

Universität Leipzig
Fakultät für Mathematik und Informatik
Institut für Informatik

USING CONSTRAINTS TO
RENDER WEBSITES
APPLICATIONS OF ARTIFICIAL INTELLIGENCE IN
E-COMMERCE ENVIRONMENTS

Diplomarbeit

Leipzig, August 2007

VORGELEGT VON:
Lars Kotthoff
geboren am 09.04.1983
Studiengang Informatik

BETREUER:
Prof. Dr. Gerhard Brewka

ABSTRACT

Constraint programming is an area of Artificial Intelligence which has many applications. This thesis applies its techniques to a new kind of problem – the rendering of online retailer websites.

First, in-depth introductions to constraint programming and the problem of rendering a shop website will be given. A prototypical implementation of a constraint problem solver and a system to solve and illustrate the problem will be described.

The architecture of the prototypical implementation and specific features, algorithms, and design decisions will be detailed, analysed, and illustrated. An overview of related work both in the fields of constraint programming and website generation will be presented and existing technologies evaluated.

Features and concepts unique to this thesis, like real-time constraint satisfaction, will be introduced and discussed.

Finally, a comprehensive example will illustrate the problem, means of modelling it, and possible solutions. An outlook to future work and a summary conclude the thesis.

ZUSAMMENFASSUNG

Constraint Programming ist ein Teilgebiet der künstlichen Intelligenz mit vielen praktischen Anwendungen. Diese Diplomarbeit wendet die Techniken auf eine neue Art von Problem an – das Rendern von Webseiten von Internetshops.

Zuerst wird eine detaillierte Einführung zu Constraint Programming und dem Problem die Webseite eines Online-Shops zu rendern gegeben werden. Eine Beispielimplementierung eines Constraint Problem Solvers und eines Systems um das Problem zu lösen und illustrieren werden beschrieben werden.

Die Architektur der Beispielimplementierung und spezielle Eigenschaften, Algorithmen und Implementierungsentscheidungen werden genau beschrieben, analysiert und illustriert werden. Ein Überblick von ähnlichen Arbeiten sowohl im Bereich des Constraint Programming als auch im Bereich des Generierens von Webseiten wird dargestellt und vorhandene Technologien bewertet werden.

Besonderheiten und Konzepte, die in dieser Diplomarbeit erarbeitet wurden, wie Echtzeit-Constraint Satisfaction, werden eingeführt und diskutiert werden.

Schließlich wird ein ausführliches Beispiel das Problem, Arten der Modellierung und mögliche Lösungen veranschaulichen. Ein Ausblick auf zukünftige Forschung und eine Zusammenfassung beschließen diese Diplomarbeit.

ACKNOWLEDGEMENTS

I would like to thank Prof. Dr. Gerhard Brewka for supervising this thesis and providing me with valuable feedback. Many thanks also go to Matthew Round and Karl McCabe of Amazon.com for reviewing the description of the problem of rendering a website.

CONTENTS

0	INTRODUCTION	1
0.1	Motivation	2
0.2	Aim and Scope	2
0.3	Related Work	2
PART I MODELLING OF THE PROBLEM		3
1	DESCRIPTION OF THE PROBLEM	5
1.1	Personalised Content	5
1.2	Generating Personalised Content	6
1.3	Constraints to consider	7
2	CONSTRAINTS	9
2.1	Introduction	9
2.2	Constraint Satisfaction Problems	9
2.3	Solution Process	11
2.3.1	Arc Consistency	14
2.4	Constrained Optimisation Problems	15
2.4.1	Extended Constrained Optimisation Problems	17
2.5	Soft Constraints	18
2.6	Real-time Constraint Satisfaction	20
2.6.1	Analysis of the Function ψ	22
3	CONSTRAINT PROBLEM MODEL	25
3.1	Slots and Campaigns	25
3.2	Values of Campaigns	25
3.3	Values of Slots	26
3.3.1	Example	26
3.4	Relaxation of Constraints	27
3.5	Duplicate Content	28
3.5.1	Example	29
3.6	Forced Promotions	31
3.6.1	Example	32
3.7	Real-time Problem Solution	32
PART II PROTOTYPICAL IMPLEMENTATION		35
4	OVERVIEW	37
4.1	System Architecture	37
4.1.1	Distributed Approach	37
4.1.2	Web Interface	38
4.1.3	Web Service	38
4.2	Implementation Language	38
4.2.1	Documentation	39
4.2.2	Testing	39
4.2.3	Packaging	40
4.3	Version Control System	40
4.4	Test Machine Setup	40

5	CONSTRAINT PROBLEM SOLVER LIBRARY	43	
5.1	Overview of existing Constraint Problem Solvers		43
5.2	Architecture	44	
5.2.1	Domain	45	
5.2.2	Variable	46	
5.2.3	AbstractConstraint		47
5.2.4	BinaryConstraint		47
5.2.5	BinaryRelation	48	
5.2.6	AllDifferentConstraint		48
5.2.7	TupleConstraint	49	
5.2.8	OneOfEqualsConstraint		49
5.2.9	ConstraintList	49	
5.2.10	Problem	50	
5.2.11	Solution	50	
5.2.12	ConstraintSolver	51	
5.2.13	Ruby Extensions	52	
5.3	Constraint Problem Solution	52	
5.3.1	Solution Process	52	
5.3.2	Modifications for Soft Constraints		55
5.3.3	Constraint Satisfaction with Time Limit		56
5.3.4	Variable and Value Ordering	58	
5.4	Constraint Revision	58	
5.4.1	Binary Constraints	58	
5.4.2	All Different Constraints		59
5.4.3	Tuple Constraints	60	
5.4.4	One-of-equals Constraint		61
5.5	Tests and Package Management	62	
5.6	Limitations	62	
6	CONSTRAINT PROBLEM SOLVER SOAP WRAPPER		65
6.1	Architecture	65	
6.1.1	Library Script	65	
6.1.2	WSDL	67	
6.1.3	Control Scripts	67	
6.2	Tests and Package Management		68
6.3	Limitations	68	
6.4	Use of the Interface	68	
7	WEB USER INTERFACE	69	
7.1	Architecture	69	
7.1.1	Data Model	69	
7.1.2	Controller	70	
7.1.3	View	72	
7.2	Interface to Amazon.com	73	
7.3	Tests and Package Management		74
7.4	Full Example	75	
8	SUMMARY	79	
8.1	Future Work	79	
8.1.1	Constraint Model	79	
8.1.2	Constraint Solver	80	
8.2	Conclusion	80	

PART III APPENDIX	83	
A PERFORMANCE EVALUATION OF CONSISTENCY ALGORITHMS		85
A.1 Binary Constraints	85	
A.1.1 Methodology	86	
A.1.2 Results	86	
A.2 All Different Constraints	89	
A.2.1 Methodology	89	
A.2.2 Results	89	
A.3 The Difference All Different makes	92	
B EFFECTIVENESS OF REAL-TIME CONSTRAINT SATISFACTION		93
B.1 Methodology	93	
B.2 Results	94	
B.2.1 Time Limit after first Solution	94	
B.2.2 Time Limit before first Solution	97	
C INSTALLATION INSTRUCTIONS AND SOFTWARE VERSIONS		103
C.1 General	103	
C.2 Constraint Problem Solver Library	103	
C.3 Constraint Problem Solver SOAP Wrapper	103	
C.4 Web User Interface	103	
GLOSSARY	105	
BIBLIOGRAPHY	107	

LIST OF FIGURES

Figure 1	Non-personalised Website	5	
Figure 2	Personalised Website	6	
Figure 3	Possible Solution for the N-Queens Problem for $n = 4$	10	
Figure 4	Constraint Network Graph for the 4-Queens Problem	11	
Figure 5	Example Search Tree built during the Solving of the 4-Queens Problem	12	
Figure 6	Solution Process for the 4-Queens Problem with Constraint Propagation	14	
Figure 7	Search Tree built during the Solving of the 4-Queens Problem with Pruning	15	
Figure 8	Example Orders for the Steel Mill Slab Problem	17	
Figure 9	Solution to the Steel Mill Slab Problem in Figure 8	17	
Figure 10	Example Soft Constraint Network Graph for an inconsistent Problem	19	
Figure 11	Example Curve of ψ with $t_l = 1$, $T = 1$, and $\alpha = 100 \cdot t_l = 100$		23
Figure 12	Constraint Network Graph for Example 3.3.1	27	
Figure 13	System Architecture of prototypical Implementation	37	
Figure 14	Structure of the Constraint Solver Library	46	
Figure 15	Actions performed during the Solving of a Constraint Problem		54
Figure 16	High-Level Actions performed while solving a Soft Constraint Problem	56	
Figure 17	Solving of a Constraint Problem with Time Limit	57	
Figure 18	Architecture of SOAP Server	66	
Figure 19	Entity-Relationship Diagram [Che76] of the Data Model of the Web User Interface	70	
Figure 20	Form to specify Problem to solve	72	
Figure 21	Result Page with rendered Solution for Input in Figure 20	73	
Figure 22	Activity Diagram for the Web User Interface	74	
Figure 23	Binary Constraint Performance for Solution of "pathological" Problems	87	
Figure 24	Binary Constraint Performance for Solution of Identity Problems	88	
Figure 25	Binary Constraint Performance for Solution of Ordering Problems	88	
Figure 26	All Different Performance for Solution of dense Problems	90	
Figure 27	All Different Performance for Solution of random Problems	91	
Figure 28	All Different Performance for Solution of "pathological" Problems	91	
Figure 29	All Different and Binary Constraint Consistency Performance for Solution of "pathological" Problems	92	
Figure 30	Deviation from the Time Limit after a Solution has been found for all different Problems with hard Constraints	94	
Figure 31	Deviation from the Time Limit after a Solution has been found for all different Problems with soft Constraints	95	
Figure 32	Deviation from the Time Limit after a Solution has been found for Identity Problems with hard Constraints	95	

Figure 33	Deviation from the Time Limit after a Solution has been found for Identity Problems with soft Constraints	96
Figure 34	Deviation from the Time Limits before a Solution has been found for all different Problems with hard Constraints	97
Figure 35	Deviation from the Time Limits before a Solution has been found for all different Problems with soft Constraints	98
Figure 36	Deviation from the Time Limits before a Solution has been found for Identity Problems with hard Constraints	100
Figure 37	Deviation from the Time Limits before a Solution has been found for Identity Problems with soft Constraints	101

LIST OF TABLES

Table 1	Overview of Arc Consistency Algorithms for Binary Constraints	16
Table 2	Overview of Consistency Algorithms for the All Different Constraint	16
Table 3	XML based Web Service Protocols considered for the Web Service	38
Table 4	Overview of Constraint Problem Solvers	44

LIST OF DEFINITIONS

Definition 1	Constraint Satisfaction Problem	9
Definition 2	Constraint	10
Definition 3	Partial Assignment	10
Definition 4	Complete Assignment	10
Definition 5	Solution to a Constraint Satisfaction Problem	11
Definition 6	Problem Class	11
Definition 7	Problem Instance	11
Definition 8	Constraint Network	11
Definition 9	Search Tree	12
Definition 10	Constraint Propagation	12
Definition 11	Consistency Properties	13
Definition 12	Local Consistency	13
Definition 13	Support	13
Definition 14	Pruning	13
Definition 15	Constraint Revision	13
Definition 16	Global Consistency	13
Definition 17	Constrained Optimisation Problem	15
Definition 18	Solution to a Constrained Optimisation Problem	16
Definition 19	Optimal Solution to a Constrained Optimisation Problem	16

Definition 20	Extended Constrained Optimisation Problem	18	
Definition 21	Solution to an Extended Constrained Optimisation Problem		18
Definition 22	Soft Constraint	18	
Definition 23	Valuation Structure	18	
Definition 24	Soft Constraint Network	19	
Definition 25	Soft Constraint Problem	19	
Definition 26	Solution to a Soft Constraint Problem	19	
Definition 27	Constraint Satisfaction Time	21	
Definition 28	Real-time Constraint Problem	21	
Definition 29	Solution to a Real-time Constraint Problem		22
Definition 30	Campaign	25	
Definition 31	Slot	25	
Definition 32	Page	25	
Definition 33	Problem of Rendering a Page	25	
Definition 34	Value of a Page	25	
Definition 35	Problem of Rendering a Page with Page Value		26
Definition 36	Value of a Page	26	
Definition 37	Relaxed Constraint	27	
Definition 38	Problem of Rendering a Page with relaxed Constraints		28
Definition 39	Value of a relaxed Page	28	
Definition 40	Places in Slots	28	
Definition 41	Allowed Tuples for a Slot - Place Pair Constraint		29
Definition 42	Problem of Rendering a Page with Time Limit		32



INTRODUCTION

In today's fast-moving society, electronic commerce environments become increasingly popular. The websites of large online retailers serve a vast number of customers who prefer shopping online from the comfort of their home to traditional shopping. Every second, thousands of transactions are handled, putting an enormous load on the backend systems.

At the same time, the requirements increase even further. Stores with personalised pages for every individual customer and targeted recommendations prove valuable for both retailers and customers. Supplying these personalisations does not only demand more from the backend systems, but also from the designers and programmers of websites. They have to take more and more factors into account and handle increasing complexity of the systems and their interaction.

The problem of generating personalised stores is a problem of combining website components such that a number of constraints are satisfied and the value for the retailer and customer is maximised. There are a lot of constraints to take into account and variables to consider. Has the customer visited the store previously, maybe bought something? Is the content reasonably varied? Has the content been generated within reasonable time?

There are established methods for solving problems which involve constraints in the field of Artificial Intelligence. These algorithms can be applied to any problem which is modelled appropriately, and separate the problem from the process of solving it. Additional variables, such as the maximisation of a value of "usefulness", can be taken into account. Most of the algorithms have been optimised such that they are able to solve even large problems in acceptable time.

The problem remains to model the generation of a website with constraints such that it can be solved effectively and efficiently. Most processes cannot directly be expressed as constraints. Global state cannot easily be expressed. The satisfaction of some constraints is optional, of others elementary.

This work will investigate the problems and variables to consider when personalising websites. It will use constraints to model the problem and present, investigate, and evaluate solution procedures. The problem, the modelling, the solution procedures, and their interaction will be researched and explored.

Amazon.com will be used as an example of an online retailer who aims to personalise its content.

This document is organised into two parts. In the first part, the problem is examined, explained and modelled. The first chapter looks at the problem of rendering a website, explains personalised content and the constraints involved. The second chapter gives an overview and definitions of constraint satisfaction and optimisation problems, representations and solution procedures. The third chapter models the problem described in chapter 1 by means of constraints introduced in chapter 2.

The second part is concerned with a prototypical implementation of the model. It will explain the approach taken to implement it, details of the implementation, and how to solve the problem of rendering a website with it.

0.1 MOTIVATION

The main point of motivation for the work carried out in this thesis is to create a means of tackling the increasing complexity of web site design and explore new applications of Artificial Intelligence. The specification of constraints to generate a web site does not require programming or computer science skills, the problem and means of solving it are cleanly separated.

A useful approach in software design is to separate design and implementation of the system [Zhuo6]. The separation of problem and means of solving it takes this approach a step further [Fre97]. It enables the expression of hard problems in easy terms and is a flexible and intuitive way of problem solving.

Currently the constraints involved in web site generation are usually handled in specialised code. This code becomes more complex and difficult to maintain as constraints are added and change. Making even minor changes involves redeploying whole subsystems and increases the risk of outages.

In a constraint-based framework, the constraints can change permanently without any disruption.

0.2 AIM AND SCOPE

The aim of this work is to provide a proof of concept and prototypical implementation of a system to generate web sites on the fly using constraint programming. In particular, the constraints for a site are assembled into a problem to be solved at run time. The constraints may be different for each problem; the problem and means of solving it are completely separate.

Providing a system which is ready for deployment in industry is beyond the scope of this thesis.

0.3 RELATED WORK

Although the design and implementation of web sites and online portals as well as Artificial Intelligence methods are both areas of very active research, there exists only very little work relating the two fields.

Among the related work is a study to design websites using analytical approaches [YHW07], a dynamic programming approach to bandwidth constraints [JLC06], and several studies investigating the integration of soft constraints with semantic web approaches [PCM⁺06a] [PCM⁺06b]. Artificial Intelligence methods have also been applied in the areas of web security [Hua06] and generic business processes [LSPG06] [Tsa02].

The specific application of Artificial Intelligence techniques which is subject to this thesis has never been investigated before.

Part I

MODELLING OF THE PROBLEM

DESCRIPTION OF THE PROBLEM

1.1 PERSONALISED CONTENT

Nowadays, online retailers aim to provide personalised content to their customers to maximise their profits and improve the shopping experience. Personalisation uses data which is known about the customer to make recommendations. Previous purchases, items the customer has looked at before, and searches can be compiled into a customer profile.

