,54	4,670
T_{Sub} in s	0	18	19,2	0,095	18,6	27,1	0,023	0,627	0,239
#Pos	-	0	0	17	0	0	49	23	21
#Neg	-	165	140	151	143	108	119	145	147
#Fail	0	3	28	0	25	60	0	0	0

Tabelle A.1: Zeitaufwände für das Substituieren von *Global Cardinality*-Constraints.

AllDifferent

AllDifferent	Original	$T(\{c\})$	$R(\{c\})$	$R(\{c\}^D)$	$B(\{c\}^S)$	$B(\{c\}^K)$	$B(\{c\}^D)$	$B(\{c\}^{DFA})$	$B(\{c\}^{Count})$
T_{Fst} in s	0,2	27,899	30,237	27,378	30,115	28,712	7,659	7,739	7,953
T_{Sub} in s	0	27,811	30,158	2,682	28,997	28,345	0,039	5,895	3,212
#Pos	-	0	0	3	0	0	13	1	1
#Neg	-	28	25	44	26	27	36	46	46
#Fail	0	21	24	2	23	22	0	2	2

Tabelle A.2: Zeitaufwände für das Substituieren von *AllDifferent*-Constraints.

AllEqual

AllEqual	Original	$T(\{c\})$	$R(\{c\})$	$R(\{c\}^D)$	$B(\{c\}^S)$	$B(\{c\}^K)$	$B(\{c\}^D)$	$B(\{c\}^{DFA})$	$B(\{c\}^{Count})$
T_{Fst} in s	0,181	0,232	0,273	0,372	0,259	30,611	0,24	0,459	0,440
T_{Sub} in s	0	0,081	0,117	0,202	0,107	28,444	0,053	0,266	0,239
#Pos	-	0	0	1	0	0	5	1	1
#Neg	-	49	49	48	49	26	44	48	48
#Fail	-	0	0	0	0	23	0	0	0

Tabelle A.3: Zeitaufwände für das Substituieren von *AllEqual*-Constraints.

Scalar

Scalar	Original	$T(\{c\})$	$R(\{c\})$	$R(\{c\}^D)$	$B(\{c\}^S)$	$B(\{c\}^K)$	$B(\{c\}^D)$	$B(\{c\}^{DFA})$	$B(\{c\}^{Count})$
T_{Fst} in s	0,187	19,5	21,9	2,09	21	35	3,15	2,25	2,21
T_{Sub} in s	0	19,4	21,8	1,91	20,4	32,2	0,026	2,03	1,98
#*Pos*	-	0	0	95	0	0	91	68	96
#*Neg*	-	283	271	293	268	190	309	320	292
#*Fail*	0	117	129	12	132	210	0	12	12

Tabelle A.4: Zeitaufwände für das Substituieren von *Scalar*-Constraints.

Sum

Sum	Original	$T(\{c\})$	$R(\{c\})$	$R(\{c\}^D)$	$B(\{c\}^S)$	$B(\{c\}^K)$	$B(\{c\}^D)$	$B(\{c\}^{DFA})$	$B(\{c\}^{Count})$
T_{Fst} in s	0,188	15,837	17,469	0,289	17,171	38,659	7,409	0,226	0,274
T_{Sub} in s	0	15,713	17,354	0,098	16,581	34,326	0,026	0,051	0,081
#*Pos*	-	0	0	9	0	0	24	16	16
#*Neg*	-	76	73	91	73	44	76	84	84
#*Fail*	0	24	27	0	27	56	0	0	0

Tabelle A.5: Zeitaufwände für das Substituieren von *Sum*-Constraints.

Table

Table	Original	$R(\{c\})$	$B(\{c\}^S)$	$B(\{c\}^K)$	$B(\{c\}^{DFA})$	$B(\{c\}^{Count})$
T_{Fst} in s	0,211	3,580	0,621	37,700	3,860	3,900
T_{Sub} in s	0	3,400	0,367	34,300	3,620	3,570
#*Pos*	-	0	0	0	0	0
#*Neg*	-	285	300	132	285	285
#*Fail*	0	15	0	168	15	15

Tabelle A.6: Zeitaufwände für das Substituieren von Table-Constraints.

Regular

Regular	Original	$T(\{c\})$	$B(\{c\}^S)$	$B(\{c\}^K)$	$B(\{c\}^D)$	$B(\{c\}^{Count})$
T_{Fst} in s	3,405	3,598	3,758	37,660	0,990	1,193
T_{Sub} in s	0	3,514	3,603	34,351	0,708	0,671
$\#Pos$	-	0	0	0	16	34
$\#Neg$	-	285	285	132	269	251
$\#Fail$	15	15	15	168	15	15

Tabelle A.7: Zeitaufwände für das Substituieren von Regular-Constraints.

B Die Evaluation einzelner globaler Constraints - Diagramme

Die im folgenden dargestellten Diagramme fassen die Ergebnisse der in Abschnitt 6.1 vorgestellten Testreihen zusammen. Zu jedem Constraint-Typ T (*Count*, *Global Cardinality* usw.) aus dem Abschnitt 5.2 wurden dabei einzelne Probleminstanzen $P = (\{x_1, ..., x_n\}, \{D_1, ..., D_n\}, \{c^T\})$ mit genau einem globalem Constraint c^T vom Typ T mit $scope(c^T) = \{x_1, ..., x_n\}$ generiert. Als Zeitlimit für das Substituieren und Finden einer ersten Lösung wurde eine Minute gesetzt. Benötigte die Substituierung mehr Zeit oder scheiterte sie an technischen Engpässen (zu wenig Arbeitsspeicher), so wurde bei der Berechnung des Durchschnittswertes, sowohl bei der Zeit für die Substituierung, als auch für das Finden einer ersten Lösung, eine Minute angenommen.

Es sind jeweils die durchschnittlichen Substituierungs- (T_{Sub}) und Lösungszeiten (T_{Fst}) für das Finden einer ersten Lösung für die generierten CSPs mit dem jeweiligen Constraint-Typ dargestellt. Die Säulen stellen den Zeitaufwand für das Finden einer ersten Lösung mittels des ursprünglichen CSP (Original), mittels eines tabularisierten CSP $T(\{c\})$ oder eines regularisierten CSPs $R(\{c\})$ beziehungsweise mittels eines Support-basierten $B(\{c\}^S)$, eines konfliktbasierten $B(\{c\}^K)$, eines *Regular*-basierten $B(\{c\}^{DFA})$ oder eines *Count*-basierten booleschen *Scalar*-CSPs $B(\{c\}^{Count})$ dar. Die schraffierten (unteren) Säulenanteile repräsentieren dabei den Zeitaufwand, der für die Substituierung des Problems notwendig war. Die grünen Säulen veranschaulichen, dass die allgemeinen Substituierungen (siehe Abschnitt 5.1) verwendet wurden, wohingegen die gelben Säulen veranschaulichen, dass die speziellen Substituierungen (siehe Abschnitt 5.2) angewendet wurden. Im Verlauf der Auswertungen zeigte sich, dass die allgemeinen Substituierungen wesentlich zeitaufwendiger sind als die direkten Transformationen, weswegen für die beiden Verfahren $B(\{c\}^{DFA})$ und $B(\{c\}^{Count})$, die eine Regularisierung als Zwischenschritt benötigen, lediglich die direkte und nicht die allgemeine Regularisierung verwendet wurde. Die durchgehend eingezeichnete Linie gibt den benötigten Zeitbedarf des ursprünglichen CSPs an, der als Vergleichswert für die substituierten dient. Eine durchgehende weiße räumliche Trennung veranschaulicht, dass sich die Skalierung der Ordinate ändert.

Global Cardinality

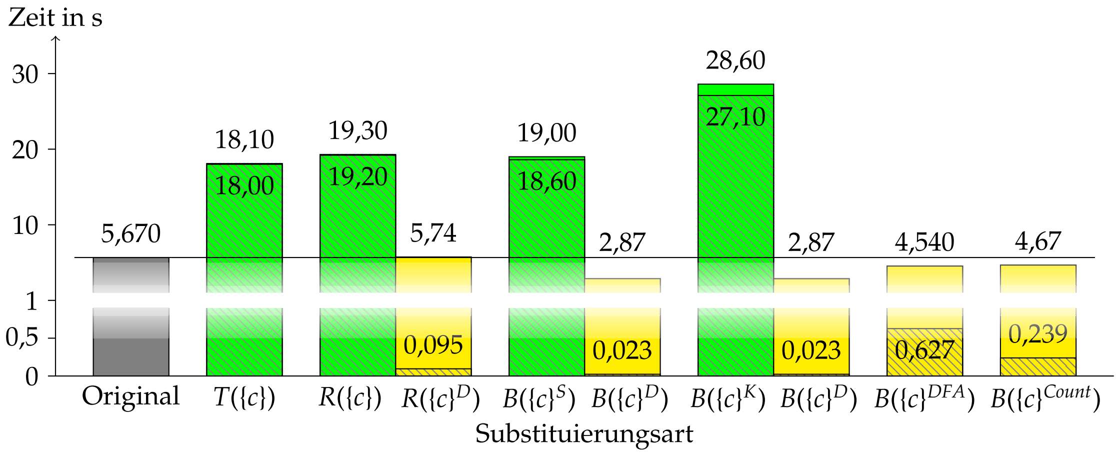


Abbildung B.1: Die durchschnittlichen Substituierungs- und Lösungszeiten (für das Finden einer ersten Lösung) für die generierten *Global Cardinality*-Constraints.

AllDifferent

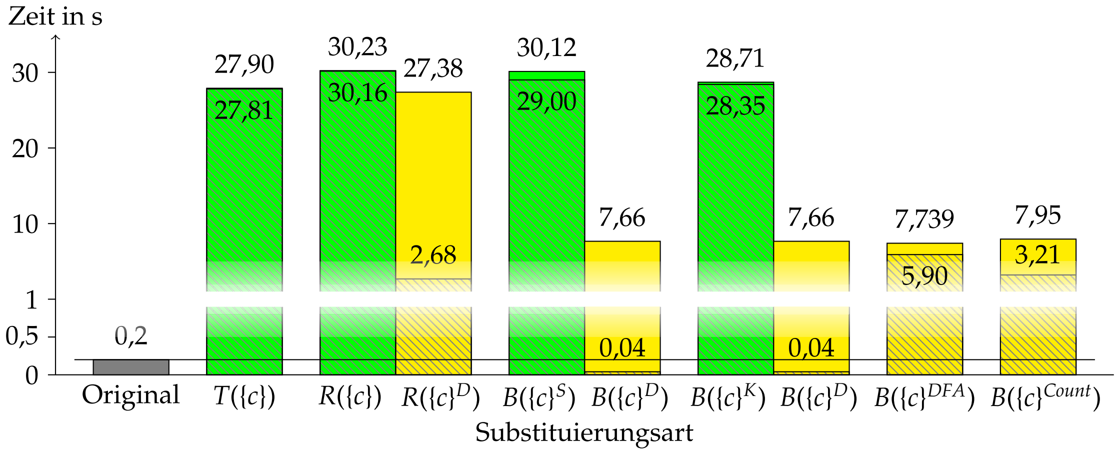


Abbildung B.2: Die durchschnittlichen Substituierungs- und Lösungszeiten (für das Finden einer ersten Lösung) für die generierten *AllDifferent*-Constraints.

AllEqual

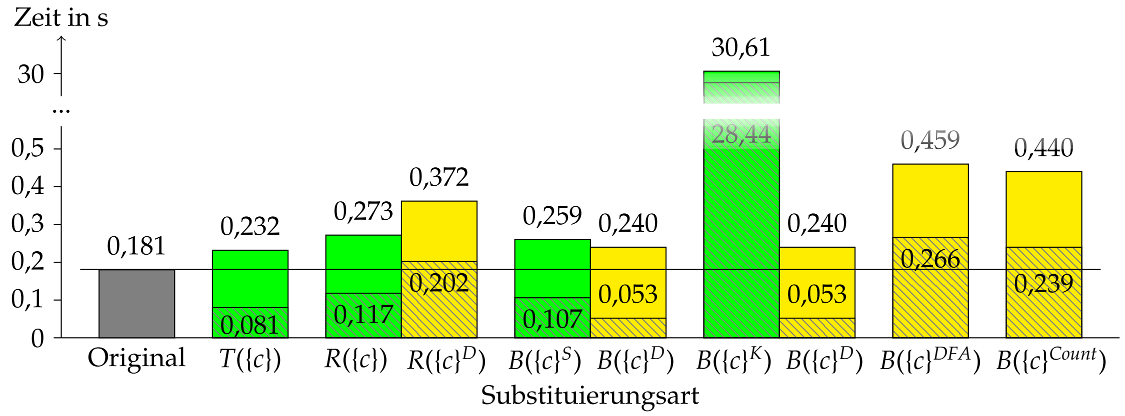


Abbildung B.3: Die durchschnittlichen Substituierungs- und Lösungszeiten (für das Finden einer ersten Lösung) für die generierten *AllEqual*-Constraints.