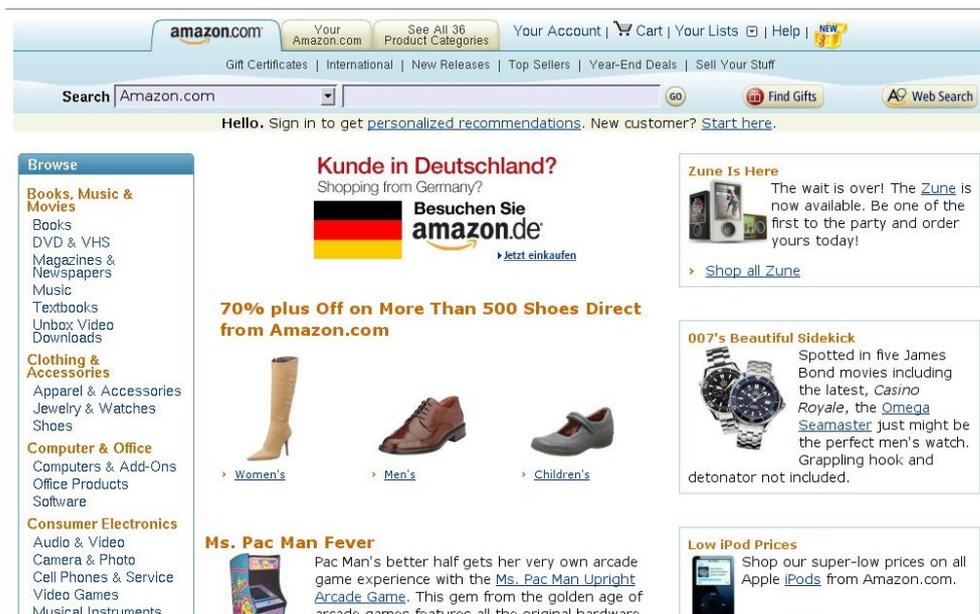


Figure 1. Non-personalised Website

Figure 1 shows an example of generic content on the Amazon.com gateway page. The promotions are unrelated and offer shoes, popular electronics and watches. The content is very easy to generate and can be created offline by a designer. Displaying it is simply a matter of fetching the stored page fragment.

Figure 2 shows content which takes into account previous activities. It gives an overview of the user's history by displaying recently viewed products. There are also helpful pointers to products the customer did not take a look at but might be interested in. The interest profile is derived from a search query.

Personalised pages are significantly more useful to customers than unpersonalised



Figure 2. Personalised Website

ones. They provide a starting point to revisit items one is considering buying, assist in exploring the store and finding the product one is looking for, discovering products one did not know about, and omit promotions one is not interested in.

1.2 GENERATING PERSONALISED CONTENT

Offering personalised content introduces new problems. Designers cannot create websites offline anymore, because the static content they can provide is not personalised. The website needs to be created as it is requested; dynamically. Computers have to generate content according to the information available. Programmers adopt the responsibilities of designers and develop systems which present personalised pages. This complex task requires knowledge of what information is available, how to use it, and how to show it.

There are more factors to take into account. Dynamic components do not only need to know what the customer is doing, but also what the other dynamic components which render the same page are doing. The result needs to be free of duplication and other bad content. The generator must filter any items which have already been bought, are unavailable, or are too similar to items the customer already owns, even if they fit the customer's profile.

Another issue is how items to promote are chosen. An item which has been released recently can be promoted as new and interesting, but if the customer has already taken a look at it, it is no longer new to him. Not all data is equally valuable when generating personalised content. Considering recent activities may result in better promotions than considering an item which was viewed a year ago.

Further difficulties arise when products customers bought as presents for relatives or friends are considered. These purchases should not be used to personalise pages, as they do not reflect the interests of the customers. On the other hand, the content

generated from this data will be of interest if they are looking for a present for the same person again.

Depending on the amount of data available which needs to be considered for each individual customer and the number of dynamic components which make up the page, the generation of personalised content becomes almost arbitrarily complex. As the number of involved constraints increases, errors are more likely to occur in the model because the number of possible interactions between the constraints increases exponentially.

1.3 CONSTRAINTS TO CONSIDER

The most important constraints are summarised below. Depending on the specific retailer and web site there might be many more to consider. Not all constraints can be modelled appropriately within the scope of this thesis. It is meant to be a proof of concept and general guideline rather than an exhaustive and accurate model.

NO DUPLICATE CONTENT Any content shown on a page must be free of duplicates. Not only does this create a bad impression with the customer, but also uses space which might instead be used to promote different products and increase the chances of showing something the user is interested in.

NO BAD RECOMMENDATIONS Bad recommendations are promotions for products the customer has already bought or is not interested in. The chances of selling the promoted item are very small, and maybe the customer will be annoyed and lose interest in the page.

MAXIMISATION OF PAGE VALUE The value of a page is determined by the content shown and how it is shown. Personalised content is more valuable than non-personalised content. Showing something valuable on top of the page is better than showing it at the bottom, such that the customer has to scroll down to notice it. Further difficulties arise because the value of content is not known or not known exactly. Different customers prefer different types of content and therefore the content does not have an intrinsic value.

LIMITED TIME The page must be presented to the customer before he loses interest – the time which can be spent generating it is limited. This does not only include the time to compute what to show where, but also the time the page fragments need to render themselves.

AVAILABLE DATA Depending on the data available, the content which can be generated is different. The value of the page has to be maximised regardless of this. There always has to be something to display, even if there is no data available at all.

EXTERNAL EVENTS External events, such as the completion of service calls to back-end systems, must be considered when generating the page. A service call may not successfully retrieve content and void the current configuration.

FORCED PROMOTIONS Sometimes it is desirable not to maximise the value of the page, but to display fixed content. This might for example be a paid advert or the promotion of a new product nobody knows about yet.

LEGAL ISSUES In some countries, it might be illegal to promote certain items on certain pages.

8 DESCRIPTION OF THE PROBLEM

RETAILER POLICY The online retailer might have a policy which types of products to promote on which types of pages, at which time, and to which customers.

CONSTRAINTS

This chapter introduces constraints and constraint problems. It gives definitions needed to model problems and illustrates the concepts.

2.1 INTRODUCTION

Constraints and constraint satisfaction problems occur in many everyday situations. Scheduling rooms to courses, buses to routes, or workforce to projects are typical examples [GNT04]. Constraint problems are not limited to scheduling however, further applications include information retrieval, resource allocation, and even games such as Sudoku. Constraint programming has a wide range of applications [Pug95] [Wal96].

Investigating constraints and their properties has long been a part of Artificial Intelligence research. For more than 30 years, scientists have described and improved methods to solve constraint problems, model them more effectively, and apply them to real-world problems, e.g. [GLSS79].

The notion of a constraint is simple and intuitive – something is required to adhere to external conventions and can therefore not be in arbitrary states. Complex states are characterised by sets of constraints.

The following sections introduce the concepts of constraint problems by presenting examples, analysing them, and formalising the informal notion of constraints and constraint problems through definitions. Most of the definitions follow the standard conventions [Migo6] [Deco3].

2.2 CONSTRAINT SATISFACTION PROBLEMS

A popular problem to introduce the concepts of constraint programming is the n -queens problem. The aim is to place n queens on an $n \times n$ chessboard such that no queen is attacking another queen [RN02]. The problem is illustrated in figure 3.

Each queen must be in a different row and column of the board and must not be diagonally in a line with any other queen. The queens are the *variables* q_1, q_2, \dots, q_n of the problem. The positions on the board each queen may take comprise the set of values each variable may have, its *domain*. The domain of each queen is the set $\{1, \dots, n\}$ to designate the position in its row. The problem is modelled with each queen being in a different row.

The analysis of the n -queens problem leads to the following definition.

Definition 1 (Constraint Satisfaction Problem). A *constraint satisfaction problem* P is a tuple (X, D, C) . $X = \langle x_1, \dots, x_n \rangle$ is a tuple of n variables and $D = \langle d_1, \dots, d_n \rangle$ is a tuple of n domains. Each domain $d_i \in D$ belongs to the variable $x_i \in X$. $C = \{c_1, \dots, c_m\}$ is a set of constraints over the variables from X .

	Q		
			Q
Q			
		Q	

Figure 3. Possible Solution for the N-Queens Problem for $n = 4$

This definition is different from the ones usually found. Instead of sets of variables and domains, the mapping is made explicit by tuples. As sets are not ordered, it would be unclear which domain belonged to which variable. Furthermore, no two variables could have a common domain, but this is often the case in constraint satisfaction problems.

The notions of variables and their domains have already been illustrated in the previous paragraph. The requirements for the values of the variables can be formalised as *constraints*.

Definition 2 (Constraint). A *constraint* $c(x_1, \dots, x_j)$ constrains the assignment of values to the variables $x_1, \dots, x_j \in X$. The *arity* of a constraint is the number of variables it constrains. The constraint specifies a subset of the Cartesian product $d_1 \times \dots \times d_j$ of the domains of the variables x_1, \dots, x_j that constitutes an allowed assignment.

Constraints can be represented extensionally and intensionally. The extensional representation specifies all tuples which are allowed assignments explicitly, the intensional representation specifies them implicitly. In some cases, the disallowed assignments are specified instead of allowed assignments.

The n -queens problem requires all queens to be in a different column. This is achieved by introducing a constraint of arity n over all variables which requires them to have different values. The extensional representation of this constraint is the set of all the tuples of allowed values, i.e. $\{\langle 1, 2, \dots, n \rangle, \langle 1, 3, \dots, n \rangle, \dots\}$. The intensional representation $\text{AllDifferent}(x_1, \dots, x_n)$ is much shorter and easier to understand.

The requirement that any queen must not be on a diagonal line with any other queen is harder to represent. For each variable, $2(n - 1)$ binary constraints are introduced which require the value of the first variable to be different from the value of the other variable plus or minus the difference in rows between the queens; $\{q_1 \neq q_2 - 1, q_1 \neq q_2 + 1, q_1 \neq q_3 - 2, q_1 \neq q_3 + 2, \dots\}$.

To solve the n -queens problem, the queens are placed on the board one at a time. When a queen is positioned on the board, a value is assigned to the variable which describes the queen.

Definition 3 (Partial Assignment). A *partial assignment* assigns a value to one or more $x_i \in X$ from their respective domains $d_i \in D$.

Positioning the first queen on the leftmost field of the board is an example of a partial assignment for the n -queens problem.

Definition 4 (Complete Assignment). A *complete assignment* assigns a value to every $x_i \in X$ from their respective domains $d_i \in D$.

In a complete assignment for the n-queens problem, all queens are positioned somewhere on the board. The positions do not necessarily fulfil the constraints. If they do, a *solution* has been found.

Definition 5 (Solution to a Constraint Satisfaction Problem). A *solution* to a constraint satisfaction problem P is a complete assignment that satisfies all constraints $c \in C$.

Constraint problems can be organised into *problem classes* and *problem instances*.

Definition 6 (Problem Class). A *problem class* is a problem specified with one or more parameters.

The n-queens problem constitutes a problem class. It has one parameter, the number of queens.

Definition 7 (Problem Instance). A *problem instance* is an instance of a problem class where all parameters of the problem class have concrete values assigned.

The 4-queens problem illustrated in figure 3 is a problem instance of the n-queens problem class. The parameter $n = 4$ and determines the number of queens and the size of the board.

Constraint problems can be represented graphically as *constraint networks*.

Definition 8 (Constraint Network). The nodes in a *constraint network* represent variables, edges represent constraints over the variables they are connecting. Edges can be directed or undirected. They are also referred to as *arcs*.

An example network for the 4-queens problem is depicted in figure 4. The 4-ary constraint which requires the assignments to all variables to be different has been decomposed into binary constraints because four-dimensional hyper arcs are hard to draw on two-dimensional paper. Likewise, the constraints that no queen must be on a diagonal line with any other queen has been omitted.

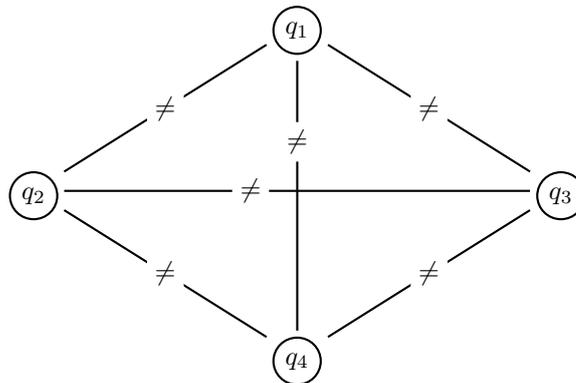


Figure 4. Constraint Network Graph for the 4-Queens Problem

2.3 SOLUTION PROCESS

There are several approaches to solving constraint problems. The most established and wide-spread algorithm is branch-and-bound search [LW66]. The variables are

The notion of *consistency* is introduced to formalise the process of constraint propagation [Mac75].

Definition 11 (Consistency Properties). A *consistency property* holds when constraint propagation of a certain kind reaches a fixed point, i.e. no new information can be deduced from the subset of constraints.

The assignment of 1 to q_1 allows not only the deduction that $q_2 \neq 1$, but also that $q_3 \neq 1$ and $q_4 \neq 1$. Even more information can be deduced from the other constraints that no two queens must be on a diagonal line. Only after all these restrictions have been deduced, the problem is consistent again.

Definition 12 (Local Consistency). A unary constraint $c(x_i)$ is *locally consistent* if and only if for every value of the variable x_i from its domain $d_j \in D_i$, $c(x_i)$ is satisfied if $x_i = d_j$.

A constraint of arity $n + 1$, $c(x_a, \dots, x_b)$, is locally consistent if and only if all n -ary constraints over the variables x_a, \dots, x_b are locally consistent and for every domain value of x_i , $d_j \in D_i$, there is at least one tuple of assignments $\langle d_c, \dots, d_d \rangle$ to the variables $\langle x_a, \dots, x_b \rangle \setminus x_i$ such that $c(x_a, \dots, x_b)$ is satisfied if $x_i = d_j$.

The constraint that all queens must be in different columns is locally consistent if after the assignment of a position to a queen this horizontal position has been excluded from the domains of all other queens.

Definition 13 (Support). A value d_j in a domain D_i is *supported* if and only if all constraints over the variable x_i are locally consistent. The set of assignments which support d_j for a constraint c contains all tuples of values which can be assigned to the other variables constrained by c such that c holds if $x_i = d_j$.

Support is bi-directional, i.e. if value d_i of a variable supports the value d_j of another variable for a constraint c , then d_j also supports d_i for c .

The value 3 in the domain of the second queen is supported after the first queen has been placed in the leftmost upper corner of the board, because only 1 and 2 are forbidden by constraints.

The following definitions formalise the notions of excluding values from a domain after an assignment and consistency of a problem.

Definition 14 (Pruning). The *pruning* of a set of values S from a domain D denoted by $D_p = D \setminus S$ is the removal of all $s \in S$ from D such that $D_p \cap S = \emptyset$. If $D_p = \emptyset$ after the pruning, a *domain wipe out* has occurred.

Definition 15 (Constraint Revision). *Constraint revision* is the process of enforcing local consistency for a constraint.

Definition 16 (Global Consistency). A problem is *globally consistent* if and only if all constraints are locally consistent.

The definitions are illustrated in figures 6 and 7 which show the allowed positions on the board and the search tree for the solution of the 4-queens problem, respectively. Dashed fields on the board designate forbidden positions.

Search algorithms which enforce consistency after an assignment to reduce the size of the search tree are usually referred to as *forward checking* algorithms [HE80]. They are extended backtracking algorithms which only backtrack when the domain of a variable is wiped out after enforcing consistency.

Both search algorithms are *complete*, i.e. if a solution to the problem exists, it will be found.

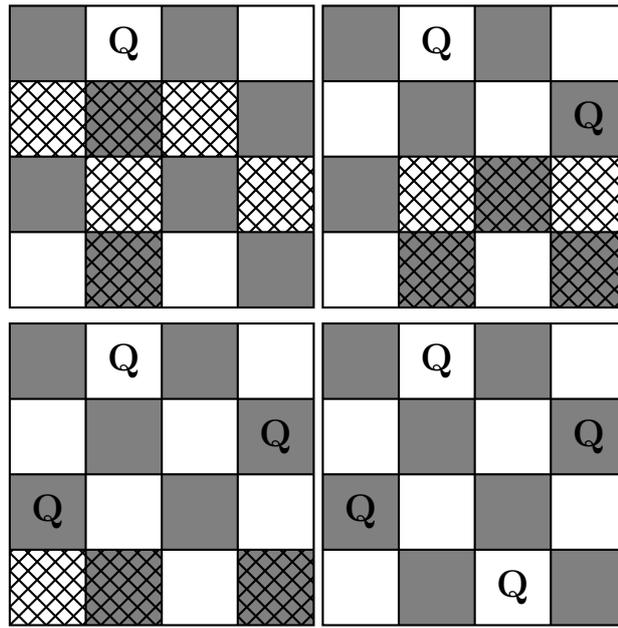


Figure 6. Solution Process for the 4-Queens Problem with Constraint Propagation

2.3.1 ARC CONSISTENCY

Local consistency or arc consistency for binary constraints is the best studied area of constraint revision and propagation. There are many different algorithms to achieve arc consistency on networks of binary constraints.

The algorithms can be separated into two basic classes – *coarse grained* and *fine grained* algorithms. Fine grained algorithms keep track of the support for every domain element of every variable and enforce consistency when individual domain values are removed or added, while coarse grained algorithms do not keep track of individual values and enforce consistency on arcs if the domains of the involved constraints change.