Scalar

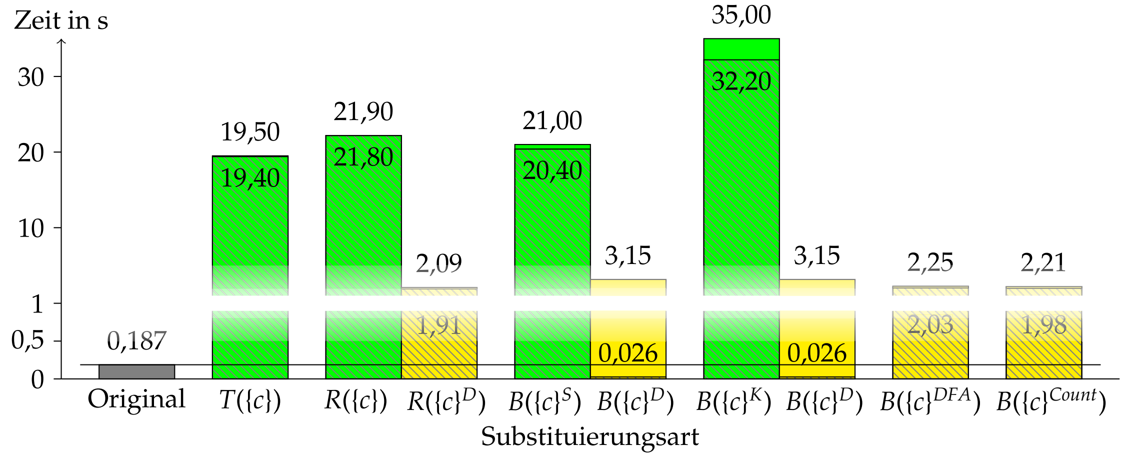


Abbildung B.4: Die durchschnittlichen Substituierungs- und Lösungszeiten (für das Finden einer ersten Lösung) für die generierten *Scalar*-Constraints.

Sum

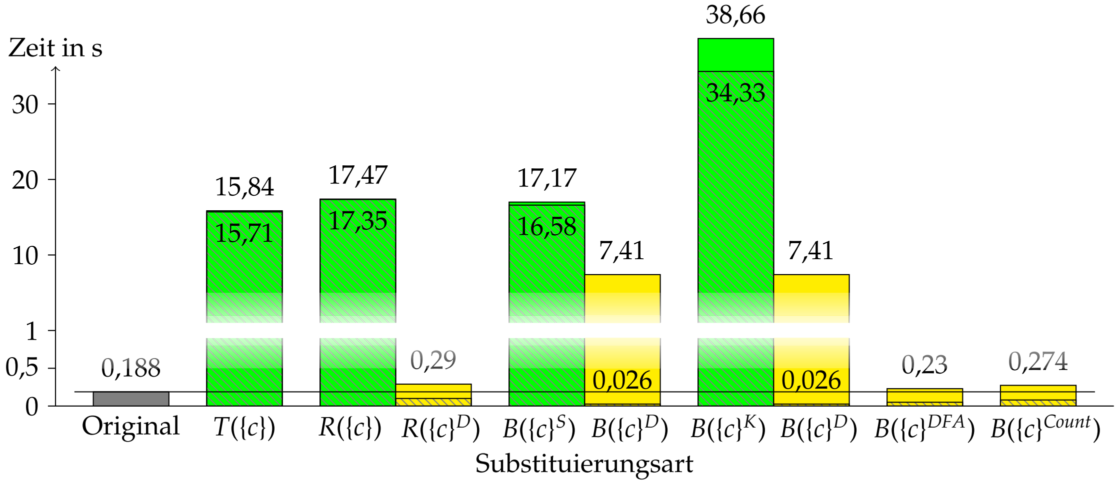


Abbildung B.5: Die durchschnittlichen Substituierungs- und Lösungszeiten (für das Finden einer ersten Lösung) für die generierten *Sum*-Constraints.

Table

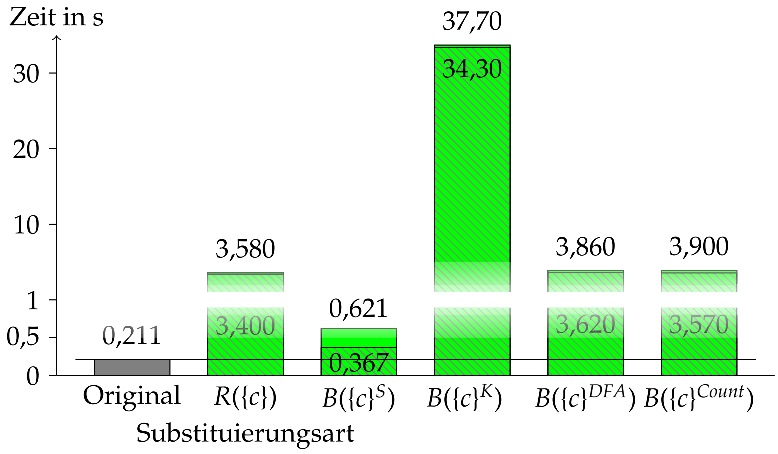


Abbildung B.6: Die durchschnittlichen Substituierungs- und Lösungszeiten (für das Finden einer ersten Lösung) für die generierten *Table*-Constraints.

Regular

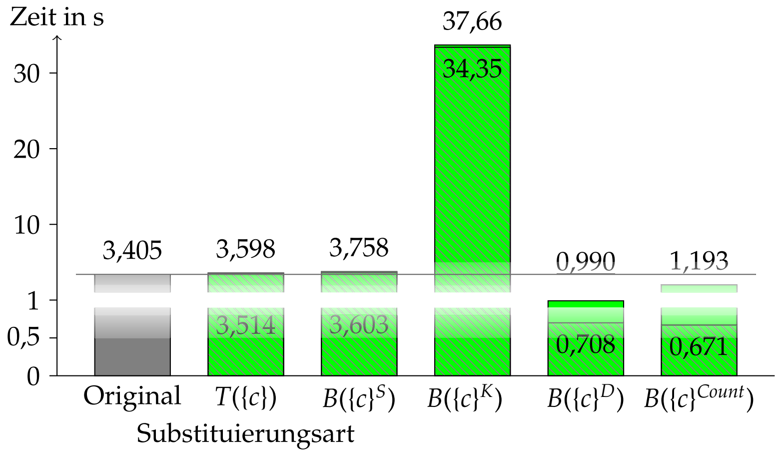


Abbildung B.7: Die durchschnittlichen Substituierungs- und Lösungszeiten (für das Finden einer ersten Lösung) für die generierten *Regular*-Constraints.

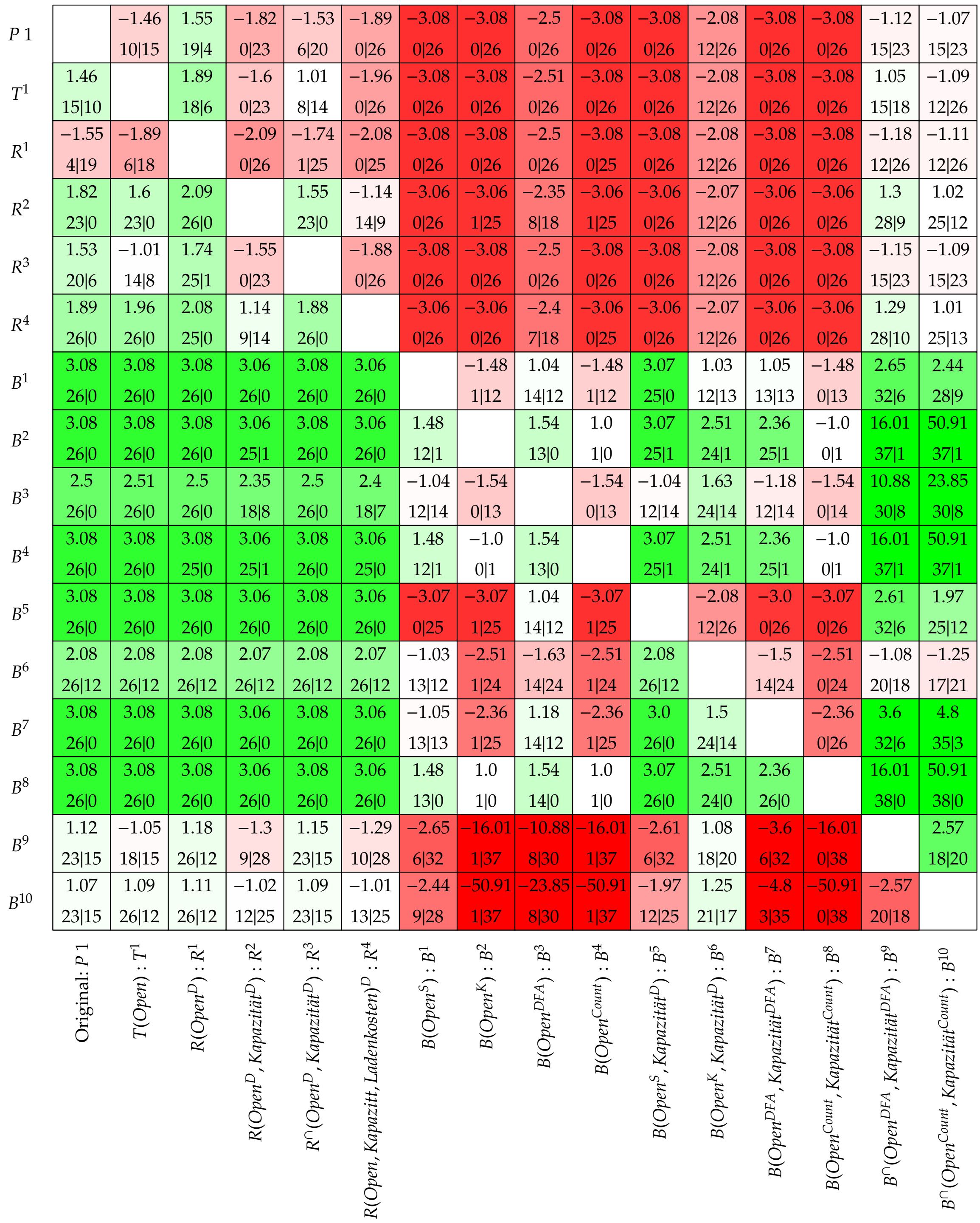
C Instanzen und Evaluation des Warehouse Location-Problems

Tabelle C.1 zeigt eine Liste der Bezeichner der Instanzen vom Warehouse Location-Problem, die für die Untersuchungen in Abschnitt 6.2.4 verwendet wurden. Die Testfälle ergeben sich aus dem Beispielproblem „cap0 “ der CSPlib [69] und weiteren 37 Testfällen aus [27]. Bei den Instanzen handelt es sich um Problembeschreibungen ohne Kapazitätsgrenzen. Zu jedem Problem sind die Kostenmatrix M und die jeweiligen Fixkosten $c_1, ..., c_n$, die für das Betreiben jedes geöffneten Lagerhauses entstehen, gegeben.

In den Abbildungen C.1 und C.2 sind die durchschnittlichen Beschleunigungsfaktoren a aller Verfahren untereinander ohne und mit Kapazität verglichen. Ist der Faktor positiv mit Wert p, so kann durch die Substituierung p-mal so schnell eine bessere oder gleichwertige Lösung gefunden werden. Ist der Faktor negativ mit Wert $-p$, so wird durch die Substituierung p-mal soviel Zeit benötigt, um eine bessere oder gleichwertige Lösung zu finden. Der Beschleunigungsfaktor ist im positiven Fall durch eine Grünfärbung und im negativen Fall durch eine Rotfärbung der jeweiligen Zelle kenntlich gemacht. Unter der Angabe des Beschleunigungsfaktors ist jeweils die Anzahl der Fälle, in denen eine Beschleunigung auftrat und die Anzahl der Fälle, in denen ein Verlangsamung auftrat, aufgeführt. Der Eintrag in Abbildung C.1 in Zeile T^1 und Spalte R^1 gibt zum Beispiel an, dass die Substituierung, bei der nur die Open-Constraints mittels allgemeiner Regularisrierung substituiert wurden, im Durchschnitt $1,89$ mal so schnell war, wie die Substituierung, bei der nur die Open-Constraints mittels allgemeiner Tabularisierung substituiert wurden. Dabei ist die Regularisierung in 18 Fällen mindestens 5% schneller und in 6 Fällen mindestens 5% langsamer als die Regularisierung. In allen anderen Fällen trat keine signifikante zeitliche Veränderung auf.