Coarse grained algorithms are generally preferred because they are easier to implement and often exhibit a better run-time behaviour than fine grained algorithms with a lower complexity due to fewer data structures and hence less overhead.

Table 1 presents an overview of algorithms to enforce arc consistency. It is not meant to be complete or exhaustive, but to show the most important algorithms and their characteristics.

Arc consistency algorithms are often integrated with forward checking algorithms and referred to as *maintaining arc consistency (MAC)* algorithms [SF94].

There are also arc consistency algorithms for constraints of higher arity. The all different constraint is the most popular and best studied non-binary constraint and several algorithms have been developed to enforce different levels of consistency on it. Table 2 shows an overview of some algorithms [vHo1]. Several studies have extended the classic notion of the all different constraint to more sophisticated filtering algorithms [KH06] or different kinds of domains [QW05].

Recent research has focused on generalised or hyper arc consistency for higher arity constraints [Rég96] [Rég02] [KT05] [KT03] [QGLOB05].

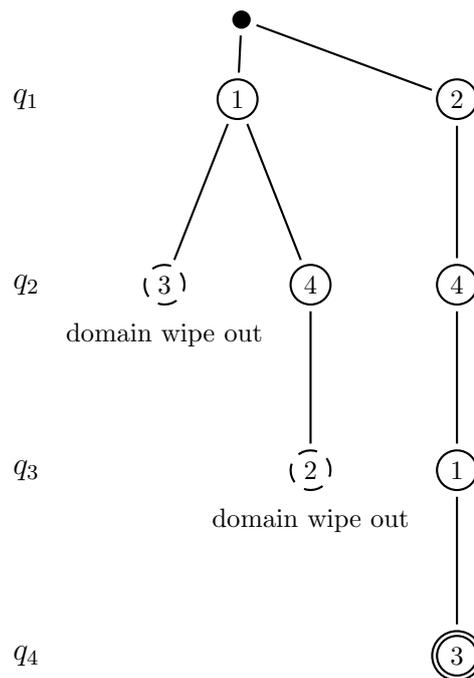


Figure 7. Search Tree built during the Solving of the 4-Queens Problem with Pruning

2.4 CONSTRAINED OPTIMISATION PROBLEMS

Constraint satisfaction problems are a kind of constraint problems where a solution which satisfies all of the constraints needs to be found. Another kind of constraint problems, *constrained optimisation problems*, requires not only a solution, but assigns a value of “goodness” to every solution, and seeks to find the best one.

The steel mill slab problem is an example for a constrained optimisation problem [FMW01]. A simplified version will be presented here to illustrate the concepts of constrained optimisation problems.

The steel mill slab problem class consists of n orders, each of a particular size. The steel mill is able to produce m different sizes of slabs. The objective is to assign the n orders to slabs such that the total waste is minimised. The problem can be modelled with n variables designating the maximum number of slabs to be produced. The domain of each variable consists of the sizes the steel mill is able to produce and 0, designating that the slab is not needed to fulfil the order. Additionally, one variable for each order is required. The domain consists of the identifiers of the slabs the order may be assigned to. The constraints require each order to be assigned to a slab and the sizes of the slabs to be at least as big as the sum of the sizes of the orders assigned to it. The “goodness” of a solution is the sum of the sizes of the produced slabs minus the sum of the sizes of the orders.

An example is given in figure 8. The steel mill can produce slabs of size 5, 4, and 2.

Definition 17 (Constrained Optimisation Problem). A *constrained optimisation problem* $P = \langle X, D, C, f \rangle$ is a constraint satisfaction problem with an objective function f which determines the “goodness” of a solution.

The solution to the example pictured in figure 8 is given in figure 9. The orders

		TIME	SPACE	
		COMPLEXITY	COMPLEXITY	REFERENCE
fine	AC 4	$O(em^2)$	$O(em^2)$	[MH86]
grained	AC 5	$O(em)$	$O(em^2)$	[HDT92]
	AC 6	$O(em^2)$	$O(em)$	[BC94]
	AC 7	$O(em^2)$	$O(em)$	[BFR99]
	AC 8	$O(em^3)$	$O(n)$	[CJ98]
coarse	AC 3	$O(em^3)$	$O(e + nm)$	[Mac75]
grained	AC 3.1/2001	$O(em^2)$	$O(em)$	[BR01] [YY01]
	AC 3.2	$O(em^2)$	$O(em)$	[LBH03]
	AC 3.3	$O(em^2)$	$O(em)$	[LBH03]

e is the number of edges in the constraint graph, m is the maximum domain size, and n is the number of variables.

Table 1. Overview of Arc Consistency Algorithms for Binary Constraints

TYPE OF CONSISTENCY	TIME COMPLEXITY	REFERENCE
arc consistency of binary decomposition	$O(n^2)$	[Hen89]
bounds consistency	$O(n \log n)$, $O(n)$ in special cases	[Pug98] [MT00] [LOQTvB03]
range consistency	$O(n^2)$	[Lec96]
hyper-arc consistency	$O(m\sqrt{n})$	[Rég94]

n is the number of variables involved in the all different constraint and m is the maximum of the cardinalities of the domains.

Table 2. Overview of Consistency Algorithms for the All Different Constraint

are packed onto two slabs of size 4 each.

Definition 18 (Solution to a Constrained Optimisation Problem). A *solution* to a constrained optimisation problem P is a solution to the contained constraint satisfaction problem which maximises or minimises the objective function f .

The solution to the example problem does not produce any waste, therefore, it is *optimal*.

Definition 19 (Optimal Solution to a Constrained Optimisation Problem). An *optimal solution* to a constrained optimisation problem P is a solution to the constrained optimisation problem for which the objective function f takes a global extremum.

Solving constrained optimisation problems is much more difficult than solving constraint satisfaction problems – not only a *solution*, but the *optimal solution*, or at least a solution which is good enough, has to be found. To reduce the size of the search tree, the notion of a *lower bound* is introduced. If a subtree can not contain any solution which is better than the current best one, it can be skipped [IMMH83]. The lower bound specifies the “goodness” a partial assignment must have for the subtree to be explored. Existing algorithms to solve constraint problems can be extended to

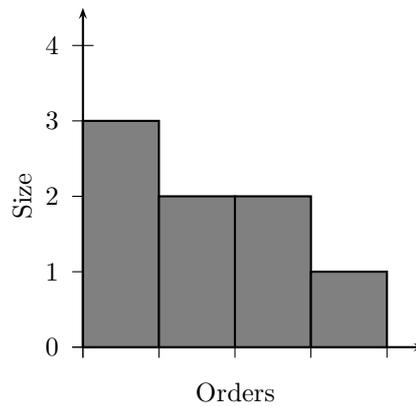


Figure 8. Example Orders for the Steel Mill Slab Problem

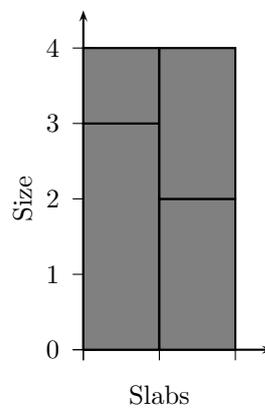


Figure 9. Solution to the Steel Mill Slab Problem in Figure 8

provide such bounds [SFV95] [DKLo1], but the quality of the bounds is often poor and leads to a lot of unnecessary work [BO03].

There are more sophisticated algorithms to determine better bounds, e.g. Russian Doll Search [VLS96], which solves increasingly large subproblems starting with a problem containing only the last variable.

The definitions given so far provide sufficient means to represent and solve numerous problems as constraint satisfaction or constrained optimisation problems. In some cases however, the definitions are too limited to support the appropriate representation of a problem.

2.4.1 EXTENDED CONSTRAINED OPTIMISATION PROBLEMS

Constrained optimisation problems allow to specify a function which determines the “goodness” of a solution (cf. definition 17). The function is limited to giving static values for assignments though; it cannot return different values depending on which variable x_i has been assigned which value $d_j \in D_i$.

The definition of an extended constrained optimisation problem addresses this issue by retaining the objective function, but introducing an additional function g

which determines the “goodness” of variables.

Definition 20 (Extended Constrained Optimisation Problem). An *extended constrained optimisation problem* $P = \langle X, D, C, f, g \rangle$ is a constrained optimisation problem with a function g which maps each variable $x_i \in X$ to a merit. The objective function f is modified to take the merit of each individual variable into account.

Definition 21 (Solution to an Extended Constrained Optimisation Problem). A *solution* to an extended constrained optimisation problem P is a solution to the contained constraint satisfaction problem which maximises the objective function f .

2.5 SOFT CONSTRAINTS

Constraint satisfaction and constrained optimisation problems require a solution to satisfy all constraints. Many real-life problems however are over-constrained, some constraints are more important than others, or the solution has to be computed in real-time and does not need to be perfect, but good enough. These kinds of problems can be modelled with *soft constraints*.

Definition 22 (Soft Constraint). A *soft constraint* is a constraint which does not necessarily need to be satisfied in a solution.

Soft constraints are a recent development in the constraint programming community and not as well researched as hard constraints. One of the first studies on soft constraints is [Fre89]. Usually, the theoretical framework for soft constraint problems is the Maximal Constraint Satisfaction Problem framework introduced in this study, which attempts to maximise the Where the simple approach of minimising constraint violations is not sufficient, other frameworks have been derived from it. There are also completely different paradigms, such as the introduction of meta constraints to model violation [PRBoo].

For a very in-depth overview of soft constraints, see [Scho5].

There are several different ways to model soft constraints. These approaches include [Baro3]

- hierarchical models [BDFB⁺87] [BMMW89],
- partial models [Fre89],
- models which introduce additional variables and hard constraints to model soft constraints [RPPo2],
- fuzzy, preference, possibilistic, weighted, or valued models [Rut94] [Sch92] [FL93] [DFP94], and
- semiring-based models [BFM⁺96] [BMR⁺99] [BMR97].

In this work, a weighted approach has been chosen. The weights are applied to constraints however, not to allowed tuples as usually assumed. This change allows constraints to be represented intensionally, but restricts the conversion into a semiring-based approach and application of consistency properties [BFM⁺96]. A *valuation structure* is introduced to model the cost of violating constraints.

Definition 23 (Valuation Structure). A *valuation structure* V is a tuple $\langle E, \oplus, \preceq, \perp, \top \rangle$. E is the set of valuations which describe the cost of violating constraints. The valuations are totally ordered by the relation \preceq , the minimum element is \perp , and

the maximum element is \top . E is closed over the binary operation \oplus which is commutative, associative, monotone, and has the neutral element property.

Informally, the basic elements of E are the costs of violation for individual constraints. They can be combined to calculate costs for violations of more than one constraint using the \oplus operator. The minimum element \perp denotes no constraint violations and the maximum element \top denotes that all constraints are violated. The relation \preceq orders the costs of violations, i.e. $A \preceq B$ iff the sum of violation costs in A is less than or equal to the sum of violation costs in B .

Definition 24 (Soft Constraint Network). A constraint network with a valuation structure V is called a *soft constraint network*. The set E of V is the set of levels of the network.

An example of a soft constraint network is given in figure 10. The edges are annotated with the relation that constrains assignments to the connected vertices and the cost of violating the constraint.

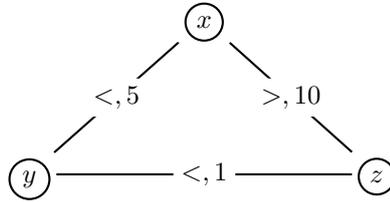


Figure 10. Example Soft Constraint Network Graph for an inconsistent Problem

The basic idea for incorporating soft constraints is to extend constraint problems with a cost of violation for every constraint and a maximum violation for the problem [SFV95]. In the approach chosen in this work, the valuation structure is defined on the system of positive real numbers \mathbb{R}_+ . The minimum element is 0, the maximum element is a value k specific to the particular problem. The ordering relation is the less-than-or-equal-to \leq relation and the binary operator to combine numbers is the $+$ operator.

Definition 25 (Soft Constraint Problem). A *soft constraint problem* P is denoted by the tuple $\langle X, D, C, V, \varphi, v, f \rangle$. X is a tuple $\langle x_1, \dots, x_n \rangle$ of n variables and D is a tuple $\langle d_1, \dots, d_n \rangle$ of n domains. Each domain $d_i \in D$ belongs to the variable $x_i \in X$. C is a set $\{c_1, \dots, c_m\}$ of m constraints over the variables from X . V is a valuation structure $\langle E \subset \mathbb{R}_+, +, \leq, 0, k \rangle$. The function φ is a projection from C to $E \subset \mathbb{R}_+$, i.e. $\forall c \in C : \varphi(c) \in E$ is the valuation of c . The maximum allowed cost of violation for solutions to P is denoted by v . The function f assesses the “goodness” of a solution.

The definition of a solution to a soft constraint problem follows the paradigm of maximal satisfaction of constraints [Fre89].

Definition 26 (Solution to a Soft Constraint Problem). A *solution* to a soft constraint problem P is a complete assignment such that $\sum_{c_i \text{ is unsatisfied}} \varphi(c_i) \leq v$, i.e. the sum of the costs of violation for all unsatisfied constraints $c_i \in C$ is at most as big as the maximum allowed cost of violation v .

The usual definition of weighted soft constraint problems is extended with a limit for the cost of violation any solution may have, v . While soft constraints express preferences and do not necessarily have to be satisfied, solutions may be required

to be of a certain “minimum quality”. Furthermore, there are fewer solutions to a problem and the search space is reduced.

Another difference to the usual definition of soft constraint problems is that a valuation, or cost of violation, is associated with a constraint, not with an allowed tuple of variable assignments.

Soft constraint problems are inherently constrained optimisation problems, as each solution has a cost of violation. The optimisation is split into satisfaction of constraints and optimisation of the solution [SW05].

The notions of local consistency and arc consistency can be extended to soft constraint problems [BMR95] [SFV95] [Schoo]. In most cases, the algorithms to enforce and maintain arc consistency on problems with hard constraints can not be applied without modification to soft constraint problems. There are approaches to modify soft constraint problems to be able to apply arc consistency algorithms for constraint satisfaction problems however [RPBP01] [RPP02]. As weighted soft constraint problems are equivalent to constrained optimisation problems, the search techniques which use lower bounds can be applied as well (cf. section 2.4).

Several algorithms have been developed to use the specific properties of soft constraints to filter domain values. These include the ones described in [PRB01] [CS04] [CdGS07] [vHo4].

2.6 REAL-TIME CONSTRAINT SATISFACTION

In some applications of constraint programming, the time available to find a solution to a problem is limited. Such applications include interactive configurations, monitoring systems, and autonomous devices. The availability of a solution after the time limit elapsed is crucial, sometimes more crucial than satisfying all of the constraints.

Although modern constraint problem solvers are fast and scalable, e.g. Minion [GJM06], and able to deliver solutions quickly, real-time constraint satisfaction is an area of constraint programming where very little research has been done.

The approaches found in the literature add a preprocessing step to the solving of a constraint problem. This usually involves finding a solution to the problem and compiling it into a form which can be used to quickly find the other solutions to the problem in real time [WF99]. Other algorithms prune values from the domains of the variables such that no backtracking is required [BCFR04]. Heuristics can be applied to speed up the solution process [SW98].

All these approaches share a common pattern – in a preprocessing step, which is not limited in time, the problem is “presolved” by removing domain values, adding constraints, computing a seed solution, or similar. Usually, a trade-off between space complexity of the intermediate representation and loss of solutions is involved. In most cases, it is not necessary to find all solutions to a problem, the first one is enough.

In this work, a different and new approach has been chosen. The problem is not known a priori, but generated when a solution is needed. Preprocessing the problem is as difficult as solving it; therefore the preprocessing step is omitted completely. Instead, the real-time requirement is integrated with soft constraints (cf. section 2.5).

Soft constraints are represented in this thesis with a weighted approach. The weights encode preferences. To integrate this with real-time constraint satisfaction, constraints are dropped as time runs out. The constraints with the lowest preference are dropped first. With fewer constraints, there are more solutions, so the time to find a solution decreases. Furthermore, less work has to be done revising constraints. When the time available to solve the problem has elapsed, all the constraints are

dropped and the solver generates a solution by just assigning the first domain value to every unassigned variable.

Dropping constraints to enable real-time solving of constraint problems represents a trade-off between time required to solve a problem and quality of the solution. If the problem is complex and only very little time is available, most of the constraints might be dropped and the quality of the solution might be very poor. If on the other hand the time limit is just below what would be needed to solve the problem without dropping any constraints, this approach provides a solution which is not significantly worse than a solution obtained without time limit instead of no solution at all. The key point is that the solver will always provide a solution within the time bounds. For many applications, this is more crucial than satisfaction of all constraints.

Another important observation about the proposed algorithm is that constraints are dropped while the problem is solved. This means that the constraints which will not be considered in the future have been considered in the past, i.e. they have been revised and domain values have potentially been pruned because of them. The quality of the resulting solution to the problem will therefore be between the quality of a solution which does not consider any of the dropped constraints and a solution which considers all constraints. In some cases, the quality of a solution will not be affected at all when a constraint is dropped because all the values which can be pruned because of it are pruned already.