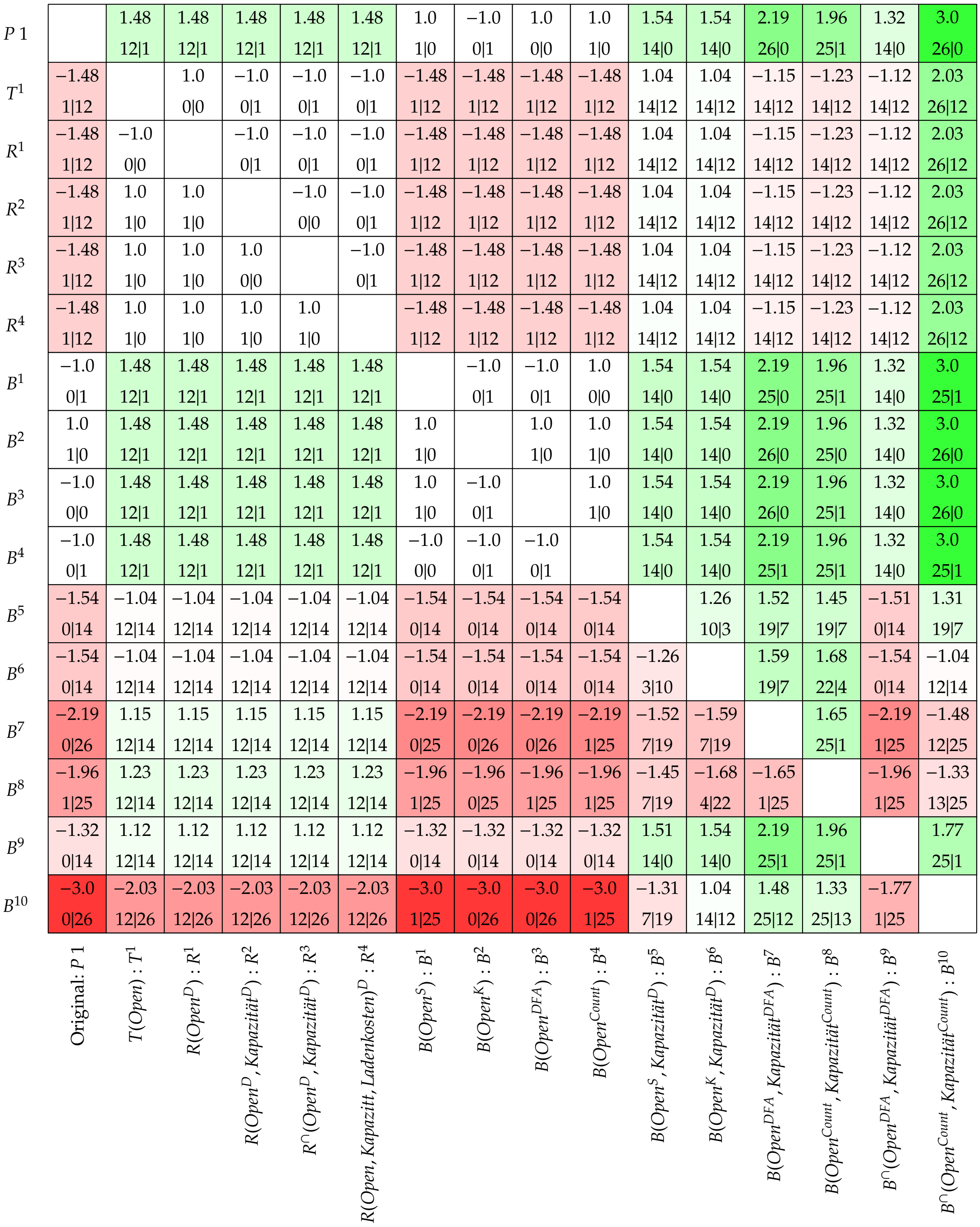
Bezeichner	Anzahl Lagerhäuser	Anzahl Läden
cap0 (CSPlib Beispielproblem)	5	10
cap41	16	50
cap42	16	50
cap43	16	50
cap44 (auch in der CSPlib)	16	50
cap51	16	50
cap61	16	50
cap62	16	50
cap63 (auch in der CSPlib)	16	50
cap64	16	50
cap71 (auch in der CSPlib)	16	50
cap72	16	50
cap73	16	50
cap74	16	50
cap81 (auch in der CSPlib)	25	50
cap82	25	50
cap83	25	50
cap84	25	50
cap91	25	50
cap92	25	50
cap93	25	50
cap94	25	50
cap101 (auch in der CSPlib)	25	50
cap102	25	50
cap103	25	50
cap104	25	50
cap111	50	50
cap112	50	50
cap113	50	50
cap114	50	50
cap121	50	50
cap122	50	50
cap123	50	50
cap124	50	50
cap131 (auch in der CSPlib)	50	50
cap132	50	50
cap133	50	50
cap134	50	50

Tabelle C.1: Namen und Größen der in Abschnitt 6.2.4 verwendeten Warehouse Location-Probleme.



	Original: $P\ 1$	$T(Open) : T^1$	$R(Open^D) : R^1$	$R(Open^D, Kapazität^D) : R^2$	$R^\cap(Open^D, Kapazität^D) : R^3$	$R(Open, Kapazitt, Ladenkosten)^D : R^4$	$B(Open^S) : B^1$	$B(Open^K) : B^2$	$B(Open^{DFA}) : B^3$	$B(Open^{Count}) : B^4$	$B(Open^S, Kapazität^D) : B^5$	$B(Open^K, Kapazität^D) : B^6$	$B(Open^{DFA}, Kapazität^{DFA}) : B^7$	$B(Open^{Count}, Kapazität^{Count}) : B^8$	$B^\cap(Open^{DFA}, Kapazität^{DFA}) : B^9$	$B^\cap(Open^{Count}, Kapazität^{Count}) : B^{10}$
$P\ 1$		−1.46 10|15	1.55 19|4	−1.82 0|23	−1.53 6|20	−1.89 0|26	−3.08 0|26	−3.08 0|26	−2.5 0|26	−3.08 0|26	−3.08 0|26	−2.08 12|26	−3.08 0|26	−3.08 0|26	−1.12 15|23	−1.07 15|23
T^1	1.46 15|10		1.89 18|6	−1.6 0|23	1.01 8|14	−1.96 0|26	−3.08 0|26	−3.08 0|26	−2.51 0|26	−3.08 0|26	−3.08 0|26	−2.08 12|26	−3.08 0|26	−3.08 0|26	1.05 15|18	−1.09 12|26
R^1	−1.55 4|19	−1.89 6|18		−2.09 0|26	−1.74 1|25	−2.08 0|25	−3.08 0|26	−3.08 0|26	−2.5 0|26	−3.08 0|25	−3.08 0|26	−2.08 12|26	−3.08 0|26	−3.08 0|26	−1.18 12|26	−1.11 12|26
R^2	1.82 23|0	1.6 23|0	2.09 26|0		1.55 23|0	−1.14 14|9	−3.06 0|26	−3.06 1|25	−2.35 8|18	−3.06 1|25	−3.06 0|26	−2.07 12|26	−3.06 0|26	−3.06 0|26	1.3 28|9	1.02 25|12
R^3	1.53 20|6	−1.01 14|8	1.74 25|1	−1.55 0|23		−1.88 0|26	−3.08 0|26	−3.08 0|26	−2.5 0|26	−3.08 0|26	−3.08 0|26	−2.08 12|26	−3.08 0|26	−3.08 0|26	−1.15 15|23	−1.09 15|23
R^4	1.89 26|0	1.96 26|0	2.08 25|0	1.14 9|14	1.88 26|0		−3.06 0|26	−3.06 0|26	−2.4 7|18	−3.06 0|25	−3.06 0|26	−2.07 12|26	−3.06 0|26	−3.06 0|26	1.29 28|10	1.01 25|13
B^1	3.08 26|0	3.08 26|0	3.08 26|0	3.06 26|0	3.08 26|0	3.06 26|0		−1.48 1|12	1.04 14|12	−1.48 1|12	3.07 25|0	1.03 12|13	1.05 13|13	−1.48 0|13	2.65 32|6	2.44 28|9
B^2	3.08 26|0	3.08 26|0	3.08 26|0	3.06 25|1	3.08 26|0	3.06 26|0	1.48 12|1		1.54 13|0	1.0 1|0	3.07 25|1	2.51 24|1	2.36 25|1	−1.0 0|1	16.01 37|1	50.91 37|1
B^3	2.5 26|0	2.51 26|0	2.5 26|0	2.35 18|8	2.5 26|0	2.4 18|7	−1.04 12|14	−1.54 0|13		−1.54 0|13	−1.04 12|14	1.63 24|14	−1.18 12|14	−1.54 0|14	10.88 30|8	23.85 30|8
B^4	3.08 26|0	3.08 26|0	3.08 25|0	3.06 25|1	3.08 26|0	3.06 25|0	1.48 12|1	−1.0 0|1	1.54 13|0		3.07 25|1	2.51 24|1	2.36 25|1	−1.0 0|1	16.01 37|1	50.91 37|1
B^5	3.08 26|0	3.08 26|0	3.08 26|0	3.06 26|0	3.08 26|0	3.06 26|0	−3.07 0|25	−3.07 1|25	1.04 14|12	−3.07 1|25		−2.08 12|26	−3.0 0|26	−3.07 0|26	2.61 32|6	1.97 25|12
B^6	2.08 26|12	2.08 26|12	2.08 26|12	2.07 26|12	2.08 26|12	2.07 26|12	−1.03 13|12	−2.51 1|24	−1.63 14|24	−2.51 1|24	2.08 26|12		−1.5 14|24	−2.51 0|24	−1.08 20|18	−1.25 17|21
B^7	3.08 26|0	3.08 26|0	3.08 26|0	3.06 26|0	3.08 26|0	3.06 26|0	−1.05 13|13	−2.36 1|25	1.18 14|12	−2.36 1|25	3.0 26|0	1.5 24|14		−2.36 0|26	3.6 32|6	4.8 35|3
B^8	3.08 26|0	3.08 26|0	3.08 26|0	3.06 26|0	3.08 26|0	3.06 26|0	1.48 13|0	1.0 1|0	1.54 14|0	1.0 1|0	3.07 26|0	2.51 24|0	2.36 26|0		16.01 38|0	50.91 38|0
B^9	1.12 23|15	−1.05 18|15	1.18 26|12	−1.3 9|28	1.15 23|15	−1.29 10|28	−2.65 6|32	−16.01 1|37	−10.88 8|30	−16.01 1|37	−2.61 6|32	1.08 18|20	−3.6 6|32	−16.01 0|38		2.57 18|20
B^{10}	1.07 23|15	1.09 26|12	1.11 26|12	−1.02 12|25	1.09 23|15	−1.01 13|25	−2.44 9|28	−50.91 1|37	−23.85 8|30	−50.91 1|37	−1.97 12|25	1.25 21|17	−4.8 3|35	−50.91 0|38	−2.57 20|18	

Abbildung C.1: Ein vollständiger direkter Vergleich der verschiedenen Substituierungen für das Warehouse Location-Problem ohne Kapazitäten.



	Original: $P\,1$	$T(Open) : T^1$	$R(Open^D) : R^1$	$R(Open^D, Kapazität^D) : R^2$	$R^\cap(Open^D, Kapazität^D) : R^3$	$R(Open, Kapazitt, Ladenkosten)^D : R^4$	$B(Open^S) : B^1$	$B(Open^K) : B^2$	$B(Open^{DFA}) : B^3$	$B(Open^{Count}) : B^4$	$B(Open^S, Kapazität^D) : B^5$	$B(Open^K, Kapazität^D) : B^6$	$B(Open^{DFA}, Kapazität^{DFA}) : B^7$	$B(Open^{Count}, Kapazität^{Count}) : B^8$	$B^\cap(Open^{DFA}, Kapazität^{DFA}) : B^9$	$B^\cap(Open^{Count}, Kapazität^{Count}) : B^{10}$
$P\,1$		1.48 12|1	1.48 12|1	1.48 12|1	1.48 12|1	1.48 12|1	1.0 1|0	−1.0 0|1	1.0 0|0	1.0 1|0	1.54 14|0	1.54 14|0	2.19 26|0	1.96 25|1	1.32 14|0	3.0 26|0
T^1	−1.48 1|12		1.0 0|0	−1.0 0|1	−1.0 0|1	−1.0 0|1	−1.48 1|12	−1.48 1|12	−1.48 1|12	−1.48 1|12	1.04 14|12	1.04 14|12	−1.15 14|12	−1.23 14|12	−1.12 14|12	2.03 26|12
R^1	−1.48 1|12	−1.0 0|0		−1.0 0|1	−1.0 0|1	−1.0 0|1	−1.48 1|12	−1.48 1|12	−1.48 1|12	−1.48 1|12	1.04 14|12	1.04 14|12	−1.15 14|12	−1.23 14|12	−1.12 14|12	2.03 26|12
R^2	−1.48 1|12	1.0 1|0	1.0 1|0		−1.0 0|0	−1.0 0|1	−1.48 1|12	−1.48 1|12	−1.48 1|12	−1.48 1|12	1.04 14|12	1.04 14|12	−1.15 14|12	−1.23 14|12	−1.12 14|12	2.03 26|12
R^3	−1.48 1|12	1.0 1|0	1.0 1|0	1.0 0|0		−1.0 0|1	−1.48 1|12	−1.48 1|12	−1.48 1|12	−1.48 1|12	1.04 14|12	1.04 14|12	−1.15 14|12	−1.23 14|12	−1.12 14|12	2.03 26|12
R^4	−1.48 1|12	1.0 1|0	1.0 1|0	1.0 1|0	1.0 1|0		−1.48 1|12	−1.48 1|12	−1.48 1|12	−1.48 1|12	1.04 14|12	1.04 14|12	−1.15 14|12	−1.23 14|12	−1.12 14|12	2.03 26|12
B^1	−1.0 0|1	1.48 12|1	1.48 12|1	1.48 12|1	1.48 12|1	1.48 12|1		−1.0 0|1	−1.0 0|1	1.0 0|0	1.54 14|0	1.54 14|0	2.19 25|0	1.96 25|1	1.32 14|0	3.0 25|1
B^2	1.0 1|0	1.48 12|1	1.48 12|1	1.48 12|1	1.48 12|1	1.48 12|1	1.0 1|0		1.0 1|0	1.0 1|0	1.54 14|0	1.54 14|0	2.19 26|0	1.96 25|0	1.32 14|0	3.0 26|0
B^3	−1.0 0|0	1.48 12|1	1.48 12|1	1.48 12|1	1.48 12|1	1.48 12|1	1.0 1|0	−1.0 0|1		1.0 1|0	1.54 14|0	1.54 14|0	2.19 26|0	1.96 25|1	1.32 14|0	3.0 26|0
B^4	−1.0 0|1	1.48 12|1	1.48 12|1	1.48 12|1	1.48 12|1	1.48 12|1	−1.0 0|0	−1.0 0|1	−1.0 0|1		1.54 14|0	1.54 14|0	2.19 25|1	1.96 25|1	1.32 14|0	3.0 25|1
B^5	−1.54 0|14	−1.04 12|14	−1.04 12|14	−1.04 12|14	−1.04 12|14	−1.04 12|14	−1.54 0|14	−1.54 0|14	−1.54 0|14	−1.54 0|14		1.26 10|3	1.52 19|7	1.45 19|7	−1.51 0|14	1.31 19|7
B^6	−1.54 0|14	−1.04 12|14	−1.04 12|14	−1.04 12|14	−1.04 12|14	−1.04 12|14	−1.54 0|14	−1.54 0|14	−1.54 0|14	−1.54 0|14	−1.26 3|10		1.59 19|7	1.68 22|4	−1.54 0|14	−1.04 12|14
B^7	−2.19 0|26	1.15 12|14	1.15 12|14	1.15 12|14	1.15 12|14	1.15 12|14	−2.19 0|25	−2.19 0|26	−2.19 0|26	−2.19 1|25	−1.52 7|19	−1.59 7|19		1.65 25|1	−2.19 1|25	−1.48 12|25
B^8	−1.96 1|25	1.23 12|14	1.23 12|14	1.23 12|14	1.23 12|14	1.23 12|14	−1.96 1|25	−1.96 0|25	−1.96 1|25	−1.96 1|25	−1.45 7|19	−1.68 4|22	−1.65 1|25		−1.96 1|25	−1.33 13|25
B^9	−1.32 0|14	1.12 12|14	1.12 12|14	1.12 12|14	1.12 12|14	1.12 12|14	−1.32 0|14	−1.32 0|14	−1.32 0|14	−1.32 0|14	1.51 14|0	1.54 14|0	2.19 25|1	1.96 25|1		1.77 25|1
B^{10}	−3.0 0|26	−2.03 12|26	−2.03 12|26	−2.03 12|26	−2.03 12|26	−2.03 12|26	−3.0 1|25	−3.0 0|26	−3.0 0|26	−3.0 1|25	−1.31 7|19	1.04 14|12	1.48 25|12	1.33 25|13	−1.77 1|25	

Abbildung C.2: Ein vollständiger direkter Vergleich der verschiedenen Substituierungen für das Warehouse Location-Problem mit Kapazitäten.

D Die Bedeutung der Kompaktheit boolescher *Scalar*-CSPs

Im Folgenden sind Diagramme dargestellt, die für alle Anwendungsbeispiele aus den Abschnitten 6.2.1 bis 6.2.4 die Beschleunigungsfaktoren a der jeweiligen booleschen Skalarisierungen ins Verhältnis zu den Vergrößerungsfaktoren V^c und V^v aus Abschnitt 6.2.5 setzen. Die Ordinate stellt dabei jeweils den Beschleunigungsfaktor und die Abszisse den jeweiligen Vergrößerungsfaktor dar. Angelegt wurden die Diagramme jeweils für große Schichtplanungs-, Black Hole-, Knight Tour- und Warehouse Location-Probleme ohne und mit Kapazitäten. Kleine Schichtplanungsprobleme wurden, da sie schnell gelöst werden können und in dieser Arbeit das Lösen großer Probleme im Vordergrund steht, vernachlässigt.

D.1 Große Schichtplanungsprobleme

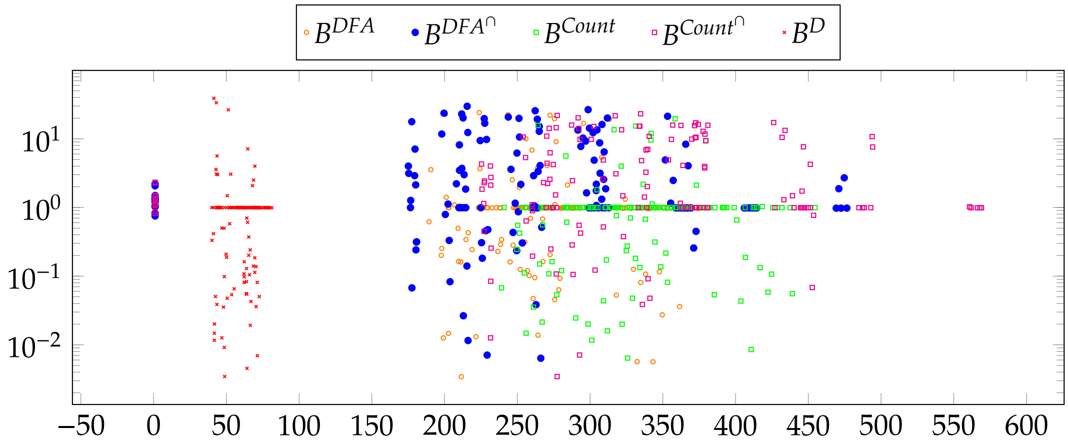


Abbildung D.1: Der Beschleunigungsfaktor a (Ordinate) in Abhängigkeit vom Vergrößerungsfaktor V^v der Anzahl an *Variablen* (Abszisse) bei der booleschen Skalarisierung großer Schichtplanungsprobleme.

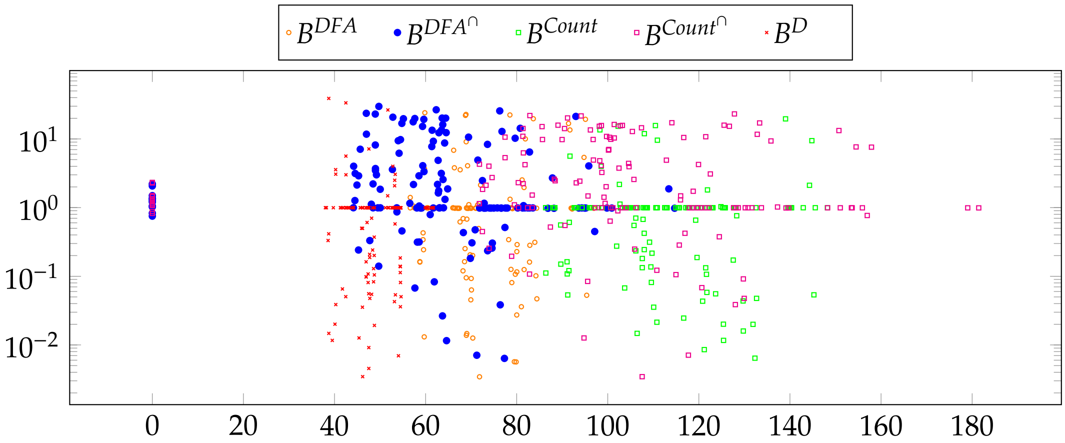


Abbildung D.2: Der Beschleunigungsfaktor *a* (Ordinate) in Abhängigkeit vom Vergrößerungsfaktor V^c der Anzahl an *Constraints* (Abszisse) bei der booleschen Skalarisierung großer Schichtplanungsprobleme.

D.2 Black Hole-Probleme

Da die 50 verschiedenen Black Hole-Probleme aus Abschnitt 6.2.2 alle den gleichen Aufbau haben (52 Karten, 17 Fächer), hat das ursprüngliche CSP immer gleich viele Variablen und Constraints, wobei sich innerhalb der *Scalar*-Constraints nur die Konstanten ändern, Relationssymbol und Variablen aber in der Regel gleich bleiben. Daher haben die CSPs verschiedener Probleminstanzen, die durch die gleiche Substituierungsart erzeugt wurden, fast die identische Anzahl an Variablen und Constraints. Somit sind in den folgenden beiden Abbildungen die Vergrößerungsfaktoren V^v und V^c für jeweils eine Substituierungsart nahezu identisch.

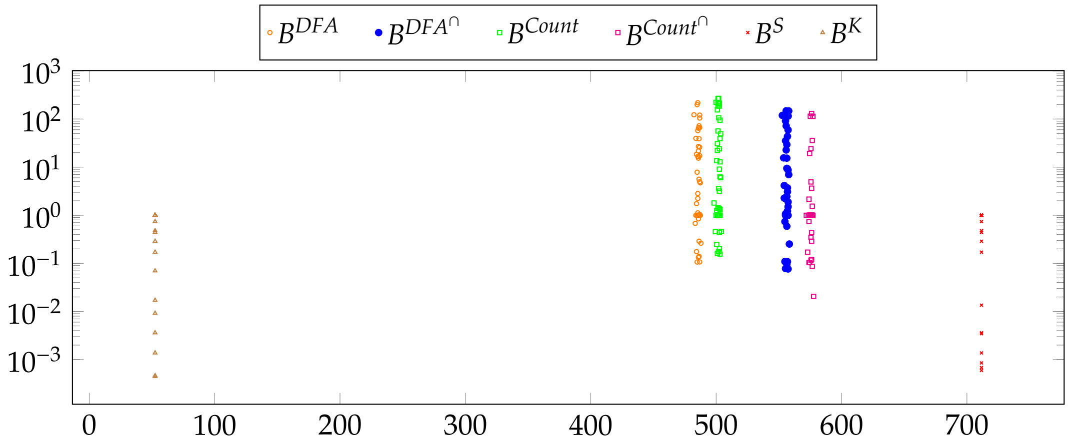


Abbildung D.3: Der Beschleunigungsfaktor a (Ordinate) in Abhängigkeit vom Vergrößerungsfaktor V^v der Anzahl an *Variablen* (Abszisse) bei der booleschen Skalarisierung von Black Hole-Problemen.

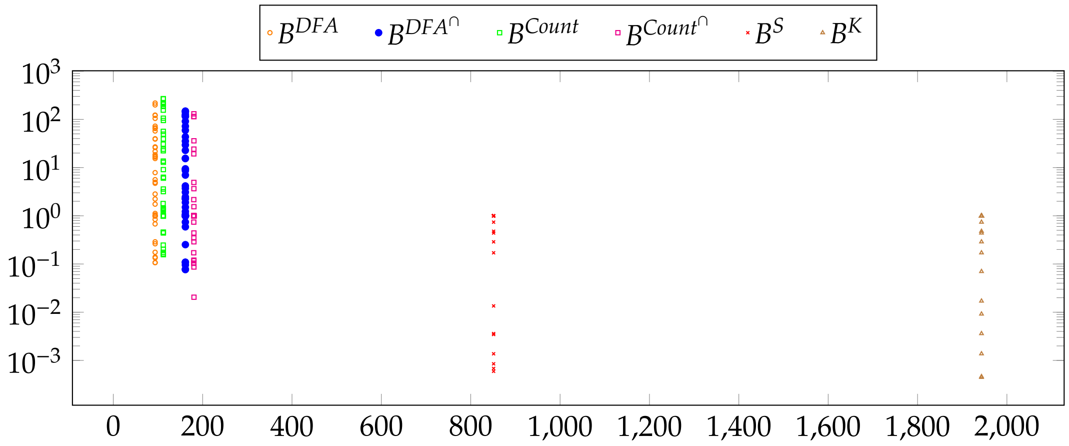


Abbildung D.4: Der Beschleunigungsfaktor a (Ordinate) in Abhängigkeit vom Vergrößerungsfaktor V^c der Anzahl an *Constraints* (Abszisse) bei der booleschen Skalarisierung von Black Hole-Problemen.