Before real-time constraint satisfaction problems can be defined, the notions of time and time limit have to be formalised.

Definition 27 (Constraint Satisfaction Time). Let P be a constraint problem and M a machine which is able to solve P . The *time* M takes to solve P is denoted by t_s . The *current time* t is the run time of M since the start of the computation. The *time limit* t_l is an upper bound for t_s on M .

The notion of a machine includes both the implementation of a constraint solver and the hardware the solver is running on. It also incorporates all environment conditions which might have an impact on the run time, e.g. operating system and other programs running at the same time. The constraint solver machine abstracts from specific implementations and computers; all parameters which influence the run must be the same to be able to compare two different machines.

The integration of soft constraint problems and real-time constraint satisfaction is formalised in the following definition.

Definition 28 (Real-time Constraint Problem). A *real-time constraint problem* P is a tuple $\langle X, D, C, V, \varphi, \psi, v, t_l, f \rangle$. X is a tuple $\langle x_1, \dots, x_n \rangle$ of n variables and D is a tuple $\langle d_1, \dots, d_n \rangle$ of n domains. Each domain $d_i \in D$ belongs to the variable $x_i \in X$. C is a set $\{c_1, \dots, c_m\}$ of m constraints over the variables from X . V is a valuation structure $\langle E \subset \mathbb{R}_+, +, \leq, 0, k \rangle$. The function φ is a projection from C to $E \subset \mathbb{R}_+$, i.e. $\forall c \in C : \varphi(c) \in E$ is the valuation of c . The maximum allowed cost of violation for solutions to P is denoted by v . ψ is a projection from $E \subset \mathbb{R}_+$ to a number from the interval $[0..t_l]$ and denotes the time when a constraint of a certain valuation should be dropped, i.e. a constraint c is valid and will be considered iff the current time $t < \psi(\varphi(c))$. The symbol t_l denotes the time available for finding a solution to the problem. The function f assesses the “goodness” of a solution to P .

The definition of a soft constraint problem (cf. definition 25) is extended with the projection ψ and the time limit t_l . ψ is used to determine which constraints need to be considered at time $t_i \in [0..t_l]$ by mapping valuations to points in the interval $[0..t_l]$.

No distinction can be made between different constraints with the same cost of violation and therefore the same validity interval. The constraints are grouped into preference classes, constraints in the same class are not distinguishable in this framework. This is a drawback of the model, but simplifies the handling of the constraints, because apart from the function ψ no extra structures are required and it builds on the soft constraint framework. In most practical applications, constraints with the same cost of violation will be equal to some extent, e.g. expressing a preference for the same property for different sets of variables. Therefore the current model is reasonable despite its simplicity.

The definition of the solution to a real-time constraint problem extends the definition of the solution of a soft constraint problem (cf. definition 26) as well.

Definition 29 (Solution to a Real-time Constraint Problem). A *solution* to a real-time constraint problem P is an assignment S to the variables in X which satisfies the following conditions,

- S is a complete assignment,
- the time t_s required to find S is less than or equal to the time limit t_l , and
- $\sum_{c_i \text{ is unsatisfied}} \varphi(c_i) \leq v + \sum_{c_j: \psi(\varphi(c_j)) \leq t_s} \varphi(c_j)$, i.e. the sum of the costs of violation for all unsatisfied constraints $c_i \in C$ is at most as big as the maximum allowed cost of violation v plus the costs of violation of the constraints dropped during the solution process.

In other words, the third condition requires a solution to fulfil all constraints which are valid at t_s to be satisfied except the ones violated in the framework of soft constraint problems. The maximum allowed cost of violation v is augmented by the cost of violating the invalid constraints $c_j : \psi(\varphi(c_j)) \leq t_s$. Thus a solution S may have a total cost of violation higher than v after constraints have been dropped.

2.6.1 ANALYSIS OF THE FUNCTION ψ

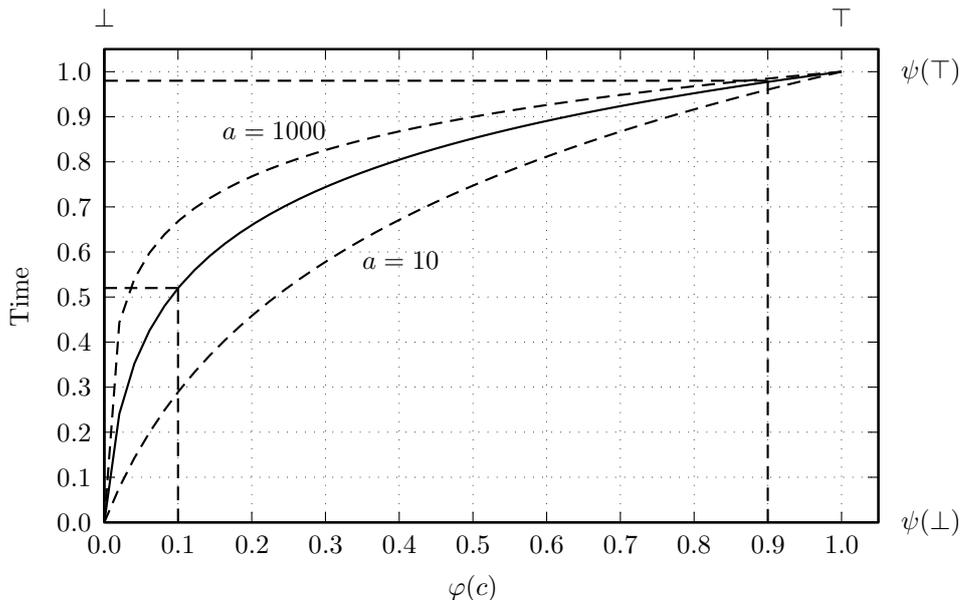
The function ψ is crucial to the definition of real-time constraint problems. It determines when a constraint will be dropped. ψ should have the following properties,

1. injective, i.e. every valuation is mapped to a time,
2. monotonically increasing, i.e. the constraints with a low valuation are dropped first and the constraints with a high valuation last,
3. $\psi(0) = 0$ and $\psi(\top) = t_l$, i.e. at the beginning only constraints with the valuation 0 are dropped and when the time limit is reached all constraints are dropped, and
4. non-linear, i.e. the constraints with a high valuation are dropped when there is only very little time left.

The last property might be optional depending on the problem, the other properties must be satisfied by all functions used in this framework however.

Property 3 assumes that the valuations are positive numbers. The minimum value was specifically chosen to be returned for 0 and not for \perp . If all constraints have a level of preference greater than zero, they should all be considered in the beginning. Furthermore, if $\perp = \top$, all constraints would be dropped before the solving of the problem started.

The function $\psi(x) = \frac{\ln a \cdot \frac{x}{t_1} + 1}{b}$ satisfies above properties. The parameters a and b are problem-specific and ensure properties 4 and 3, respectively. The value of b can be computed, the value of a must be estimated. a controls the steepness of the function curve for a particular time limit t_1 ; a good rule of thumb is $a = 100 \cdot t_1$. Given property 3, b can be computed as $b = \frac{\ln a \cdot \frac{T}{t_1} + 1}{t_1}$. An example is shown in figure 11.



$b = 4.6151205168$ computed with previous parameters. The dashed curves show different values of a .

Figure 11. Example Curve of ψ with $t_1 = 1$, $T = 1$, and $a = 100 \cdot t_1 = 100$

Until half the time available to solve the problem has passed, only the constraints up to a tenth of the valuation of the most important constraint are dropped. The more important constraints are only dropped when more than 90% of the available time has elapsed.

Increasing a leads to a more conservative behaviour, i.e. constraints are dropped later, while decreasing a leads to constraints being dropped earlier.

The model also extends to hard constraint problems. All constraints have the maximum valuation and are dropped when the time to solve the problem is up. The solver proceeds normally until the time limit is reached and then assigns the first value in the respective domain to all unassigned variables.

Real-time constraint satisfaction introduces an important aspect into constraint programming – Quality of Service. The framework guarantees that a complete assignment will be computed within the specified time. It is possible that the time limit is big enough to solve the problem without dropping any constraints. Still the specification of the limit guarantees that, even if the conditions change and more time is required to find a solution, the limit will not be exceeded.

3

CONSTRAINT PROBLEM MODEL

This chapter introduces ways to model the problems described in chapter 1 with constraints. The first simple models will be improved, refined and extended in later sections.

3.1 SLOTS AND CAMPAIGNS

To model a problem as a constraint satisfaction problem, the variables, their domains, and the constraints have to be specified (cf. chapter 2).

Definition 30 (Campaign). A *campaign* c is a component which returns content. The content can be personalised and dynamic or non-personalised and static. A set of campaigns is a *domain* d .

Definition 31 (Slot). A *slot* x is a part of a *page* which can hold exactly one *campaign*. A slot is a *decision variable*.

Definition 32 (Page). A *page* p contains a tuple X of n slots.

Definition 33 (Problem of Rendering a Page). The *problem of rendering a page* p is the constraint satisfaction problem $P = \langle X, D, C \rangle$, where X is the tuple of slots which make up p and D is the tuple of domains. The number of slots and domains is equal. The domain d_i for slot x_i is the set $\{c_1, \dots, c_m\}$ of campaigns which are scheduled in the slot. C is the set of constraints.

The only constraint in this model is the AllDifferent constraint which ensures that there are no duplicate campaigns on the page. Thus, $C = \{\text{AllDifferent}(x \in X)\}$.

A solution (cf. definition 5) to P is an assignment of campaigns to slots such that no campaigns shows up twice.

3.2 VALUES OF CAMPAIGNS

Each campaign has a number associated with it which determines how valuable the campaign is. Introducing this parameter requires only slight modifications of the model, but turns the constraint satisfaction problem into a constrained optimisation problem.

Definition 34 (Value of a Page). The *value of a page* p is determined by the function v ;

$$v(p) = \sum_{x_i \in X} \text{value}(\text{assignment}(x_i)).$$

Definition 35 (Problem of Rendering a Page with Page Value). The problem P of rendering a page can be redefined as $P = \langle X, D, C, v \rangle$, where v is the objective function to maximise. The set of constraints C and the set of domains D remain the same.

The extension of the existing model with the introduction of values has changed the type of the problem. It is no longer enough to find campaigns for all slots which satisfy the constraint, the value of the page has to be optimised. Therefore, finding one solution to the problem does not mean that it can be returned. There may be a better solution. The best solution remains unknown until the search space is exhausted, which adds significantly to the required resources.

The search for solutions can not terminate before every possible solution has been explored because the value of the objective function has to be maximised and there is no upper bound. If the maximum value was known, the search could terminate as soon as this value was reached because any other solution could not be better, only as good. Determining the maximum value of a page is as difficult as solving the problem of assigning campaigns to slots however.

3.3 VALUES OF SLOTS

The association of values with slots can not be adequately represented by classical constraint optimisation problems. They do not allow values which determine the importance or usefulness of uninstantiated decision variables. The model has to be changed to extended constrained optimisation problems (cf. definition 20).

Definition 36 (Value of a Page). The *value of a page* p is determined by the function v ;

$$v(p) = \sum_{x_i \in X} \text{value}(\text{assignment}(x_i)) \cdot \text{value}(x_i).$$

Definition 34 has been changed to include the value of the slot x_i , $\text{value}(x_i)$, in the calculation of the value of the page.

The definition of the problem p remains unchanged. The addition affects only the function v .

3.3.1 EXAMPLE

Let p be a page consisting of three slots;

$$X = \langle \text{left}, \text{center}, \text{right} \rangle.$$

Let there be three campaigns;

$$d = \{\text{recentlyViewed}, \text{recommendations}, \text{cerealPromotion}\}.$$

All three campaigns are scheduled in each of the three slots;

$$D = \langle d, d, d \rangle.$$

Let the value of the slots be

$$\text{value}(\text{left}) = 0.2$$

$$\text{value}(\text{center}) = 1.0$$

$$\text{value}(\text{right}) = 0.5.$$

and the value of the campaigns

$$\begin{aligned} \text{value}(\text{recentlyViewed}) &= 1.0 \\ \text{value}(\text{recommendations}) &= 0.5 \\ \text{value}(\text{cerealPromotion}) &= 0.2. \end{aligned}$$

The set of constraints is

$$C = \{\text{AllDifferent}(\text{left}, \text{center}, \text{right})\}.$$

Figure 12 shows the constraint network graph for this problem.

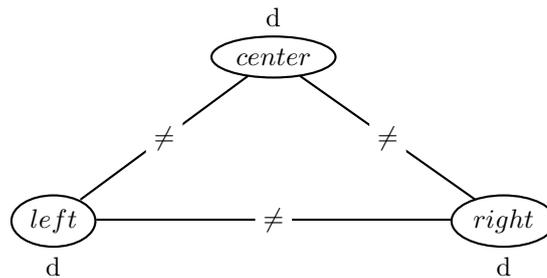


Figure 12. Constraint Network Graph for Example 3.3.1

A complete assignment of campaigns to slots which satisfies C is

$$\begin{aligned} \text{assignment}(\text{left}) &= \text{cerealPromotion} \\ \text{assignment}(\text{center}) &= \text{recentlyViewed} \\ \text{assignment}(\text{right}) &= \text{recommendations}. \end{aligned}$$

Then the value of the page is

$$v(p) = (0.2 \cdot 0.2) + (1.0 \cdot 1.0) + (0.5 \cdot 0.5) = 1.29.$$

No combination of slots and campaigns will give a greater value, therefore this solution is optimal.

3.4 RELAXATION OF CONSTRAINTS

The model requires the campaigns which are shown in each slot to be different from all other displayed campaigns. There might still be duplicate products in different campaigns, e.g. in personal recommendations and bestsellers if the customer is interested in bestsellers. On the other hand, there might be enough products to promote for one campaign to display it in more than one slot without duplicate products. The resulting page might even have a higher value than a page where all campaigns are different.

The following section relaxes the constraint of difference for slots and campaigns.

Definition 37 (Relaxed Constraint). A *relaxed constraint* is a soft constraint (cf. definition 22). The relaxation or importance of the constraint is defined through a valuation structure V (cf. definition 23) and a function φ which maps the constraint to a valuation in $E \in V$.

Introducing relaxed constraints changes the type of problem to be solved into a valued soft constraint problem (cf. definition 25).

Definition 38 (Problem of Rendering a Page with relaxed Constraints). The problem P of rendering a page is redefined as $P = \langle X, D, C, V, \varphi, f, v \rangle$, where V is the valuation structure, φ the function which maps constraints to their valuation, and f the maximum cost of constraint violations allowed for a solution.

The value of the page is redefined to account for violations of constraints.

Definition 39 (Value of a relaxed Page). The *value of a relaxed page* p is determined by the function v ;

$$v(p) = \sum_{x_i \in X} \text{value}(\text{assignment}(x_i)) \cdot \text{value}(x_i) - \sum_{c_i \text{ is unsatisfied}} \varphi(c_i).$$

In other words, the value of the page determined by the assignment of campaigns to slots is reduced by the sum of the cost of all constraint violations.

The constraint $\text{AllDifferent}(x \in X)$ is relaxed and transformed into a soft constraint.

The transformation from a constrained optimisation problem into a valued soft constraint problem increases the complexity of the model further, but also its expressiveness. It is now possible to provide a solution to over-constrained problems, i.e. there are fewer campaigns than slots and the constraint of difference can not be satisfied. The improved model also enables to express preferences if there are many constraints involved. The uniqueness of campaigns is not as important as the uniqueness of the promoted products, for example.

With the current model, soft constraints are of limited use because there is only one constraint. The following sections introduce new constraints and make use of the features of the extended model.

3.5 DUPLICATE CONTENT

The main constraint when generating a page is that all displayed products should be different. Duplicate promotions impair the user experience and waste space which could be used to promote other products. The approach taken in the model so far was to require the difference of all campaigns displayed on the page. The following section extends this model to take product uniqueness into account.

Definition 40 (Places in Slots). The *places in a slot* x_i are defined by the tuple $h_i = \langle k_i^1, \dots, k_i^n \rangle$ of n places for products. For each slot x_i , n new decision variables x_i^1, \dots, x_i^n are introduced, one for each place. The assignment to a place x_i^j is a product o_i , which is displayed at that place. The domain of each place x_i^j is the set $S = \{o_1, \dots, o_m\}$ of the products which can be promoted by any campaign.

The function to assess the value of a page remains unchanged. The value of promoting different products will be different, but quantifying this is very difficult. The assessment through the campaign value provides a good approximation and does not complicate the model further.

Using the definition of campaign contents, the constraint of difference for all products displayed can be formalised.

In addition to the soft constraint $\text{AllDifferent}(x_i \in X, x_i \text{ is a slot variable})$, the hard constraint $\text{AllDifferent}(x_j \in X, x_j \text{ is a place variable})$ is introduced. Hard constraints are modelled in the framework of valued soft constraints by assigning a valuation $\varphi(c)$ to them which is above the maximum allowed cost of constraint violations f for the problem P . In other terms, the model expresses the *preference* to have a different campaign displayed in each slot and the *requirement* to have distinct products promoted on all places of a page.