D.3 Knight Tour-Probleme

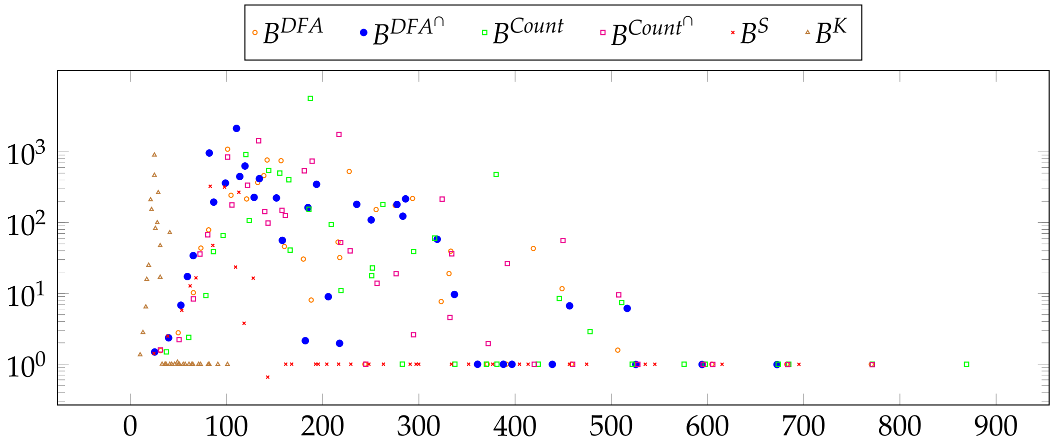


Abbildung D.5: Der Beschleunigungsfaktor *a* (Ordinate) in Abhängigkeit vom Vergrößerungsfaktor V^v der Anzahl an *Variablen* (Abszisse) bei der booleschen Skalarisierung von Knight Tour-Problemen.

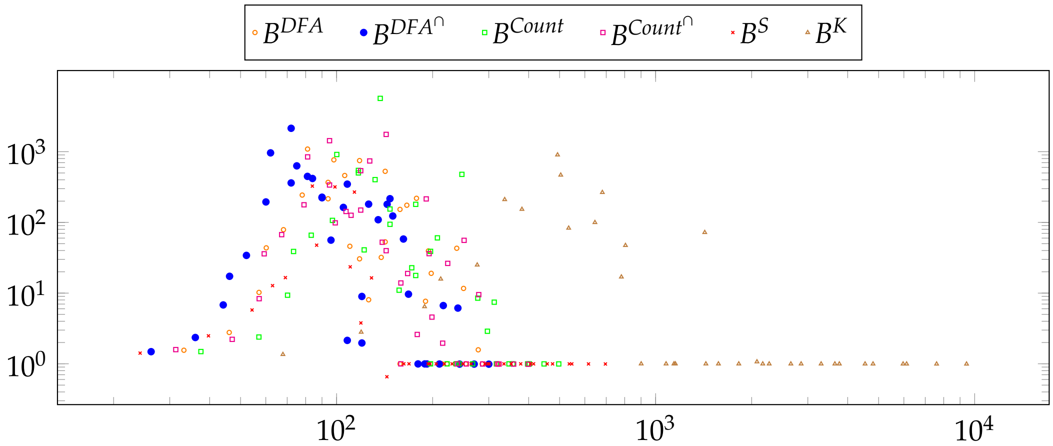


Abbildung D.6: Der Beschleunigungsfaktor *a* (Ordinate) in Abhängigkeit vom Vergrößerungsfaktor V^c der Anzahl an *Constraints* (Abszisse) bei der booleschen Skalarisierung von Knight Tour-Problemen.

D.4 Warehouse Location-Probleme ohne Kapazitäten

Bei den Diagrammen in den folgenden Abbildungen D.7 bis D.10 fehlen Angaben zu B^1 bis B^4. Diese Verfahren wurden ausgelassen, da sie nur in sehr wenigen Fällen eine Lösung fanden und für die Diagramme keine zusätzliche Information liefern.

In den nachfolgenden Abbildungen sind jeweils 38 Wertepaare pro Substituierungsart dargestellt. Aufgrund der großen Ähnlichkeit der verschiedenen Warehouse Location-Probleme sind jeweils nur wenige verschiedene Punkte pro Substituierungsart in den Abbildungen zu erkennen. Es handelt sich dabei um Überlagerungen durch mehrere Instanzen des Warehouse Location-Problems mit nahezu identischen Vergrößerungs- und Beschleunigungsfaktoren.

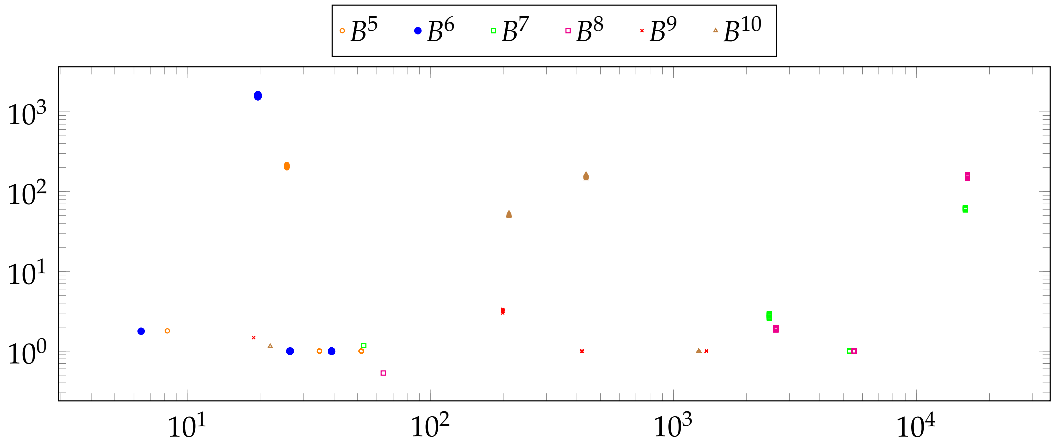


Abbildung D.7: Der Beschleunigungsfaktor *a* (Ordinate) in Abhängigkeit vom Vergrößerungsfaktor V^v der Anzahl an *Variablen* (Abszisse) bei der booleschen Skalarisierung von Warehouse Location-Problemen ohne Kapazitäten.

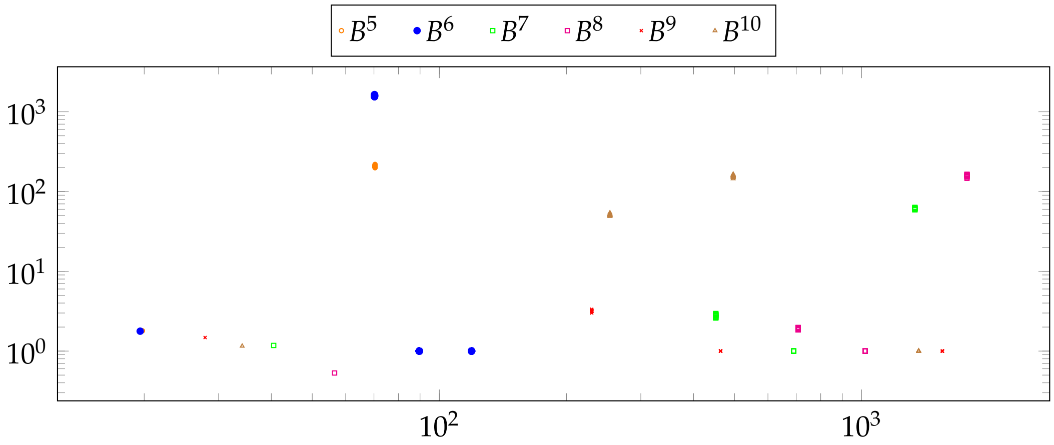


Abbildung D.8: Der Beschleunigungsfaktor a (Ordinate) in Abhängigkeit vom Vergrößerungsfaktor V^c der Anzahl an *Constraints* (Abszisse) bei der booleschen Skalarisierung von Warehouse Location-Problemen mit Kapazitäten.

D.5 Warehouse Location-Probleme mit Kapazitäten

Es gelten die gleichen Anmerkungen wie zu Anhang D.4.

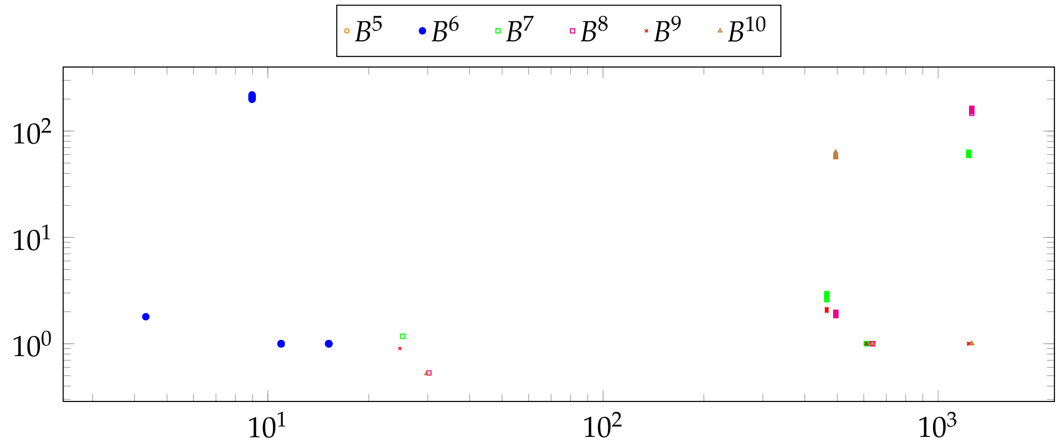


Abbildung D.9: Der Beschleunigungsfaktor a (Ordinate) in Abhängigkeit vom Vergrößerungsfaktor V^v der Anzahl an *Variablen* (Abszisse) bei der booleschen Skalarisierung von Warehouse Location-Problemen mit Kapazitäten.

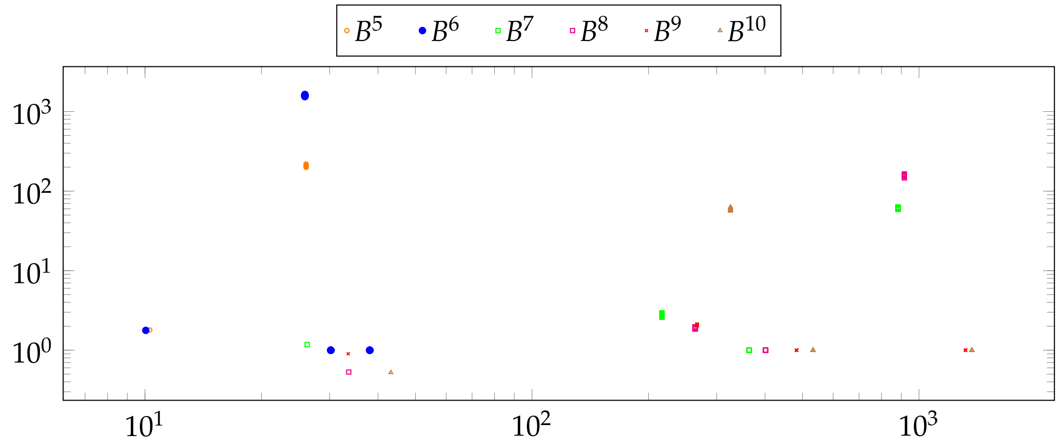


Abbildung D.10: Der Beschleunigungsfaktor a (Ordinate) in Abhängigkeit vom Vergrößerungsfaktor V^c der Anzahl an *Constraints* (Abszisse) bei der booleschen Skalarisierung von Warehouse Location-Problemen mit Kapazitäten.