The current model misses the relation between the products promoted in a slot and the campaign scheduled in it. The two constraints of difference do not ensure that a product is promoted in a slot where a campaign which promotes the product is displayed.

To establish the missing link, binary constraints for every slot - place (x_i, x_i^j) pair are introduced. For every campaign c_i there exists a set $S_i = \{o_1, \dots, o_n\}$ of n products which the campaign may promote. The new constraints limit the domains of the places depending on the campaign which has been assigned to the slot.

Definition 41 (Allowed Tuples for a Slot - Place Pair Constraint). The set of allowed tuples T for the binary constraint on the pair of slot x_i and place x_i^j is defined as

$$T = \{\langle c_k, o \rangle \mid c_k \in d_i, o \in S_k\}.$$

In other words, the set of allowed tuples is constituted by every possible campaign - product combination for the slot, i.e. for all campaigns scheduled in the slot, all the products which may be promoted by the particular campaign are allowed.

When a campaign is assigned to a slot, the domain of the places in that slot is reduced to the products which may be promoted through the assigned campaign.

3.5.1 EXAMPLE

Let p be a page consisting of three slots;

$$X_s = \langle \text{left}, \text{center}, \text{right} \rangle.$$

Each slot has three different places for product displays. The variables are

$$X = \langle \text{left}, \text{center}, \text{right}, \\ \text{left}_1, \text{left}_2, \text{left}_3, \\ \text{center}_1, \text{center}_2, \text{center}_3, \\ \text{right}_1, \text{right}_2, \text{right}_3 \rangle.$$

Let there be three campaigns;

$$d_c = \{\text{recentlyViewed}, \text{recommendations}, \text{cerealPromotion}\}.$$

All three campaigns are scheduled in each of the three slots. The sets of the products which each campaign may promote is

$$S_{\text{recentlyViewed}} = \{\text{book1}, \text{book2}, \text{book3}, \text{book4}, \text{book5}, \text{book6}\} \\ S_{\text{recommendations}} = \{\text{dvd1}, \text{dvd2}, \text{dvd3}\} \\ S_{\text{cerealPromotion}} = \{\text{cereal1}, \text{cereal2}, \text{cereal3}\}.$$

The set of all products which may be promoted is

$$d_s = S_{\text{recentlyViewed}} \cup S_{\text{recommendations}} \cup S_{\text{cerealPromotion}}$$

The domains of the variables are

$$D = \langle d_c, d_c, d_c, d_s, d_s, d_s, d_s, d_s, d_s, d_s, d_s \rangle$$

Let the value of the slots be

$$\begin{aligned} \text{value}(\text{left}) &= 0.2 \\ \text{value}(\text{center}) &= 1.0 \\ \text{value}(\text{right}) &= 0.5. \end{aligned}$$

and the value of the campaigns

$$\begin{aligned} \text{value}(\text{recentlyViewed}) &= 1.0 \\ \text{value}(\text{recommendations}) &= 0.5 \\ \text{value}(\text{cerealPromotion}) &= 0.2. \end{aligned}$$

The constraints which link displayed products to assigned campaigns are specified by the allowed tuples;

$$\begin{aligned} C_{sp} = \{ & \langle \text{left}, \text{left}_1, \{ \langle \text{recentlyViewed}, \text{book1} \rangle, \\ & \langle \text{recentlyViewed}, \text{book2} \rangle, \\ & \langle \text{recentlyViewed}, \text{book3} \rangle, \\ & \langle \text{recentlyViewed}, \text{book4} \rangle, \\ & \langle \text{recentlyViewed}, \text{book5} \rangle, \\ & \langle \text{recentlyViewed}, \text{book6} \rangle, \\ & \langle \text{recommendations}, \text{dvd1} \rangle, \\ & \langle \text{recommendations}, \text{dvd2} \rangle, \\ & \langle \text{recommendations}, \text{dvd3} \rangle, \\ & \langle \text{cerealPromotion}, \text{cereal1} \rangle, \\ & \langle \text{cerealPromotion}, \text{cereal2} \rangle, \\ & \langle \text{cerealPromotion}, \text{cereal3} \rangle \} \}, \\ & \dots \}. \end{aligned}$$

The set of constraints is

$$\begin{aligned} C = \{ & \text{AllDifferent}(\text{left}, \text{center}, \text{right}), \\ & \text{AllDifferent}(\text{left}_1, \text{left}_2, \text{left}_3, \text{center}_1, \text{center}_2, \text{center}_3, \\ & \text{right}_1, \text{right}_2, \text{right}_3) \} \cup C_{sp}. \end{aligned}$$

The valuation of the first constraint is $\varphi(\text{AllDifferent}(\text{left}, \text{center}, \text{right})) = 0.2$ and the maximum cost of constraint violations allowed is $f = 0.5$.

A complete assignment which satisfies C is

```

assignment(left) = cerealPromotion
assignment(center) = recentlyViewed
assignment(right) = recommendations
assignment(left1) = cereal1
assignment(left2) = cereal2
assignment(left3) = cereal3
assignment(center1) = book1
assignment(center2) = book2
assignment(center3) = book3
assignment(right1) = dvd1
assignment(right2) = dvd2
assignment(right3) = dvd3.

```

There are no constraint violations. The value of the page is

$$v(p) = (0.2 \cdot 0.2) + (1.0 \cdot 1.0) + (0.5 \cdot 0.5) - 0 = 1.29.$$

A solution to the problem which involves constraint violations is

```

assignment(left) = recommendations
assignment(center) = recentlyViewed
assignment(right) = recentlyViewed
assignment(left1) = dvd1
assignment(left2) = dvd2
assignment(left3) = dvd3
assignment(center1) = book1
assignment(center2) = book2
assignment(center3) = book3
assignment(right1) = book4
assignment(right2) = book5
assignment(right3) = book6.

```

The value of the page is

$$v(p) = (0.5 \cdot 0.2) + (1.0 \cdot 1.0) + (1.0 \cdot 0.5) - 0.2 = 1.4.$$

Despite the violation of one constraint, the value of this page is larger than the value of the page with no constraint violations. The most valuable campaign is displayed twice and the least valuable campaign is not displayed at all.

3.6 FORCED PROMOTIONS

In some situations an online retailer might wish to promote a specific set of products regardless of the value of the campaign. This is especially useful for new products for which no value is known. The requirement can be modelled with the one-of-equals constraint, which requires at least one of a set of variables to be assigned a specific value.

3.6.1 EXAMPLE

Let p be a page consisting of three slots;

$$X = \langle \text{left}, \text{center}, \text{right} \rangle.$$

Let there be four campaigns;

$$d = \{\text{recentlyViewed}, \text{recommendations}, \text{wishlist}, \text{cerealPromotion}\}.$$

All four campaigns are scheduled in each of the three slots;

$$D = \langle d, d, d \rangle.$$

Let the value of the slots be

$$\text{value}(\text{left}) = 0.2$$

$$\text{value}(\text{center}) = 1.0$$

$$\text{value}(\text{right}) = 0.5.$$

and the value of the campaigns

$$\text{value}(\text{recentlyViewed}) = 1.0$$

$$\text{value}(\text{recommendations}) = 0.5$$

$$\text{value}(\text{wishlist}) = 0.7$$

$$\text{value}(\text{cerealPromotion}) = 0.2.$$

The retailer wishes to promote the new cereal section of the shop, therefore the set of constraints is

$$C = \{\text{AllDifferent}(\text{left}, \text{center}, \text{right}), \\ \text{OneOfEquals}(\{\{\text{left}, \text{center}, \text{right}\}, \text{cerealPromotion}\})\}.$$

An assignment of campaigns to slots that satisfies C is

$$\text{assignment}(\text{left}) = \text{cerealPromotion}$$

$$\text{assignment}(\text{center}) = \text{recentlyViewed}$$

$$\text{assignment}(\text{right}) = \text{wishlist}.$$

The value of the page is

$$v(p) = (0.2 \cdot 0.2) + (1.0 \cdot 1.0) + (0.5 \cdot 0.7) = 1.39.$$

If recommendations was shown instead of cerealPromotion, the value of the page would be higher. The one-of-equals constraint however requires the campaign cerealPromotion to be part of any solution.

3.7 REAL-TIME PROBLEM SOLUTION

The model so far takes various functional constraints into account. There are non-functional constraints on the generation of a page as well. The time required to compute a solution to the problem is limited, for example. The attention span of the customer is short, so a page has to be generated almost instantly. This section extends the model with a limit on the time taken to solve the problem P .

Real-time constraint satisfaction is considered in the framework described in section 2.6.

Definition 42 (Problem of Rendering a Page with Time Limit). The problem P of rendering a page is redefined as $P = \langle X, D, C, V, \varphi, \psi, f, t_l, v \rangle$ (cf. definition 28). The function ψ maps each constraint $c \in C$ to a time when it is dropped. t_l denotes the time limit.

The introduction of real-time constraint satisfaction to the model extends the expressiveness to non-functional constraints which are ubiquitous and important in numerous industry applications. The current model is complete enough to mirror real applications accurately.

Part II

PROTOTYPICAL IMPLEMENTATION

4

OVERVIEW

In this chapter, the implementation of a system to render websites using constraint programming techniques is introduced. A high-level overview of the system architecture will be given, the interfaces between the parts will be explained, and the main implementation decisions discussed. The following chapters will describe each component in detail.

4.1 SYSTEM ARCHITECTURE

The system consists of the following parts:

- the constraint problem solver library (see chapter 5),
- the [SOAP Web Service](#) which provides an interface to the library (see chapter 6), and
- the web user interface to the service (see chapter 7).

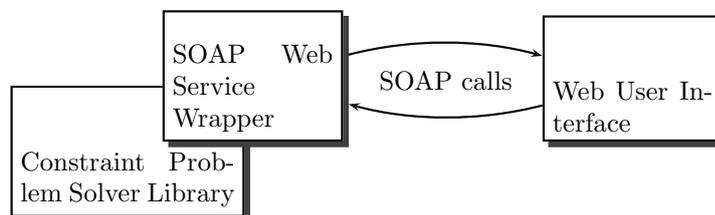


Figure 13. System Architecture of prototypical Implementation

4.1.1 DISTRIBUTED APPROACH

The distributed approach is considered best because it reflects real systems in industry and supports [Service-oriented Architecture](#) designs. The main advantage over a monolithic implementation is that the components can easily be substituted with components of similar functionality and the system can be distributed across a network to provide high performance, reliability and availability.

Distributed systems avoid single points of failure and make better use of heterogeneous resources scattered over different locations. The individual components are used as building blocks of the system. Each module can be a part of several systems. The resulting structures are easier to manage and more flexible than traditional monolithic systems.

4.1.2 WEB INTERFACE

The choice of a web interface to the system seemed natural because its intention is to render websites. A high-level demonstration of the workings is the purpose of the user interface. It is not an integral part of the system, but an addition to present the results and aid understanding.

Web interfaces have the additional advantage that no special software is required to interact with them; every modern operating system comes with a web browser. Nothing needs to be installed to make it work and the ubiquitousness of the World Wide Web has produced many frameworks for web user interfaces which save developer work.

Although there are a number of drawbacks for web interfaces, they do not impact the purpose of this system and are not considered further.

4.1.3 WEB SERVICE

The building blocks of the system – constraint problem solver library and user interface – are tied together by a web service. This choice reflects the earlier choice to implement a distributed system. There is no single standard for web services, but a number of different approaches. XML based protocols are most widely used and supported.

	SOAP	REST	XML-RPC
query structure	complex	simple	complex
implementation complexity	high	low	low
infrastructure integration mechanisms	WSDL	-	-

Table 3. XML based Web Service Protocols considered for the Web Service

Several different protocols were considered for the implementation of the web service. SOAP was chosen because the queries to the service – i.e. the problem definitions – can become arbitrarily complex, and [Representational State Transfer \(REST\)](#) only provides facilities for relatively simple queries. Related technologies such as [XML-RPC](#) do not provide comparable means for integration with existing infrastructures.

SOAP allows to publish a [Web Services Description Language \(WSDL\)](#) document containing service definitions. It enables clients to use the service with only minimal setup costs as well as being aware of changes to the interface.

Implementation complexity was not considered at all, because such an implementation would be beyond the scope of this thesis. Instead, existing libraries are used.

4.2 IMPLEMENTATION LANGUAGE

The whole system is implemented in Ruby¹.

“Ruby is a dynamic, open source programming language with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write.”

¹ <http://www.ruby-lang.org/>

Ruby combines features from different programming paradigms, such as functional programming elements, object orientation, and [domain specific languages](#). The wide range of libraries and frameworks available makes it especially easy to develop web-based and service-oriented applications. Therefore Ruby is an excellent choice for all parts of the system.

It was chosen because it enabled to develop a homogeneous system without much implementation-specific overhead or setup costs. Furthermore, there are facilities for packaging, code distribution, and dependency management.

Another reason for choosing Ruby was the ability to extend the language at runtime. Existing classes can be modified, features added, and behaviour modified in place without the need to implement the changes in new classes which inherit from the base classes. This is especially important for the handling of lists – the existing Ruby classes could be used and required additional functionality added with minimal effort (cf. section [5.2.13](#)).

Ruby focuses on speed of implementation, not on runtime speed. Therefore, similar applications implemented in other programming languages often run faster. As the purpose of this thesis is to provide a prototypical implementation for further evaluation and not a production-ready system, this drawback is not considered to be a disadvantage.

The decision to implement all the parts of the system in the same programming language was due to simplicity reasons. Systems deployed in industry are often written by different teams in different programming languages, so this decision may impair the comparability to real-life applications, but the benefits for the speed of implementation are not negligible.

4.2.1 DOCUMENTATION

To keep documentation concise and up-to-date, Ruby's in line documentation format, RDoc², is used. Classes and methods are documented in the source code, no separate files are required. The adjacency of code and documentation makes it easy to keep both synchronised, document the important aspects, and aids understanding of the code by others.

All of the most important methods are documented. Documentation is omitted where the purpose of the method is clear, e.g. an overwritten inherited method such as the iterator `each`, or the method is trivial.

The tools that come with RDoc provide means to extract the documentation from the source files, organise it, and add navigation elements. Output to a variety of formats, including [HTML](#), XML, and Ruby's own documentation format, is supported.

4.2.2 TESTING

For testing, Ruby's `Test::Unit` unit-test framework is used. The tests can be run as regression tests to ensure integrity after code modifications, to specify desired and undesired behaviour, and determine the functional interface. Tests assure the correct implementation of algorithms as well as proper error handling and other robustness measures.

² <http://rdoc.sourceforge.net/>

4.2.3 PACKAGING

RubyGems³ is used to package the components into distributable files and manage dependencies of parts on libraries and between them. This package management system is widely used by Ruby developers and many libraries are available as [gems](#). The use of a standard system eliminates the need to do explicit dependency checking in the code and makes installation easier for the end-user.

To manage the creation of gems, tests, and documentation, Rake⁴ is used. Rake is a system similar to UNIX make, but implemented in Ruby and for Ruby applications. Similar to Makefiles, Rakefiles define tasks which can be run with Rake. For example, the information needed to create a gem can be specified in a Rakefile, and the gem is created by calling the corresponding Rake task.

The constraint solver library gem additionally provides a script which can be run stand-alone on problem descriptions. The script does not provide any facilities to the library, it simply evaluates the problem description which must be given in Ruby code. It is only provided for completeness and not as a substitute for a user interface.

Example input files with problem descriptions are also included in the constraint solver gem.

4.3 VERSION CONTROL SYSTEM

Subversion⁵ is used as [revision control system](#). A revision control system provides code repository management facilities, enables developers to revert to previous versions of the code if changes break it, and compare different revisions of files. Subversion was chosen because it is widely used and stable, but provides more features than older systems.

The revision control system is not only used for the source code, but for all parts of the thesis, including this document. It serves as a single point of reference for the whole project. All the files required to build each part of the system are put under revision control.

Subversion provides facilities for actions to run at various stages of a [commit](#). For this project, a post-commit hook for the code part is implemented.

Every time a code change is committed to the repository, all the tests are run on the changed code. If no errors occur, the latest version of the software is built and deployed to the test machine. If one or more tests fail, the new version will not be deployed but a transcript of the failed tests will be sent to the developer. This assured that the latest stable version of the software is always available for testing and defects are noticed immediately.

4.4 TEST MACHINE SETUP

In addition to the machine the project is developed on, another machine is set up for testing purposes. This assures that the system is easily deployable to a target machine. As the test machine is connected to the internet, testing by more people is made possible.

The SOAP web service wraps around the constraint problem solver library and runs as a standalone server. Both the solver library and the web service are installed as Ruby gems.

³ <http://www.rubygems.org/>

⁴ <http://rake.rubyforge.org/>

⁵ <http://subversion.tigris.org/>

The web user interface is integrated with the Apache webserver⁶ and runs as **FastCGI** processes. It communicates with the SOAP web service through SOAP service calls.

The test machine is a standard PC with an Intel Pentium II CPU running at 400 MHz and 512 MB memory. It is running Gentoo Linux and also hosts the code repository.

⁶ <http://httpd.apache.org/>

CONSTRAINT PROBLEM SOLVER LIBRARY

This chapter introduces the implementation of the constraint problem solver. An overview of the architecture will be given and the parts described and explained. The algorithms and concepts behind the solver will be described and discussed.

The solver library constitutes the main part of the prototypical implementation. No existing libraries were used, the whole system was implemented from scratch. An overview of existing constraint problem solvers and reasons for the decision not to use any of them but to implement a new solver will be given in the next section.

The implementation mirrors the definitions from chapter 2. It is intended to be as general as possible to support the solution of a wide range of constraint problems. The description of the implementation concludes with a list of limitations.

5.1 OVERVIEW OF EXISTING CONSTRAINT PROBLEM SOLVERS

The constraint problem solving library used in the prototypical implementation is not an adaption or extension of an existing solver, but an entirely new project. Various reasons led to the decision to implement a new constraint programming system.

An overview of existing solvers is given in table 4. It is not meant to be exhaustive; there are many more solvers available. A more elaborate comparison of some solvers is given in [Dun93]. The study [Bar99] presents an overview of limitations and shortcomings of existing systems. A list of constraint problem languages and solvers can be found at <http://4c.ucc.ie./web/archive/solver.jsp>.

The only systems to provide support for soft constraints are Michel Lemaitre's Library and Choco. Lemaitre's Library does not support constrained optimisation problems though and limits the arity of constraints. Choco provides only rudimentary support for weighted soft constraint problems. All other solvers only support hard constraint problems. Soft constraints have to be supported by the solver of the prototypical implementation. None of the solvers supports real-time constraint satisfaction, which is also an important part of the solver library. Therefore, any solver would have to be modified.

Modification of the existing code is only possible for the constraint problem solvers with an open source license, i.e. the ones in the first part of table 4. Most of those are very complex systems with large amounts of code and features which are not relevant to the requirements of the prototypical implementation. To modify the code, one needs to be familiar with it. Gaining familiarity with the code of any of the existing implementations would be a huge and difficult task. Most of the surveyed solvers do not have comprehensive documentation of the employed concepts, algorithms, and implementation details. Often unit tests, which aid understanding and enable regression testing, do not exist. Sometimes, the documentation is outdated and the solver not actively maintained anymore.

NAME	LICENSE	LANGUAGE	FOCUS	REFERENCE
Choco	BSD	Java	all-purpose system, explanation-based solving	[Cho]
Comet	GPL	C++	visualisations, domain specific language	[HM05] [Com]
FaCiLe	GPL	OCaml	functional constraint problem solving	[BB01] [BB04]
Gecode	BSD	C++, Java and Ruby bind- ings	modular structure	[Gec]
Michel Lemaitre's Library	-	LISP	valued constraint satisfaction problems	[Lem]
Minion	GPL	C++	fast, scalable solving	[GJM06] [GJM ⁺ 07]
python- constraint	GPL	Python	constraint satisfaction problems	[Pyc]
CHIP	commercial	C, C++, Prolog	industry applications	[BSKC97]
Disolver	commercial	C++	distributed constraint solving	[Hamo6]
ILOG Solver	commercial	C++, Java, .NET	industry applications	[Ilo]
Koalog	commercial	Java	industry applications	[Koa]

Table 4. Overview of Constraint Problem Solvers

Choosing a solver to extend to fulfil the requirements at hand is not a trivial task as well. This decision can only be made after evaluating the code of the implementations and estimating the effort of implementing the additional features. Gathering all the required information is a very labour-intensive and time-consuming task.

The reasons stated above had the biggest influence on the decision to implement a new solver from scratch. The basic data structures and algorithms are easy to implement and not all of the sophisticated and difficult algorithms are required. The extensions described in chapter 2 can be considered when designing the solver architecture. A custom implementation can furthermore be integrated into the overall system architecture described in the previous chapter more easily. Different algorithms can be evaluated and new concepts explored without needing to investigate possible hidden effects of local changes on the solver as a whole.

5.2 ARCHITECTURE

The library is implemented as an object-oriented system, with classes representing the core entities and concepts. Only the methods which are directly needed to

interact are exposed, everything else is kept private to the class. Where possible, interfaces are implemented to abstract concepts from particular instantiations of them. The complexity of performing a particular task is hidden in the implementation of the class performing it, other classes which need it done do not need to worry about how it gets done.

The object-oriented nature of Ruby makes it possible to take advantage of all the points mentioned above. Another one of Ruby's features, the ability to extend the language at run time, is used, too (cf. section 5.2.13). During the implementation, it became apparent that choosing Ruby as implementation language was a very good decision. The conciseness of the syntax makes the code easy to read and understand, the flexibility enables to solve problems with less code than other programming languages, and excellent support for tests and debugging supports early and efficient finding of bugs.

Some classes are derived from standard Ruby classes to avoid reimplementing existing functionality. Other parts use [reflection](#) to delegate method calls. For some classes, operators are overloaded.

The structure of the library is illustrated in figure 14. The diagram represents the most important classes of the implementation. It is not intended to give an exhaustive overview of the classes or their methods, but rather to illustrate the explanation of the parts of the system. The associations between the classes are annotated and quantified. They are not meant to show all associations but to identify the most important relationships between the entities and concepts. The semantics of the diagram is similar to [UML](#) class diagrams [[UML](#)].

The following sections will explain each of the parts depicted in the diagram. First, the most basic parts of the system are dealt with. They will be gradually assembled into the more complex components. At the end, the high-level algorithms used to solve constraint problems will be discussed.

5.2.1 DOMAIN

The class `Domain` represents a domain which consists of a set of values. It provides methods to

- add and remove values from the set,
- test for inclusion of values, and
- test how many and whether there are any values at all in it.

During the process of solving a constraint problem, values are by and by removed from the domain – the domain is pruned. When a dead-end is spotted, the values have to be reinserted to be able to attempt to solve the problem again. Therefore, a stack which saves the state of the domain when it is pruned and a method to undo the pruning are implemented.

Duplicate values in a domain are avoided by using the Ruby class `Set`, which ensures that every single value occurs at most once.

Several other methods are implemented to allow interaction with the underlying set of values. All these methods implement the proxy pattern [[GHJV94](#)]. They hide the internal implementation of the domain values by providing methods to directly interact with them, regardless of the representation.

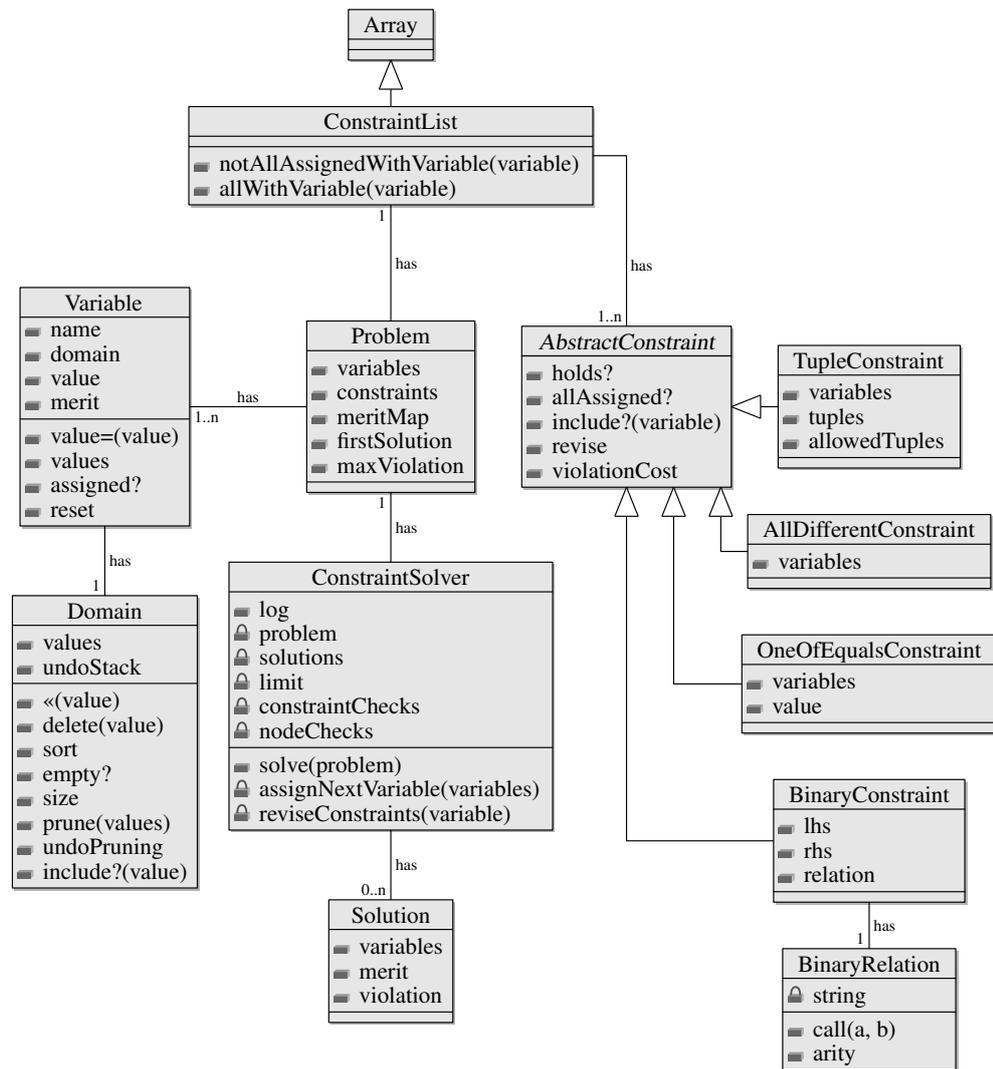


Figure 14. Structure of the Constraint Solver Library

5.2.2 VARIABLE

The class `Variable` represents a variable. It has

- a name,
- a domain,
- a value which is currently assigned to it, and
- a merit towards the solution of a problem.

The implementation provides methods to

- retrieve the set of values which can be assigned to the variable from its domain,
- assign a value to it,

- check whether the variable has a value assigned to it, and
- reset the assignment to a null value.

The operator which assigns a value to the variable checks whether its domain contains the value. If the value is not in the domain, an error is raised.

5.2.3 ABSTRACTCONSTRAINT

The class `AbstractConstraint` defines the interface to a constraint. The specific constraints which derive from it will be explained in the following sections.

The methods required for the rest of the library to interface with a constraint are

- a method to check whether the constraint holds,
- a method to check whether all the variables which are involved in the constraint have values assigned to them,
- a method to check whether a given variable is involved in the constraint,
- a method to revise the constraint, and
- a method to retrieve the cost of violating the constraint.

Revising a constraint usually means that for all the variables involved in the constraint all values from their respective domains are checked for whether they can be assigned to the variable such that when all variables have values assigned to them the constraint holds. In other words, all values which are not supported are removed from the domains of their respective variables – local consistency is enforced (cf. definition 12). The strong form of consistency is not always implemented because the process can be very expensive compared to its benefit. Sometimes it is better to enforce a weaker form of consistency where not all the values which could potentially be pruned are pruned. The process of constraint revision is detailed in section 5.4. The `revise` method is expected to return a list of the variables whose domains have been revised and the number of times it was checked whether the constraint holds during the revision process.

The cost of violating a constraint is implemented as an attribute to the class. It represents the mapping to the valuation structure as defined in definition 25. The default value for the cost of violating a constraint is 1.

Ruby has no support for interfaces or abstract classes; every class can be instantiated. Therefore, `AbstractConstraint` is implemented as a regular class. The method stubs raise errors when they are called to ensure that every class which inherits from `AbstractConstraint` – i.e. implements the interface – provides its own proper implementation of the required methods.

5.2.4 BINARYCONSTRAINT

`BinaryConstraint` implements the type of constraint which relates two variables with a binary relation. The relation must return true for the constraint to be satisfied. The members of a binary constraint are

- the variable on the left hand side of the relation,
- the variable on the right hand side of the relation, and
- the binary relation which relates the two variables.

The class `BinaryConstraint` derives from `AbstractConstraint` and implements all methods defined in it. The particulars of the revision of the constraint will be explained in section 5.4.1.

The `holds?` method always returns `true` when not all variables have values assigned to them; it is assumed that the constraint potentially holds. The arity of the relation which relates the two variables is checked when a new binary constraint is created and an error raised if it is different from two.

5.2.5 BINARYRELATION

The class `BinaryRelation` represents a binary relation which is used to relate two variables in binary constraints. It provides a string representation of the relation, a method to return its arity, and a method to call itself, i.e. compute the result of the application of the relation to the two variables it relates.

The `arity` method always returns two and is used during the construction of binary constraints to check that the relation has the correct arity.

The implementation of the `call` method uses the command pattern [GHJV94]. The relation is represented as a string which doubles as a Ruby operator or method, such as `<`, `>`, or `==`. Reflection is used to call the appropriate Ruby method when the relation is called. The use of the command pattern enables the implementation to provide all operators which Ruby supports without specifying them explicitly. All methods the two objects which are related implement may be specified as relation.

Ruby provides a `send` method for all objects, which can be used to send arbitrary messages to the object. In this case, `send` is used to send the name of the method to call and the argument – the second object in the relation – to the first object. This is semantically equivalent to calling the method name on the first object with the second object as an argument, but enables the use of the command pattern by specifying the name of the method at run time. Passing messages which request information or actions to be performed between objects is one of the fundamental principles of object-oriented programming [Boo93].

5.2.6 ALLDIFFERENTCONSTRAINT

`AllDifferentConstraint` is the class which implements a constraint that requires all variables involved in it to have different values. The only member of the class is a list of those variables.

At least two variables need to be involved in the constraint. If the construction of a new `AllDifferentConstraint` object is attempted with no or just one variable, an error is raised.

The `holds?` method checks whether all assigned variables have different values. The unassigned variables are not considered; if no variable has a value assigned, it is assumed that the constraint potentially holds and `true` is returned.

An all different constraint with n variables can be specified as $\frac{n(n-1)}{2}$ binary constraints which require that the values of the variables are not equal. The all different constraint provides “syntactic sugar” and dramatically reduces the number of constraints the user needs to specify. Furthermore, the revision of the constraint can be optimised, because instead of $\frac{n(n-1)}{2}$ independent pieces of information about n variables, one large piece of information which relates all of those variables is available [SW99]. For detailed information on the revision of the all different constraint, see section 5.4.2.

5.2.7 TUPLECONSTRAINT

The class `TupleConstraint` implements a constraint of arbitrary arity which is specified by the tuples of allowed or disallowed assignments. The members of the class are

- the list of variables involved in the constraint,
- the list of tuples, and
- a boolean specifying whether the list of tuples constitutes the allowed or disallowed assignments.

The constructor of the class checks that every tuple specified has exactly one value for every variable involved in the constraint. An error is raised if this requirement is not met.

The `holds?` method returns `false` if at least one variable has a value which is not in the list of allowed tuples. If disallowed tuples have been specified, the method will only return `false` if all variables have values assigned to them and the tuple of their assignments is contained in the list of disallowed tuples.

The details of the revision of the constraint are explained in section [5.4.3](#).

5.2.8 ONEOFEQUALSCONSTRAINT

The class `OneOfEqualsConstraint` implements a constraint which requires at least one of the variables involved in it to be assigned a particular value. The members of the class are the list of variables and the value.

At least one variable needs to be involved in the constraint. If the construction of a new `OneOfEqualsConstraint` object is attempted with an empty list, an error is raised.

The `holds?` method only returns `false` if all variables have values assigned to them and none of the values is equal to the value specified during the instantiation of the constraint. If not all variables have values assigned to them or at least one variable has the value requested assigned to it, `true` is returned.

For detailed information on the revision of the one-of-equals constraint, see section [5.4.4](#).

5.2.9 CONSTRAINTLIST

The class `ConstraintList` inherits from the Ruby base class `Array`. It represents a list of constraints. Ruby does not support static typing, therefore any object may be inserted into the list; there are no measures to prevent this. The list is intended to hold objects which implement `AbstractConstraint`, adding any other object to it will result in run time errors.

Most of the functionality is not implemented in the class itself, but inherited from `Array`. All the methods for list management are available without specifying them explicitly.

`ConstraintList` extends the `Array` base class with methods specific to lists of constraints.

The method `notAllAssignedWithVariable` takes a variable as an argument and returns a `ConstraintList` of those constraints which involve the variable and have at least one unassigned variable involved. In other words, all constraints with a certain variable which need to be revised are returned to the caller. A constraint where all

involved variables have values assigned to them does not need to be revised, even if there are unsupported values left in the domains of the variables. Further revision would be unnecessary work, because the purpose of revision and value pruning is to shrink the set of values which can be assigned to a variable to reduce the search space. Once a particular value has been assigned to a variable, there is no need to search for an assignment anymore and the size of the potential search space is irrelevant.

The method `allWithVariable` takes a variable as an argument and returns a `ConstraintList` with all constraints which involve the variable. No check for assigned and unassigned variables is performed.

The class furthermore implements methods to sort the list of constraints and remove a single constraint from it.

5.2.10 PROBLEM

The class `Problem` represents a constraint problem. It consists of

- a list of variables,
- a list of constraints,
- a mapping from domain elements to their merit towards a solution,
- a boolean indicating whether all solutions or just the first solution to the problem should be determined, and
- a number designating the maximum allowed cost of constraint violations for any solution to the problem.