Literaturverzeichnis

[1] Global Constraint Catalog. `http://sofdem.github.io/gccat/`, 07.04.2015

[2] Google LLC, Google OR-Tools, 2019, `lastvisited2019-11-22,https://developers.google.com/optimization/`

[3] Jacop solver 4.4. `http://jacop.osolpro.com/`, `http://jacop.osolpro.com/`

[4] Pbsugar: A sat-based pseudo-boolean solver 1.1.2. `https://cspsat.gitlab.io/pbsugar/`, `https://cspsat.gitlab.io/pbsugar/`

[5] Sat4j solver 2.3.4. `https://www.sat4j.org/`, `https://www.sat4j.org/`

[6] Arbeitszeitgesetz vom 6.Juni 1994 (BGBl.IS.1170,1171), zul. geänd. durch Art.3 Abs.6 des Ges.v.20.April 2013 (BGBl.IS.868). `http://www.gesetze-im-internet.de/arbzg/BJNR117100994.html` (1994), last visited 2017-05-10

[7] Bundesanstalt für Arbeitsschutz und Arbeitsmedizin (BAuA) - Gestaltung von Nacht- und Schichtarbeit. `http://www.baua.de/de/Informationen-fuer-die-Praxis/Handlungshilfen-und-Praxisbeispiele/Arbeitszeitgestaltung/Nacht-%20und%20Schichtarbeit.html` (2013), last visited 2017-05-10

[8] Achterberg, T., Wunderling, R.: Mixed Integer Programming: Analyzing 12 Years of Progress, pp. 449–481 (01 2013)

[9] Aiken, A.: Set constraints: Results, applications and future directions. In: Borning, A. (ed.) Principles and Practice of Constraint Programming. pp. 326–335. Springer Berlin Heidelberg, Berlin, Heidelberg (1994)

[10] Akgün, Ö., Gent, I.P., Jefferson, C., Miguel, I., Nightingale, P., Salamon, A.Z.: Automatic discovery and exploitation of promising subproblems for tabulation. In: Hooker [75], pp. 3–12, `https://doi.org/10.1007/978-3-319-98334-9_1`

[11] Amadini, R., Gange, G., Stuckey, P.J., Tack, G.: A novel approach to string constraint solving. In: Beck, J.C. (ed.) Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10416, pp. 3–20. Springer (2017), `https://doi.org/10.1007/978-3-319-66158-2_1`

[12] Amilhastre, J., Fargier, H., Niveau, A., Pralet, C.: Compiling csps: A complexity map of (non-deterministic) multivalued decision diagrams. Int. J. Artif. Intell. Tools **23**(4) (2014), `https://doi.org/10.1142/S021821301460015X`

[13] Ansótegui, C., Manyà, F.: Mapping problems with finite-domain variables into problems with boolean variables. In: SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings (2004), `http://www.satisfiability.org/SAT04/programme/53.pdf`

[14] Apt, K.: Constraint satisfaction problems: examples. In: [16] (2003), chapter 2.

[15] Apt, K.: Issues in constraint programming. In: [16] (2003), chapter 9.

[16] Apt, K.: Principles of Constraint Programming. Cambridge University Press, New York, NY, USA (2003)

[17] Apt, K., Wallace, M.: Constraint Logic Programming using Eclipse. Cambridge University Press (2007)

[18] Argelich, J., Cabiscol, A., Lynce, I., Manyà, F.: Encoding max-csp into partial max-sat. In: 38th IEEE International Symposium on Multiple-Valued Logic (ISMVL 2008), 22-23 May 2008, Dallas, Texas, USA. pp. 106–111. IEEE Computer Society (2008), `https://doi.org/10.1109/ISMVL.2008.22`

[19] Asratian, A.S., Denley, T.M.J., Häggkvist, R.: Bipartite Graphs and Their Applications. Cambridge University Press, New York, NY, USA (1998)

[20] Bacchus, F., van Beek, P.: On the conversion between non-binary and binary constraint satisfaction problems. In: Mostow, J., Rich, C. (eds.) Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, July 26-30, 1998, Madison, Wisconsin, USA. pp. 310–318. AAAI Press / The MIT Press (1998), `http://www.aaai.org/Library/AAAI/1998/aaai98-044.php`

[21] Bacchus, F., Chen, X., van Beek, P., Walsh, T.: Binary vs. non-binary constraints. Artif. Intell. **140**(1/2), 1–37 (2002), `https://doi.org/10.1016/S0004-3702(02)00210-2`

[22] Bailleux, O., Boufkhad, Y.: Efficient CNF encoding of boolean cardinality constraints. In: Rossi, F. (ed.) Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2833, pp. 108–122. Springer (2003), `https://doi.org/10.1007/978-3-540-45193-8_8`

[23] Bailleux, O., Boufkhad, Y., Roussel, O.: New encodings of pseudo-boolean constraints into cnf. In: Kullmann, O. (ed.) Theory and Applications of Satisfiability

Testing - SAT 2009. pp. 181–194. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)

[24] Barták, R., Toropila, D.: Reformulating constraint models for classical planning. In: Proceedings of the Twenty-First International Florida Artificial Intelligence Research Society Conference, May 15-17, 2008, Coconut Grove, Florida, USA. pp. 525–530 (2008), `http://www.aaai.org/Library/FLAIRS/2008/flairs08-121.php`

[25] Bartàk, R.: On-line guide to Constraint Programming. `http://kti.ms.mff.cuni.cz/~bartak/constraints/` (1998), 15.06.2015

[26] Bayer, K.M., Michalowski, M., Choueiry, B.Y., Knoblock, C.A.: Reformulating constraint satisfaction problems to improve scalability. In: Abstraction, Reformulation, and Approximation, 7th International Symposium, SARA 2007, Whistler, Canada, July 18-21, 2007, Proceedings. pp. 64–79 (2007), `https://doi.org/10.1007/978-3-540-73580-9_8`

[27] Beasley, J.: Lagrangean heuristics for location problems. European Journal of Operational Research **65**(3), 383–399 (1993), `https://www.sciencedirect.com/science/article/pii/0377221793901187`

[28] van Beek, P.: Backtracking Search Algorithms. In: [136], First edn. (2006), chapter 4

[29] van Beek, P.: Backtracking search algorithms. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming, Foundations of Artificial Intelligence, vol. 2, pp. 85–134. Elsevier (2006), `https://doi.org/10.1016/S1574-6526(06)80008-8`

[30] Beldiceanu, N., Carlsson, M., Flener, P., Pearson, J.: On the reification of global constraints. Constraints An Int. J. **18**(1), 1–6 (2013), `https://doi.org/10.1007/s10601-012-9132-0`

[31] Bellman, R.: The theory of dynamic programming. Bull. Amer. Math. Soc. **60**(6), 503–515 (11 1954), `https://projecteuclid.org:443/euclid.bams/1183519147`

[32] Bergman, D., Ciré, A.A., van Hoeve, W.J., Hooker, J.N.: Variable ordering for the application of bdds to the maximum independent set problem. In: Beldiceanu, N., Jussien, N., Pinson, E. (eds.) Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems - 9th International Conference, CPAIOR 2012, Nantes, France, May 28 - June1, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7298, pp. 34–49. Springer (2012), `https://doi.org/10.1007/978-3-642-29828-8_3`

[33] Bessiere, C., Hebrard, E., Hnich, B., Walsh, T.: The complexity of reasoning with global constraints. Constraints An Int. J. **12**(2), 239–259 (2007), `https://doi.org/10.1007/s10601-006-9007-3`

[34] Bessière, C., Régin, J.: Arc consistency for general constraint networks: Preliminary results. In: Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes. pp. 398–404. Morgan Kaufmann (1997), `http://ijcai.org/proceedings/1997-1`

[35] Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: de Mántaras, R.L., Saitta, L. (eds.) Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004. pp. 146–150. IOS Press (2004)

[36] Breiman, L.: Random forests. Mach. Learn. **45**(1), 5–32 (2001), `https://doi.org/10.1023/A:1010933404324`

[37] Brey, M.: Konfiguration und Gestaltung mit Constraintsystemen. Ph.D. thesis, Braunschweig University of Technology, Germany (2003), `http://d-nb.info/966588258`

[38] Carlsson, M., Mildner, P.: Sicstus prolog - the first 25 years. Theory and Practice of Logic Programming – TPLP **12**(1-2), 35–66 (2012)

[39] Cheng, B.M.W., Lee, J.H., Wu, J.C.K.: Speeding up constraint propagation by redundant modeling. In: Freuder, E.C. (ed.) Proceedings of the Second International Conference on Principles and Practice of Constraint Programming, Cambridge, Massachusetts, USA, August 19-22, 1996. Lecture Notes in Computer Science, vol. 1118, pp. 91–103. Springer (1996), `https://doi.org/10.1007/3-540-61551-2_68`

[40] Cheng, K.C.K., Yap, R.H.C.: Ad-hoc global constraints for life. In: van Beek, P. (ed.) Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3709, pp. 182–195. Springer (2005), `https://doi.org/10.1007/11564751_16`

[41] Choi, C.W., Lee, J.H., Stuckey, P.J.: Propagation redundancy for permutation channels. In: Gottlob, G., Walsh, T. (eds.) IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003. pp. 1370–1371. Morgan Kaufmann (2003), `http://ijcai.org/Proceedings/03/Papers/203.pdf`

[42] Christian Schulte, Mikael Lagerkvist, G.T.: Gecode 6.2.0, 2019, https://www.gecode.org/, last visited 2019-11-22 (2019), `https://www.gecode.org/,lastvisited2019-11-22`

[43] Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM **5**(7), 394–397 (Jul 1962), `https://doi.org/10.1145/368273.368557`

[44] Dechter, R.: Consistency-enforcing and constraint propagation. In: Constraint processing [46], chap. 3, pp. 51–84

[45] Dechter, R.: Constraint networks. In: Constraint processing [46], chap. 2, pp. 25–49

[46] Dechter, R.: Constraint processing. Elsevier Morgan Kaufmann (2003)

[47] Dechter, R.: General search strategies: Look-back. In: Constraint processing [46], chap. 6, pp. 151–190

[48] Dechter, R., Pearl, J.: Tree clustering for constraint networks. Artif. Intell. **38**(3), 353–366 (1989), `https://doi.org/10.1016/0004-3702(89)90037-4`

[49] Dekker, J.J., Björdal, G., Carlsson, M., Flener, P., Monette, J.: Auto-tabling for subproblem presolving in minizinc. Constraints An Int. J. **22**(4), 512–529 (2017), `https://doi.org/10.1007/s10601-017-9270-5`

[50] Demeulenaere, J., Hartert, R., Lecoutre, C., Perez, G., Perron, L., Régin, J., Schaus, P.: Compact-table: Efficiently filtering table constraints with reversible sparse bit-sets. In: Rueher, M. (ed.) Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9892, pp. 207–223. Springer (2016), `https://doi.org/10.1007/978-3-319-44953-1_14`

[51] Domschke, W., Drexl, A.: Einführung in Operations Research. Springer-Lehrbuch, Springer, Berlin [u.a.], 6. überarbeitete und erweiterte auflage edn. (2005), `http://gso.gbv.de/DB=2.1/CMD?ACT=SRCHA&SRT=YOP&IKT=1016&TRM=ppn+394637046&sourceid=fbw_bibsonomy`

[52] Eén, N., Sörensson, N.: An extensible sat-solver. In: Giunchiglia, E., Tacchella, A. (eds.) Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers. Lecture Notes in Computer Science, vol. 2919, pp. 502–518. Springer (2003), `https://doi.org/10.1007/978-3-540-24605-3_37`

[53] Ernst, M.D., Millstein, T.D., Weld, D.S.: Automatic sat-compilation of planning problems. In: Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes. pp. 1169–1177. Morgan Kaufmann (1997), `http://ijcai.org/Proceedings/97-2/Papers/055.pdf`

[54] Faltings, B.: Distributed constraint programming. In: Handbook of Constraint Programming, pp. 699–729 (2006), `https://doi.org/10.1016/S1574-6526(06)80024-6`

[55] Gaschnig, J.: Experimental case studies of backtrack vs. Waltz-type vs. new

algorithms for satisficing assignment problems. In Proceedings of the Second Canadian Conference on Artificial Intelligence, pages 268–277, Toronto, 1978.