The implementation follows the definition of a constrained optimisation problem with a set of “goodness” values for domain elements (cf. definition 20) and the definition of a soft constraint problem with a maximum cost of violation (cf. definition 25). The mapping from constraint to cost of violation is encapsulated in the implementation of the specific constraint. The objective function is not specified explicitly; the merit of a solution is always the sum of the merit of the individual variable - assignment pairs (cf. definition 39).

The list of variables must not be empty. Each variable may already have a value assigned to it. The list of constraints is expected to be a non-empty `ConstraintList`. If either the list of variables or constraints is empty, an error is raised.

The map of domain elements to merit is optional. If it is specified, the values are used during the construction of a solution, as detailed in the next section.

The boolean indicating the required number of solutions is optional, too. If it is omitted, all solutions to the problem are computed.

The default value for the maximum cost of constraint violations is 0. If no value is specified when constructing a problem, no constraint violations are allowed.

Problems are the high-level objects which can be passed to the constraint problem solver. They contain all the information needed to specify and solve a problem.

5.2.11 SOLUTION

The class `Solution` represents a solution to a constraint problem, i.e. a complete assignment which has a lower cost of constraint violations than the allowed limit.

A solution consists of a list of all variables specified in the problem description. Each one has a value assigned to it. When a new solution is constructed, all variables

are checked for assigned values. If not all of them have values assigned to them, an error is raised.

The merit of a solution specifies its “goodness”. The value is computed from the merit of the variables which are part of the solution and, if specified, the mapping from domain element to merit. The total merit of a solution is the sum of the merits of all variable - assignment pairs. The merit of a variable - assignment pair is the product of the merit of the variable and the merit of its assignment. If the merit of the assignment is not specified, only the merit of the variable contributes to the merit of the pair (cf. definition 36).

The total cost of constraint violations is another member of the class (cf. definition 26). It influences the merit of the solution – the cost of constraint violations is subtracted from the merit of the solution calculated as outlined above.

5.2.12 CONSTRAINTSOLVER

The class `ConstraintSolver` represents the high-level concept of a constraint problem solver. The input to the solver is a problem definition and the output is a – possibly empty – list of solutions to the problem.

The most important member variables of `ConstraintSolver` are

- a log to provide feedback on the solution process,
- a problem definition,
- a list of solutions,
- the time limit for solving,
- the number of constraint checks performed, and
- the number of node checks performed.

The log uses the `Log4r` library¹. Several different levels of logging are used to provide the operator of the solver with granular feedback, depending on whether general information about a run or detailed debugging data is required.

The problem definition is expected to be a `Problem` class and the list of solutions contains `Solution` classes. The time limit for solving the problem is either a real number designating the time available in seconds, or the boolean `false` designating that there is unlimited time available. Auxiliary member variables keep track of the elapsed time and determine when to drop constraints because time is running out. The number of constraint checks is the total number of times the `holds?` method was called on any constraint in the problem specification. The number of node checks designates how many times a variable was assigned a value from its domain. A node is a vertex in the search tree (cf. figure 5). Vertices represent assignments to variables, edges link subsequent assignments.

`ConstraintSolver` provides methods to

- solve a given problem,
- attempt an assignment to the next variable in a list of unassigned variables, and
- revise the constraints which involve a certain variable.

¹ <http://log4r.sourceforge.net/>

Only the first method is public and meant to be called by users of the constraint problem solver library, the latter two methods are used internally to solve a problem. The actions performed during the process of solving a problem and the specifics of this implementation are beyond the scope of the description of the class and detailed in section 5.3.

5.2.13 RUBY EXTENSIONS

For some of the base classes provided by Ruby, additional functionality is required.

The class `Array` is extended with the following methods,

REST This method returns a new array which contains all elements but the first of the original array.

EACHAFTER This method requires an element of the array and a block of code as arguments. The block of code will be executed for each element of the sub array starting with the first element after the specified element of the current array.

EACHSTARTWITH This method requires an element of the array and a block of code as arguments. The block of code will be executed for each element of the sub array starting with the element specified.

The class `Fixnum` is extended with the `not_equal?` method. This method checks whether a specified argument is not equal to the number. The method addition is required because of the string representation for inequality binary relations (cf. section 5.2.5). The intuitive representation `!=` can not be used as a Ruby method name.

The class `Set` is extended with the `include_any?` method. It checks whether any value contained in the list passed as an argument is also contained in the set. As soon as such a value is found, the method terminates.

The class `Hash` is extended with a method to check if another hash is contained in it, `include_hash?`. The method returns true iff the hash contains all the keys and maps them to the same values as the hash given as an argument.

5.3 CONSTRAINT PROBLEM SOLUTION

This section explains the algorithms employed to solve a constraint problem. A detailed overview of the process will be given and its parts will be illustrated. First, the general algorithm will be explained. Then, the modifications of the algorithm to incorporate soft constraints and real-time constraint satisfaction will be detailed.

5.3.1 SOLUTION PROCESS

To solve a constraint problem, successive assignments to all the variables are attempted, verifying that the constraints hold and revising them after each assignment. The top-level solver procedure operates on a list of variables which are ordered by their merit, highest merit first. The variables may have values assigned to them.

During an assignment step, the following actions are performed,

- the first variable is removed from the list of pending variables,
 - if the variable has a value assigned to it, it is checked whether all constraints hold

- else the domain of the variable is sorted according to the merit of each element if a corresponding mapping has been provided, and all domain values are successively assigned to it
- for each assignment, all constraints which involve the variable are revised,
- if no domain was wiped out and
- there are no pending variables left, the solution is recorded, or
- the next pending variable is handled,
- else nothing is done.

After these steps, the constraint revision is reverted by undoing the pruning of all pruned domains and the current variable is reset to be unassigned. This ensures that the state from before the execution of the assignment step is recreated and subsequent attempts to solve the problem have the same starting point. Then the method returns, stepping one level back in the search tree.

If a solution is found and the solver is only required to find the first solution to a problem, the solver breaks and returns the solution. No further exploration of the search tree or constraint revision is performed.

The basic algorithm implemented is Forward Checking [HE80]. Other approaches to solving constraint problems include Backtracking, where after each assignment the satisfaction of the constraints is checked and the search aborts if all variables have values assigned to them or a constraint is violated. In the latter case, the algorithm “backtracks” by removing the most recent assignment and trying another one. The advantage of Forward Checking over Backtracking is that the size of the search tree is reduced by enforcing consistency.

The type of consistency enforced after each assignment is arc consistency [SF94] [Fre78]. It ensures that the domains of the unassigned variables remain consistent with the new assignment. Values which can not be part of a solution are removed from their domains and do not contribute to the size of the search space anymore.

To revise all the constraints a variable is involved in, the following steps are taken,

- all constraints which involve the variable and at least one variable which has no value assigned to it are assembled into a queue,
- for each constraint in the queue, the revise method is called (cf. section 5.4),
- for all the variables whose domains have been revised,
 - their domain is recorded to be able to undo the pruning later and
 - all constraints which involve the variable and at least one variable which has no value assigned to it are added to the queue of constraints to be revised, unless they are in the queue already or have just been revised.

If during the revision of a constraint the domain of a variable is wiped out, i.e. all the values which have been in the domain are pruned because they are not supported, the wipe out is signalled to the calling method. No further constraint revisions are performed. The current set of assignments can not be part of a solution because there is at least one variable for which no valid assignment exists.

The solution process builds up a search tree which explores the space of possible solutions (cf. definition 9).

The actions performed during the solution process are summarised in figure 15. The details of constraint revision and the reversion of pruning are omitted for simplicity reasons. The figure shows the solution process for a list of unassigned variables.

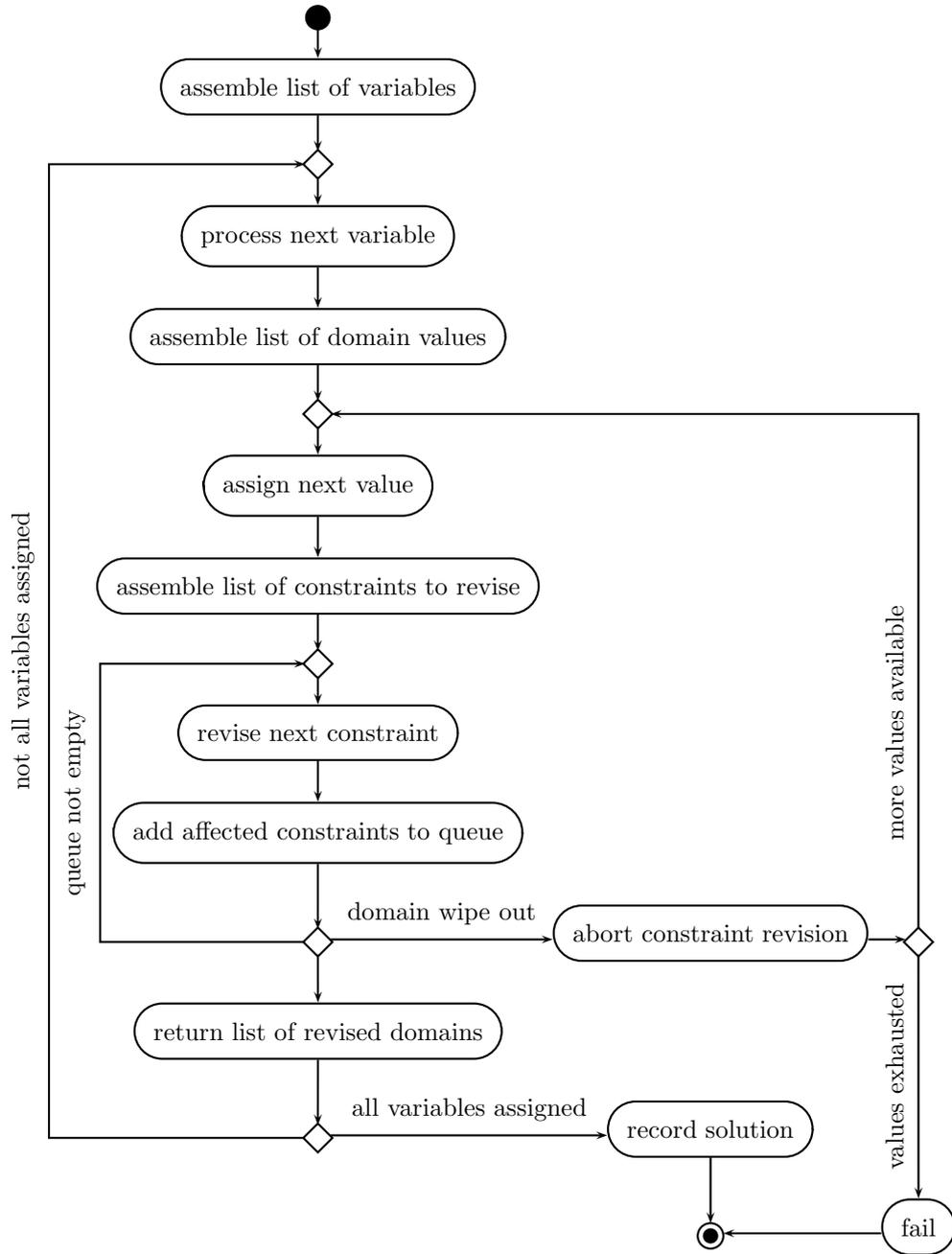


Figure 15. Actions performed during the Solving of a Constraint Problem

5.3.2 MODIFICATIONS FOR SOFT CONSTRAINTS

The implementation of soft constraints does not employ any specific algorithms to filter values for soft constraints; all constraints are handled uniformly. For every soft constraint violated, a relaxed problem is defined and solved. This approach is similar to modelling soft constraints as hard constraints and managing violations as proposed in [RPP02].

The main method of the solver is modified to operate on a queue of problems instead on a single one. When the solver starts, the problem which was passed to it is the only problem in the queue. During the solution process, an additional variable maintains the current cost of constraint violations. If a constraint is violated or the domain of a variable is wiped out during constraint revision, the cost of violation is checked. If the current cost of violation plus the cost of violating the constraint which caused the wipe out or is not satisfied is at most as big as the maximum allowed cost of constraint violation, a relaxed problem is defined without the offending constraint and added to the queue of problems to solve with the updated current cost of violation. The solver then continues to solve the current problem with updated cost of violation and the offending constraint removed. Only the values pruned immediately before the wipe out occurred are restored. This means that the solver will not find all solutions to the relaxed problem, as the values which have been pruned because of the removed constraint before the current step are not restored. The remaining solutions will be found when the solver processes the relaxed problem.

This approach will find certain solutions to relaxed problems twice and represents a compromise between time to find a solution and completeness. When a soft constraint is dropped, the solver continues to search for a solution without time-consuming restoration of all values pruned because of the removed constraint, but hence does not find all solutions. After the exploration of the search tree is complete, the solver is restarted with the relaxed problem to find the remaining solutions. Duplicate solutions are avoided by checking whether a solution exists already before adding it to the list of solutions.

Constructing relaxed problems eliminates the need to prune solutions to the problem which are not good enough, i.e. violate too many or too expensive constraints – such solutions will not be computed at all.

This method is less efficient than employing specialised algorithms for soft constraints, but is easier to implement and enables a uniform handling of hard and soft constraints. The solutions with least constraint violations will be computed first, therefore, if only the first solution to the problem is required, there is potentially very little overhead. For the specific class of problems considered in this thesis, only one solution is required and the best solution is the most relevant.

The high-level actions of the modified solver are illustrated in figure 16.

The addition of soft constraints is unobtrusive to both users of the constraint solver library and developers. The arguments specifying the cost of constraint violations are optional, the default values specify hard constraint problems. New algorithms for constraint revision can exploit the specific properties of soft constraints, but they are not required to. Arc consistency algorithms for hard constraints can be applied without modification.

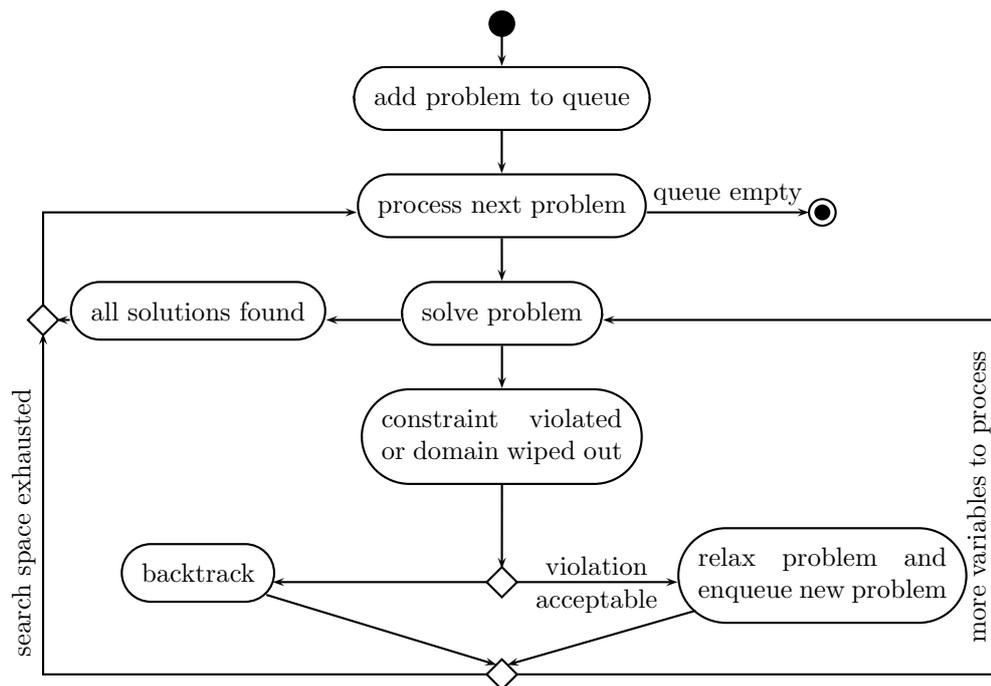


Figure 16. High-Level Actions performed while solving a Soft Constraint Problem

5.3.3 CONSTRAINT SATISFACTION WITH TIME LIMIT

The constraint solver implements the approach to real-time constraint satisfaction described in section 2.6. As the time available to solve the problem runs out, constraints are dropped according to their cost of violation, cheapest constraint first. When the time is up, all constraints are dropped. The function which determines when to drop which constraint is the logarithmic function illustrated in section 2.6.1.

When the solver is invoked, the current wall clock time is recorded. Then a hash table which maps times to lists of constraints to drop at that particular time is computed with the logarithmic function. At any given time, the hash can be used to determine which constraints are still relevant by partitioning the keys into times before and after the current time.

While solving the problem, the time is checked at the beginning of each assignment step. For problems with a very small time limit, the granularity of the checks might be too coarse to meet the limit, but the overhead of checking the time after e.g. every constraint revision would increase the run time significantly. The list of constraints which can be dropped at the current time is assembled using the auxiliary hash computed at the beginning. The constraints in the list which have not been dropped already are discarded and added to a list. If the time to solve the problem has elapsed and at least one solution has been found already, the solver simply terminates and returns the solutions found so far. Otherwise, all constraints are dropped and the unassigned variables are assigned the first value from their respective domains.