[56] Gavanelli, M.: The log-support encoding of CSP into SAT. In: Bessiere, C. (ed.) Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4741, pp. 815–822. Springer (2007), `https://doi.org/10.1007/978-3-540-74970-7_59`

[57] Gelder, A.V.: Another look at graph coloring via propositional satisfiability. Discret. Appl. Math. **156**(2), 230–243 (2008), `https://doi.org/10.1016/j.dam.2006.07.016`

[58] Gendreau, M., Potvin, J.Y.: Handbook of Metaheuristics. Springer Publishing Company, Incorporated, 2. edn. (2010)

[59] Gent, I.P.: Arc consistency in SAT. In: van Harmelen, F. (ed.) Proceedings of the 15th Eureopean Conference on Artificial Intelligence, ECAI'2002, Lyon, France, July 2002. pp. 121–125. IOS Press (2002)

[60] Gent, I.P., Harvey, W., Kelsey, T.: Groups and constraints: Symmetry breaking during search. In: Hentenryck, P.V. (ed.) Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings. Lecture Notes in Computer Science, vol. 2470, pp. 415–430. Springer (2002), `https://doi.org/10.1007/3-540-46135-3_28`

[61] Gent, I.P., Jefferson, C., Kelsey, T., Lynce, I., Miguel, I., Nightingale, P., Smith, B.M., Tarim, A.: Search in the patience game 'black hole'. AI Commun. **20**(3), 211–226 (2007), `http://content.iospress.com/articles/ai-communications/aic405`

[62] Gent, I.P., Jefferson, C., Miguel, I., Nightingale, P.: Data structures for generalised arc consistency for extensional constraints. In: Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22-26, 2007, Vancouver, British Columbia, Canada. pp. 191–197. AAAI Press (2007), `http://www.aaai.org/Library/AAAI/2007/aaai07-029.php`

[63] GENT, I.P., MIGUEL, I., NIGHTINGALE, P., MCCREESH, C., PROSSER, P., MOORE, N.C.A., UNSWORTH, C.: A review of literature on parallel constraint solving. Theory and Practice of Logic Programming **18**(5-6), 725–758 (2018)

[64] Getoor, L., Ottosson, G., Fromherz, M.P.J., Carlson, B.: Effective redundant constraints for online scheduling. In: Kuipers, B., Webber, B.L. (eds.) Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, July 27-31, 1997, Providence, Rhode Island, USA. pp. 302–307. AAAI Press / The MIT Press (1997), `http://www.aaai.org/Library/AAAI/1997/aaai97-047.php`

[65] Goldsztejn, A., Jermann, C., de Angulo, V.R., Torras, C.: Symmetry breaking in numeric constraint problems. In: Lee, J.H. (ed.) Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6876, pp. 317–324. Springer (2011), `https://doi.org/10.1007/978-3-642-23786-7_25`

[66] Hamadi, Y.: Optimal distributed arc-consistency. Constraints **7**(3-4), 367–385 (2002), `https://doi.org/10.1023/A:1020594125144`

[67] Hentenryck, P.V.: Constraint satisfaction in logic programming. Logic programming, MIT Press (1989)

[68] Hentenryck, P.V.: The OPL Optimization Programming Language. The MIT Press (1999)

[69] Hnich, B.: CSPLib problem 034: Warehouse location problem. `http://www.csplib.org/Problems/prob034`, last visited 2019-03-28

[70] van Hoeve, W.J., Katriel, I.: Global Constraints. In: [136], First edn. (2006), chapter 6

[71] Hofstedt, P.: Constraint-based object-oriented programming. IEEE Software **27**(5), 53–56 (2010)

[72] Hofstedt, P.: Multiparadigm Constraint Programming Languages. Springer (2011)

[73] Hofstedt, P., Löffler, S.: Constraints. In: Handbuch der Künstlichen Intelligenz. 6. Auflage, pp. 713–754. De Gruyter, Berlin, Boston (2021), `https://doi.org/10.1515/9783110659948-016`

[74] Hofstedt, P., Wolf, A.: Einführung in die Constraint-Programmierung: Grundlagen, Methoden, Sprachen, Anwendungen. Springer (01 2007)

[75] Hooker, J.N. (ed.): Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings, Lecture Notes in Computer Science, vol. 11008. Springer (2018), `https://doi.org/10.1007/978-3-319-98334-9`

[76] Hoos, H.H.: Sat-encodings, search space structure, and local search performance. In: Dean, T. (ed.) Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99, Stockholm, Sweden, July 31 - August 6, 1999. 2 Volumes, 1450 pages. pp. 296–303. Morgan Kaufmann (1999), `http://ijcai.org/Proceedings/99-1/Papers/044.pdf`

[77] Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to automata theory, languages, and computation - international edition (2. ed). Addison-Wesley (2003)

[78] Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley (1979)

[79] Huberman, B.A., Lukose, R.M., Hogg, T.: An economics approach to hard computational problems. Science **275**(5296), 51–54 (1997), `https://www.science.org/doi/abs/10.1126/science.275.5296.51`

[80] IBM: Ibm ilog cplex optimization studio. `https://www.ibm.com/de-de/products/ilog-cplex-optimization-studio` (2019), zuletzt besucht 2019-08-21

[81] Iwama, K., Miyazaki, S.: Sat-varible complexity of hard combinatorial problems. In: Pehrson, B., Simon, I. (eds.) Technology and Foundations - Information Processing '94, Volume 1, Proceedings of the IFIP 13th World Computer Congress, Hamburg, Germany, 28 August - 2 September, 1994. IFIP Transactions, vol. A-51, pp. 253–258. North-Holland (1994)

[82] Jaffar, J., Michaylov, S.: Methodology and implementation of a CLP system. In: Lassez, J. (ed.) Logic Programming, Proceedings of the Fourth International Conference, Melbourne, Victoria, Australia, May 25-29, 1987 (2 Volumes). pp. 196–218. MIT Press (1987)

[83] Jefferson, C., Özgür Akgün: Csplib: A problem library for constraints. `https://www.csplib.org/`, last visited 2021-08-10

[84] Jefferson, C., Petrie, K.E.: Automatic generation of constraints for partial symmetry breaking. In: Lee, J.H. (ed.) Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6876, pp. 729–743. Springer (2011), `https://doi.org/10.1007/978-3-642-23786-7_55`

[85] Jégou, P.: Decomposition of domains based on the micro-structure of finite constraint-satisfaction problems. In: Fikes, R., Lehnert, W.G. (eds.) Proceedings of the 11th National Conference on Artificial Intelligence. Washington, DC, USA, July 11-15, 1993. pp. 731–736. AAAI Press / The MIT Press (1993), `http://www.aaai.org/Library/AAAI/1993/aaai93-109.php`

[86] Kiziltan, Z.: Symmetry breaking ordering constraints. In: Rossi, F. (ed.) Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2833, p. 979. Springer (2003), `https://doi.org/10.1007/978-3-540-45193-8_103`

[87] Lecoutre, C.: Optimization of simple tabular reduction for table constraints. In: Principles and Practice of Constraint Programming, 14th International Conference,

CP 2008, Sydney, Australia, September 14-18, 2008. Proceedings. pp. 128–143 (2008), `https://doi.org/10.1007/978-3-540-85958-1_9`

[88] Lecoutre, C.: STR2: optimized simple tabular reduction for table constraints. Constraints An Int. J. **16**(4), 341–371 (2011), `https://doi.org/10.1007/s10601-011-9107-6`

[89] Lecoutre, C., Likitvivatanavong, C., Yap, R.H.C.: STR3: A path-optimal filtering algorithm for table constraints. Artif. Intell. **220**, 1–27 (2015), `https://doi.org/10.1016/j.artint.2014.12.002`

[90] Lecoutre, C., Paris, N., Roussel, O., Tabary, S.: Propagating soft table constraints. In: Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings. pp. 390–405 (2012), `https://doi.org/10.1007/978-3-642-33558-7_30`

[91] Lecoutre, C., Szymanek, R.: Generalized arc consistency for positive table constraints. In: Benhamou, F. (ed.) Principles and Practice of Constraint Programming - CP 2006, 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4204, pp. 284–298. Springer (2006), `https://doi.org/10.1007/11889205_22`

[92] Lee, K., Takefuji, Y.: Finding knight's tours on an *M*×*N* chessboard with *O(MN)* hysteresis mcculloch-pitts neurons. IEEE Trans. Syst. Man Cybern. Syst. **24**(2), 300–306 (1994), `https://doi.org/10.1109/21.281427`

[93] Lhomme, O., Régin, J.: A fast arc consistency algorithm for n-ary constraints. In: Veloso, M.M., Kambhampati, S. (eds.) Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA. pp. 405–410. AAAI Press / The MIT Press (2005), `http://www.aaai.org/Library/AAAI/2005/aaai05-065.php`

[94] Löffler, S., Becker, I., Hofstedt, P.: Ml-based decision support for CSP modelling with regular membership and table constraints. In: Rocha, A.P., Steels, L., van den Herik, H.J. (eds.) Proceedings of the 13th International Conference on Agents and Artificial Intelligence, ICAART 2021, Volume 2, Online Streaming, February 4-6, 2021. pp. 974–981. SCITEPRESS (2021), `https://doi.org/10.5220/0010299109740981`

[95] Löffler, S., Liu, K., Hofstedt, P.: The regularization of csps for rostering, planning and resource management problems. In: Artificial Intelligence Applications and Innovations - 14th IFIP WG 12.5 International Conference, AIAI 2018, Rhodes, Greece, May 25-27, 2018, Proceedings. pp. 209–218 (2018), `https://doi.org/10.1007/978-3-319-92007-8_18`

[96] Löffler, S., Liu, K., Hofstedt, P.: Decomposing constraint satisfaction problems by

means of meta constraint satisfaction optimization problems. In: Rocha, A.P., Steels, L., van den Herik, H.J. (eds.) Proceedings of the 11th International Conference on Agents and Artificial Intelligence, ICAART 2019, Volume 2, Prague, Czech Republic, February 19-21, 2019. pp. 755–761. SciTePress (2019), `https://doi.org/10.5220/0007455907550761`

[97] Löffler, S., Liu, K., Hofstedt, P.: A meta constraint satisfaction optimization problem for the optimization of regular constraint satisfaction problems. In: Rocha, A.P., Steels, L., van den Herik, H.J. (eds.) Proceedings of the 11th International Conference on Agents and Artificial Intelligence, ICAART 2019, Volume 2, Prague, Czech Republic, February 19-21, 2019. pp. 435–442. SciTePress (2019), `https://doi.org/10.5220/0007260204350442`

[98] Löffler, S., Liu, K., Hofstedt, P.: The regularization of small sub-constraint satisfaction problems. CoRR **abs/1908.05907** (2019), `http://arxiv.org/abs/1908.05907`

[99] Löffler, S., Liu, K., Hofstedt, P.: An introduction of fd-complete constraints. In: Maglogiannis, I., Iliadis, L., Pimenidis, E. (eds.) Artificial Intelligence Applications and Innovations - 16th IFIP WG 12.5 International Conference, AIAI 2020, Neos Marmaras, Greece, June 5-7, 2020, Proceedings, Part II. IFIP Advances in Information and Communication Technology, vol. 584, pp. 27–38. Springer (2020), `https://doi.org/10.1007/978-3-030-49186-4_3`

[100] Löffler, S., Liu, K., Hofstedt, P.: Optimizing constraint satisfaction problems by regularization for the sample case of the warehouse location problem. In: Schmid, U., Klügl, F., Wolter, D. (eds.) KI 2020: Advances in Artificial Intelligence - 43rd German Conference on AI, Bamberg, Germany, September 21-25, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12325, pp. 297–304. Springer (2020), `https://doi.org/10.1007/978-3-030-58285-2_26`

[101] López-Ortiz, A., Quimper, C., Tromp, J., van Beek, P.: A fast and simple algorithm for bounds consistency of the alldifferent constraint. In: IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003. pp. 245–250 (2003), `http://ijcai.org/Proceedings/03/Papers/036.pdf`

[102] Mackworth, A.K.: On reading sketch maps. In: Reddy, R. (ed.) Proceedings of the 5th International Joint Conference on Artificial Intelligence. Cambridge, MA, USA, August 22-25, 1977. pp. 598–606. William Kaufmann (1977), `http://ijcai.org/Proceedings/77-2/Papers/006.pdf`

[103] Mairy, J., Deville, Y., Lecoutre, C.: The smart table constraint. In: Integration of AI and OR Techniques in Constraint Programming - 12th International Conference, CPAIOR 2015, Barcelona, Spain, May 18-22, 2015, Proceedings. pp. 271–287 (2015), `https://doi.org/10.1007/978-3-319-18008-3_19`

[104] Mairy, J., Hentenryck, P.V., Deville, Y.: An optimal filtering algorithm for table constraints. In: Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings. pp. 496–511 (2012), https://doi.org/10.1007/978-3-642-33558-7_37

[105] Mairy, J., Hentenryck, P.V., Deville, Y.: Optimal and efficient filtering algorithms for table constraints. Constraints An Int. J. **19**(1), 77–120 (2014), https://doi.org/10.1007/s10601-013-9156-0

[106] Marriott, K., Stuckey, P.J.: Constraints. In: [109] (1998), chapter 1

[107] Marriott, K., Stuckey, P.J.: Finite Constraint Domains. In: [109] (1998), chapter 3

[108] Marriott, K., Stuckey, P.J.: Modelling with Finite Domain Constraints. In: [109] (1998), chapter 8

[109] Marriott, K., Stuckey, P.J.: Programming with Constraints - An Introduction. MIT Press, Cambridge (1998)

[110] Michalewicz, Z.: How to Solve It: Modern Heuristics 2e. Springer-Verlag, Berlin, Heidelberg (2010)

[111] Michalowski, M., Knoblock, C.A., Choueiry, B.Y.: Reformulating constraint models using input data. In: Abstraction, Reformulation, and Approximation, 7th International Symposium, SARA 2007, Whistler, Canada, July 18-21, 2007, Proceedings. pp. 402–404 (2007), https://doi.org/10.1007/978-3-540-73580-9_36

[112] Mohr, R., Masini, G.: Good old discrete relaxation. In: Kodratoff, Y. (ed.) 8th European Conference on Artificial Intelligence, ECAI 1988, Munich, Germany, August 1-5, 1988, Proceedings. pp. 651–656. Pitmann Publishing, London (1988)

[113] Nadel, B.A.: Representation selection for constraint satisfaction: A case study using n-queens. IEEE Expert **5**(3), 16–23 (1990), https://doi.org/10.1109/64.54670

[114] Nguyen, T., Deville, Y.: A distributed arc-consistency algorithm. Science of Computer Programming **30**(1-2), 227–250 (1998), https://doi.org/10.1016/S0167-6423(97)00012-9

[115] Niederlinski, A.: A Quick and Gentle Guide to Constraint Logic Programming via ECL^{i}PS^e. Jacek Skalmierski Computer Studio Gliwice, Poland, 3. edn. (2014)

[116] Nightingale, P.: CSPLib problem 081: Black hole. http://www.csplib.org/Problems/prob081

[117] Nightingale, P.: Savile row (2021), https://savilerow.cs.st-andrews.ac.uk/index.html,lastvisited2021-05-04

[118] Nightingale, P., Akgün, Ö., Gent, I.P., Jefferson, C., Miguel, I., Spracklen, P.:

Automatically improving constraint models in savile row. Artif. Intell. **251**, 35–61 (2017), `https://doi.org/10.1016/j.artint.2017.07.001`

[119] OscaR: Operational research in scala. `https://bitbucket.org/oscarlib/oscar` (2018), zuletzt besucht 2019-08-22

[120] Papadimitriou, C.H., Steiglitz, K.: Combinatorial Optimization: Algorithms and Complexity. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1982)

[121] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research **12**, 2825–2830 (2011)

[122] Peirce, C.S., Hartshorne, C., Weiss, P.: Collected papers, vol. III. In: Cited in: F. Rossi, C. Petrie, and V. Dhar, 1989. Harvard University Press

[123] Perez, G.: Decision diagrams : constraints and algorithms. (Diagrammes de décision : contraintes et algorithmes). Ph.D. thesis, University of Côte d'Azur, France (2017), `https://tel.archives-ouvertes.fr/tel-01677857`

[124] Pesant, G.: A regular language membership constraint for finite sequences of variables. In: Wallace, M. (ed.) Principles and Practice of Constraint Programming - CP 2004. Lecture Notes in Computer Science, vol. 3258, pp. 482–495. Springer (2004)

[125] Petke, J.: Bridging Constraint Satisfaction and Boolean Satisfiability. Artificial Intelligence: Foundations, Theory, and Algorithms, Springer (2015), `https://doi.org/10.1007/978-3-319-21810-6`

[126] Petke, J.: SAT encodings. In: Artificial Intelligence: Foundations, Theory, and Algorithms [125] (2015), `https://doi.org/10.1007/978-3-319-21810-6`, chapter 4

[127] Prestwich, S.D.: CNF encodings. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 75–97. IOS Press (2009), `https://doi.org/10.3233/978-1-58603-929-5-75`

[128] Prosser, P.: Hybrid algorithms for the constraint satisfaction problem. Computational Intelligence **9**, 268–299 (1993), `https://doi.org/10.1111/j.1467-8640.1993.tb00310.x`

[129] Prud'homme, C., Fages, J.G., Lorca, X.: Choco documentation (2017), `http://www.choco-solver.org,lastvisited2019-03-28`

[130] Régin, J.: Generalized arc consistency for global cardinality constraint. In: Clancey, W.J., Weld, D.S. (eds.) Proceedings of the Thirteenth National Conference on

Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Portland, Oregon, USA, August 4-8, 1996, Volume 1. pp. 209–215. AAAI Press / The MIT Press (1996), `http://www.aaai.org/Library/AAAI/1996/aaai96-031.php`

[131] Régin, J., Malapert, A.: Parallel constraint programming. In: Hamadi, Y., Sais, L. (eds.) Handbook of Parallel Constraint Reasoning, pp. 337–379. Springer (2018), `https://doi.org/10.1007/978-3-319-63516-3_9`

[132] Régin, J.C., Malapert, A.: Parallel Constraint Programming, pp. 337–379. Springer International Publishing, Cham (2018), `https://doi.org/10.1007/978-3-319-63516-3_9`

[133] Régin, J., Rezgui, M., Malapert, A.: Embarrassingly parallel search. In: Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings. pp. 596–610 (2013), `https://doi.org/10.1007/978-3-642-40627-0_45`

[134] Rolf, C.C., Kuchcinski, K.: Parallel consistency in constraint programming. In: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2009, Las Vegas, Nevada, USA, July 13-17, 2009, 2 Volumes. pp. 638–644 (2009)

[135] Rolf, C.C., Kuchcinski, K.: Distributed constraint programming with agents. In: Bouchachia, A. (ed.) Adaptive and Intelligent Systems. pp. 320–331. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)

[136] Rossi, F., Beek, P.v., Walsh, T.: Handbook of Constraint Programming. Elsevier, Amsterdam, First edn. (2006)

[137] Rossi, F., Petrie, C.J., Dhar, V.: On the equivalence of constraint satisfaction problems. In: 9th European Conference on Artificial Intelligence, ECAI 1990, Stockholm, Sweden, 1990. pp. 550–556 (1990)

[138] Rousseau, L., Stergiou, K. (eds.): Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 16th International Conference, CPAIOR 2019, Thessaloniki, Greece, June 4-7, 2019, Proceedings, Lecture Notes in Computer Science, vol. 11494. Springer (2019), `https://doi.org/10.1007/978-3-030-19212-9`

[139] Sabin, D., Freuder, E.C.: Contradicting conventional wisdom in constraint satisfaction. In: Cohn, A.G. (ed.) Proceedings of the Eleventh European Conference on Artificial Intelligence, Amsterdam, The Netherlands, August 8-12, 1994. pp. 125–129. John Wiley and Sons, Chichester (1994)

[140] Sabin, M.C., Freuder, E.C.: Detecting and resolving inconsistency and redundancy

in conditional constraint satisfaction problems. AAAI Technical Report (1999), https://scholars.unh.edu/unhmcis_facpub/21

[141] Schiex, T., de Givry, S. (eds.): Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019, Proceedings, Lecture Notes in Computer Science, vol. 11802. Springer (2019), https://doi.org/10.1007/978-3-030-30048-7

[142] Schrijvers, T., Stuckey, P., Wadler, P.: Monadic constraint programming. Journal of Functional Programming **19**(6), 663–697 (2009)

[143] Selman, B., Levesque, H.J., Mitchell, D.G.: A new method for solving hard satisfiability problems. In: Swartout, W.R. (ed.) Proceedings of the 10th National Conference on Artificial Intelligence, San Jose, CA, USA, July 12-16, 1992. pp. 440–446. AAAI Press / The MIT Press (1992), http://www.aaai.org/Library/AAAI/1992/aaai92-068.php

[144] Siarry, P., Michalewicz, Z.: Advances in Metaheuristics for Hard Optimization (Natural Computing Series). 1. edn. (2007)

[145] Smith, B.M., Stergiou, K., Walsh, T.: Using auxiliary variables and implied constraints to model non-binary problems. In: Kautz, H.A., Porter, B.W. (eds.) Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence, July 30 - August 3, 2000, Austin, Texas, USA. pp. 182–187. AAAI Press / The MIT Press (2000), http://www.aaai.org/Library/AAAI/2000/aaai00-028.php

[146] Stergiou, K., Walsh, T.: Encodings of non-binary constraint satisfaction problems. In: Hendler, J., Subramanian, D. (eds.) Proceedings of the Sixteenth National Conference on Artificial Intelligence and Eleventh Conference on Innovative Applications of Artificial Intelligence, July 18-22, 1999, Orlando, Florida, USA. pp. 163–168. AAAI Press / The MIT Press (1999), http://www.aaai.org/Library/AAAI/1999/aaai99-024.php

[147] Stuckey, P.J., Becket, R., Brand, S., Brown, M., Feydy, T., Fischer, J., de la Banda, M.G., Marriott, K., Wallace, M.: The evolving world of minizinc. In: Frisch, A.M., Lee, J. (eds.) International Workshop on Constraint Modelling and Reformulation (ModRef). pp. 156–170 (2009)

[148] Stuckey, P.J., Feydy, T., Schutt, A., Tack, G., Fischer, J.: The minizinc challenge 2008-2013. AI Magazine **35**(2), 55–60 (2014)

[149] Stuckey, P.J., Marriott, K., Tack, G.: MiniZinc Handbook, Release 2.3.2 (September 2019), https://www.minizinc.org/resources.html, zuletzt besucht 2019-09-16

[150] Sven Löffler, Ilja Becker, P.H.: CSPLib problem 087: Rotating rostering problem. http://www.csplib.org/Problems/prob087

[151] Trick, M.A.: A dynamic programming approach for consistency and propagation for knapsack constraints. Annals OR **118**(1-4), 73–84 (2003), `https://doi.org/10.1023/A:1021801522545`

[152] Tsang, E.P.K.: Foundations of constraint satisfaction. Computation in cognitive science, Academic Press (1993)

[153] Van Hentenryck, P., Michel, L.: Constraint-Based Local Search. MIT Press (2005)

[154] Van Hoeve, W.J.: The alldifferent constraint: A survey. In: Sixth Annual Workshop of the ERCIM Working Group on Constraints (2001), prague

[155] Velev, M.N.: Exploiting hierarchy and structure to efficiently solve graph coloring as SAT. In: Gielen, G.G.E. (ed.) 2007 International Conference on Computer-Aided Design, ICCAD 2007, San Jose, CA, USA, November 5-8, 2007. pp. 135–142. IEEE Computer Society (2007), `https://doi.org/10.1109/ICCAD.2007.4397256`

[156] Vion, J., Piechowiak, S.: From MDD to BDD and arc consistency. Constraints An Int. J. **23**(4), 451–480 (2018), `https://doi.org/10.1007/s10601-018-9286-5`

[157] Walsh, T.: SAT v CSP. In: Dechter, R. (ed.) Principles and Practice of Constraint Programming - CP 2000, 6th International Conference, Singapore, September 18-21, 2000, Proceedings. Lecture Notes in Computer Science, vol. 1894, pp. 441–456. Springer (2000), `https://doi.org/10.1007/3-540-45349-0_32`

[158] Walsh, T.: General symmetry breaking constraints. In: Benhamou, F. (ed.) Principles and Practice of Constraint Programming - CP 2006, 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4204, pp. 650–664. Springer (2006), `https://doi.org/10.1007/11889205_46`

[159] Weigel, R., Bliek, C.: On reformulation of constraint satisfaction problems. In: Prade, H. (ed.) 13th European Conference on Artificial Intelligence, Brighton, UK, August 23-28 1998, Proceedings. pp. 254–258. John Wiley and Sons (1998)

[160] Weise, T.: Global Optimization Algorithms - Theory and Application. Self-Published, 2. edn. (2009), `http://www.it-weise.de/`, online verfügbar unter http://www.it-weise.de/

[161] Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: Swi-prolog. Theory and Practice of Logic Programming – TPLP **12**(1-2), 67–96 (2012)

[162] Yokoo, M., Hirayama, K.: Algorithms for distributed constraint satisfaction: A review. Autonomous Agents and Multi-Agent Systems **3**(2), 185–207 (2000), `https://doi.org/10.1023/A:1010078712316`

[163] Zemke, G.: Einführung in die lineare Optimierung, pp. 26–48. Vieweg+Teubner Verlag, Wiesbaden (1971), `https://doi.org/10.1007/978-3-322-88788-7_3`

[164] Zhou, J.: Sort-merge join. In: Liu, L., Özsu, M.T. (eds.) Encyclopedia of Database Systems, Second Edition. Springer (2018), https://doi.org/10.1007/978-1-4614-8265-9_867

[165] Zhou, N.F., Kjellerstrand, H., Fruhman, J.: Constraint Solving and Planning with Picat. Springer-Verlag (2015)

Abbildungsverzeichnis

Tabellenverzeichnis