When a solution has been found, the total cost of constraint violations is determined by iterating over the list of constraints which were dropped because time was running out and checking whether each one holds. This is the only way to accurately account for the cost of violating constraints dropped because of limited time.

When a constraint is dropped, it will not necessarily be violated. All the domain values which would cause a violation might have been pruned before. Therefore a solution to the problem needs to be checked whether it violates any of the dropped constraints.

The high-level actions which result from the addition of constraint solving with time limit are illustrated in figure 17.

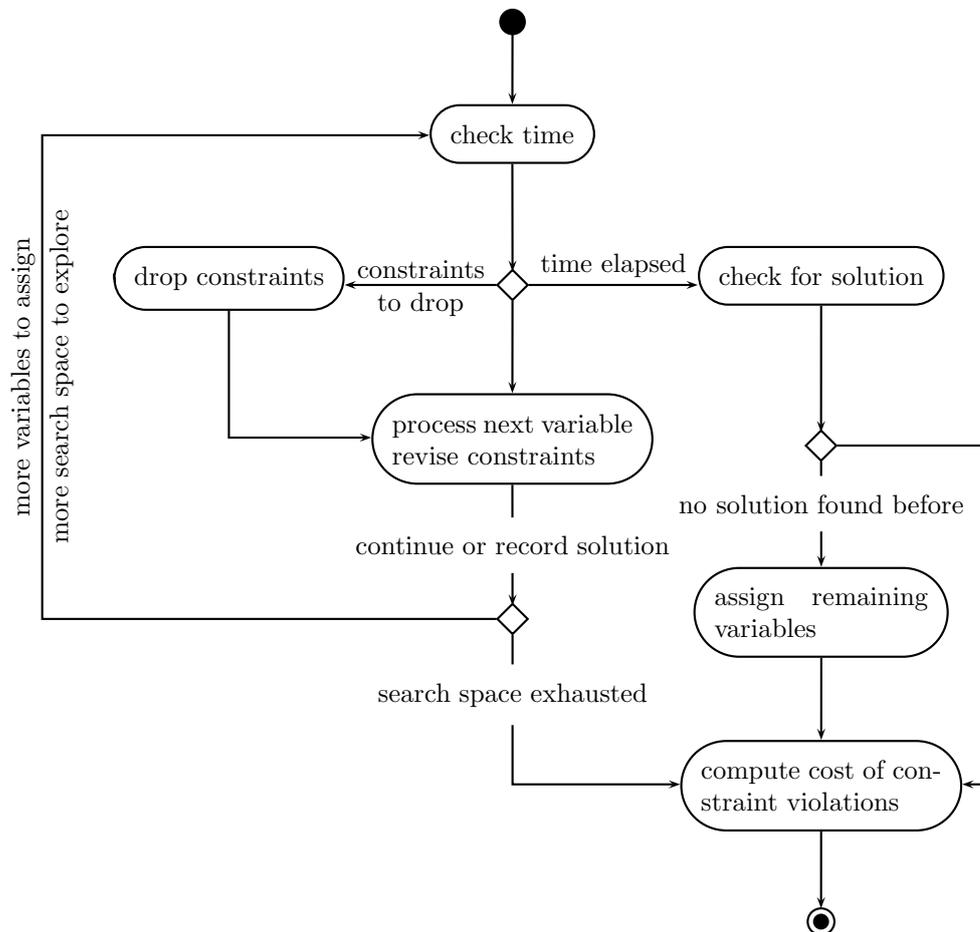


Figure 17. Solving of a Constraint Problem with Time Limit

Like all the other extensions of the solver, the addition of real-time constraint solving has been implemented as transparently as possible. The specification of a time limit is optional and if it is not specified, the default behaviour does not impose any restrictions on the time to solve. The algorithms for constraint revision are not affected in any way by time limits.

The effectiveness of the implementation with regards to returning a solution within the time limit is evaluated in appendix B.

5.3.4 VARIABLE AND VALUE ORDERING

The solution process starts with the variables sorted by their merit, i.e. the variable with the highest merit will be processed first. This ensures that the total merit of a solution is maximised, as the first variable has the broadest choice of values. The values are ordered by their merit, too, such that the value with the highest merit is assigned first to a variable. If no mapping of value to merit is provided, the values are assigned in the order they were specified in.

The variable and value ordering facilitates that the first solution found is the solution with the highest merit. It ensures that the variable with the highest merit is assigned the value with the highest possible merit, as restricted by the set of constraints. This mechanism reduces the effort needed to find the best solution to a constrained optimisation problem to the effort needed to solve the problem at all, i.e. solve a constraint satisfaction problem. Instead of having to find all solutions to determine the one with the highest merit, the solver is guaranteed to return the solution with the highest merit first. The additional time required compared to a constraint satisfaction problem is the up-front sorting of variables and values.

The addition of the ordering mechanism greatly improves the efficiency of the solver for constrained optimisation problems. In contrast to the usual approach taken with constraint satisfaction problems – processing the variable with the smallest domain first [CN88] – the objective is not to reduce the size of the search tree, but to determine the solution with the highest merit first. Decreasing the width of the search tree by ordering variables by the size of their domains might speed up the solution process, but having to find all solutions instead of one is likely to be slower in general.

5.4 CONSTRAINT REVISION

This section explains the particulars of the implementation of constraint revision. The concept is related to constraint networks [Mac75], but the nodes are implicitly specified. Each constraint represents an arc or edge in the network, binary constraints two-dimensional arcs, all different, tuple, and one-of-equals constraints n -dimensional hyper arcs.

Standard algorithms to enforce and maintain arc consistency are employed to ensure the consistency of the domains of all variables.

The object-oriented architecture of the solver separates traditional arc consistency algorithms into two parts. One part maintains the list of arcs to revise and is implemented in the solver, the other part revises individual arcs and is encapsulated in the `revise` method of the constraint classes.

This adaption enables the part which maintains the list of arcs to revise to handle constraints uniformly, regardless of whether they represents a two-dimensional arc, or an n -dimensional hyper arc.

5.4.1 BINARY CONSTRAINTS

The algorithm for binary constraint revision differentiates between three cases. If both variables have values assigned to them, no revision is necessary and the algorithm terminates immediately. If one of the variables has a value assigned to it, every value in the domain of the unassigned variable is checked whether it is supported by the assignment. The values which are not supported can be pruned. If none of the variables has a value assigned to it, support is checked for every pair of

values. The values in the domain of one variable which are not supported by any of the values in the domain of the other variable can be pruned.

The revision of binary constraints uses the AC 3 algorithm [Mac75] to find support for a value. AC 3 is a coarse grained algorithm with good time and space complexity. It was chosen because it is easier to implement than fine grained algorithms and does not require the domain elements to be sorted as the more sophisticated AC 3.x algorithms (cf. section 2.3.1 and appendix A.1).

A list of the variables with domains which were pruned is returned along with the total number of times the constraint was checked to hold during revision.

The implemented algorithm is compared to other approaches to binary constraint consistency in appendix A.1.

5.4.2 ALL DIFFERENT CONSTRAINTS

During the revision of all different constraints, local consistency of the binary decomposition of the constraint is enforced [GSW99]. The original algorithm described in [vHo1] has been slightly modified. Additionally, a satisfiability check is performed after constraint revision. This check does not increase the run time complexity of the algorithm, but enables to determine inconsistency even if no variable is assigned and all domains contain more than one element.

Constraint revision of all different constraints assembles a list of all assignments to variables involved in the constraint and prunes these values from the domains of all unassigned variables involved in the constraint. Values which have been assigned to a variable can not be assigned to any other variable involved in the constraint because that would violate the requirement that all assignments have to be different. Therefore, all assigned values can be pruned from the domains of all unassigned variables. Similarly, all values contained in domains of size one of unassigned variables can be pruned from the domains of the other unassigned variables.

The domains of assigned variables could potentially be pruned as well, but since the purpose of pruning is to reduce the search space and no search is performed with assigned variables, this would be wasted effort.

An additional check ensures that only the domains of variables which contain at least one of the values to prune are processed and added to the list of revised variables.

After pruning is complete, the satisfiability of the constraint is checked. If there exists an assignment to all variables such that the constraint holds, the cardinality of the union of the domains of all unassigned variables must be at least as big as the number of unassigned variables. In other words, for each unassigned variable there must be at least one distinct value. If there are fewer values than unassigned variables, a domain wipe out is signalled to the calling function. The algorithm uses Hall's Matching Theorem [Hal35].

The revise method requires $O(n)$ time to partition the list of involved variables in assigned and unassigned variables, where n is the number of variables involved in the constraint.

It takes $O(n)$ time to assemble the list of assigned values. This is achieved by collecting all assigned values and merging them with an initially empty set. Ruby implements sets with hash tables; therefore merging one value into the set is $O(1)$. At most n values are added to the set, so the run time is $O(n)$.

Filtering the unassigned variables which would not have any values pruned from their domains requires $O(n^2)$ time. The number of unassigned variables is bound

by $O(n)$, so iterating over them is, too. During each iteration, it is checked whether any of the values to prune is contained in the domain of the variable. The number of values to prune is at most n and the method to check for inclusion of any value iterates over all values passed to it in the worst case. Therefore, for each one of a maximum of n variables, a maximum of n values is tested for inclusion in the respective domain.

The pruning process requires $O(n^2)$ time. It iterates over the list of unassigned variables with domains to be pruned; the size of this list is at most n . For each variable, each value from the list of values to prune is removed from its domain. A single removal requires $O(1)$ time as domains are implemented as sets. The list of values to prune contains at most n items.

After the initial pruning, a list of variables with domains of size one is gathered in $O(n)$. For each of those variables, the value from their domain is removed from the domains of all other unassigned variables. If the size of the domain of a variable is one after pruning, it is added to the list of variables to process. This part requires $O(n^2)$ run time – iterating over the list of unassigned variables is $O(n)$; this is done at most n times, once for every variable with domain size one.

The check for satisfiability assembles the union of the domains of all unassigned variables in $O(n \cdot m)$ time, where m is the cardinality of the largest domain of all unassigned variables. This is done by iterating over the list of unassigned variables and for each one merging the values from the domain with an initially empty set in $O(m)$.

Therefore, the time complexity of the revise method is $O(\max(n^2, n \cdot m))$. The space complexity is $O(n + m)$ because only a list of values to prune, limited by the cardinality of the largest domain, and lists of assigned and unassigned variables and variables with domain size one, all limited by the number of variables, are maintained.

The implemented algorithm is compared to other approaches to consistency of the all different constraint in appendix [A.2](#).

5.4.3 TUPLE CONSTRAINTS

The revision of the tuple constraint differentiates between two cases – the list of allowed or disallowed tuples is known.

First the list of variables involved in the constraint is partitioned into assigned and unassigned variables. If the list of allowed tuples has been specified, the tuples which contain the current partial assignment are assembled into a list. If the disallowed tuples have been given, all the tuples which contain any value from the current assignment are gathered. More formally, the procedure for allowed tuples computes the intersection of the sets of tuples which contain the assignment to a variable for all assigned variables while the procedure for disallowed tuples computes the union of the same sets.

For every unassigned variable, a list of values which can be pruned from its domain is gathered. The assignments to the variable from the list of tuples assembled in the previous step are collected. If disallowed tuples have been specified, this list is the list of values to prune from the domain. If allowed tuples have been given, the list constitutes the allowed assignments. The values which can be pruned from the domain are all values which are not in this list. They are computed with the set difference between the domain and the list of assignments from the list of tuples.

If there are values to prune from the domain of an unassigned variable, it is added to the list of revised variables. If a domain wipe out occurs during the pruning

of a variable, constraint revision stops and the wipe out is signalled to the calling method.

The revise method requires $O(n)$ time to partition the list of involved variables in assigned and unassigned variables, where n is the number of variables involved in the constraint.

The list of tuples which contain the current assignment is assembled in $O(n \cdot t)$, where t is the total number of tuples. For each assigned variable, the list of tuples is traversed and those which contain the current assignment are collected. The size of this list is bound by the total number of tuples t .

The prune step iterates over the list of unassigned variables. The size of this list is bound by n . During each iteration, the assignments from the list of tuples assembled in the previous step are gathered. The size of the list of tuples to consider is bound by t . The actual pruning of the values is done in constant time.

Therefore, the time complexity of the revise method for tuple constraints is $O(n \cdot t)$. The space complexity is $O(n + (t \cdot m))$, where m is the maximum domain size. The list of variables is partitioned, the new lists require $O(n)$ space. The list of values to prune is bound by the number of tuples times the maximum domain size, as each tuple might forbid all values from a domain.

The maximum number of tuples is $n \cdot m$. Substituting this expression for t , the time complexity can be rewritten as $O(n^2 \cdot m)$ and the space complexity as $O(n + n \cdot m^2)$.

The revision of tuple constraints is completely independent from the domain size, although its complexity can be expressed in terms of the domain size. This is a notable difference to other implementations of constraint solvers which do not differentiate between intensional and extensional representation of constraints (cf. definition 2). The separate treatment yields significant performance improvements especially for problems with large domains and few allowed or disallowed tuples. Allowed and disallowed tuples are complementary for finite domains, thus the representation which specifies fewer tuples can be chosen for every particular problem.

A similar procedure for handling extensionally specified constraints regardless of their arity has been proposed in [BR97].

5.4.4 ONE-OF-EQUALS CONSTRAINT

The revision of the one-of-equals constraint only needs to perform work if exactly one of the variables involved in it has no value assigned to it and all the other variables have values different from the value requested assigned to them. If an assigned variable has the value requested assigned to it, no values can be pruned as the one-of-equals constraint does not impose any further restrictions on the assignments. If the value requested has been assigned to no variable and there is more than one variable unassigned, either of those variables can be assigned the requested value and therefore no other values can be pruned from their domains.

The revision of the constraint is a straightforward process. If the condition explained above applies, the unassigned variable is retrieved from the list of variables. The list of values which can be pruned from its domain are all the values in its domain except the requested value. If the domain of the unassigned variable contains only the requested value, the list of values to prune is empty.

If there are values to prune from the domain of the unassigned variable, it is added to the list of revised variables. If a domain wipe out occurs during the pruning, constraint revision stops and the wipe out is signalled to the calling method.

The time complexity of the `revise` method is $O(n)$, where n is the number of variables involved in the constraint. The traversal of the list of variables to look for unassigned variables and assigned variables with the requested value is linear in time. All other operations only involve the modifications of sets and can be done in constant time.

The method requires constant space, as there are no auxiliary data structures needed to revise the constraint.

5.5 TESTS AND PACKAGE MANAGEMENT

There are tests for all components of the system. The constructors of all the classes are tested with valid and invalid parameters, the most important methods are invoked with valid and invalid arguments and their return values checked. Different use cases are simulated and verified. The tests amount to a comprehensive specification of desired and undesired behaviour in various situations.

The tested functionality ranges from basic methods, such as the addition of a value to a domain, to high-level concepts, such as the solution of a constraint problem. The correctness of the implementation of algorithms is verified and the proper handling of the data entities ensured.

All tests are grouped together in a test suite. When invoked, it scans the current directory for all files that follow the naming conventions for test files and runs them. The advantage over running the tests individually is that the output of test runs is combined.

A Rakefile provides tasks to run the tests, create the documentation, and assemble the source files into a Ruby [gem](#).

5.6 LIMITATIONS

Although the library is meant to be as unspecific as possible to particular problems, there are limitations due to architecture and design decisions.

The objective function which assesses the “goodness” of a solution can not be specified when a problem is created. The merit is always determined by the sum of the merits of the individual variable - assignment pairs (cf. section 5.2.11). This follows definition 36.

Domains have been implemented as sets. Most recently developed algorithms to enforce and maintain arc consistency require the domains of variables to be ordered [BR01] [YY01] [LBH03]. To use these algorithms, the implementation of domains needs to be changed to ordered sets or arrays.

The more sophisticated algorithms for revision of the all different constraint require the domains of the variables to be intervals of numbers [Pug98] [MToo] [LOQTVBo3] [Lec96]. While the current implementation allows intervals to be domain elements, changes would be required to be able to employ these algorithms properly.

The implementation of soft constraints does not exploit the specific characteristics of soft constraints to filter domain values but runs the solver on a relaxed version of the problem with constraints removed.

Real-time constraint problems with a very small time frame might not terminate in time because of the granularity of the time checks. For soft constraint problems which require a lot of constraints to be disregarded before any solution at all can be found, the solver might drop a large number of additional constraints at the beginning when processing a relaxed problem because only little time remains to solve the problem.

The implementation of real-time constraint satisfaction was not thoroughly tested as the results depend on the particular machine used, the execution environment and a lot of external influences. The time limits given for finding a solution should be carefully examined for every problem.

6

CONSTRAINT PROBLEM SOLVER SOAP WRAPPER

This chapter describes the [SOAP Web Service](#) which provides an interface to the constraint problem solver library. It discusses the architecture of the wrapper and the limitations of the implementation.

6.1 ARCHITECTURE

The SOAP server uses the SOAP4R¹ library. As the latest version of the library has stability issues, the version which comes bundled with Ruby is used instead.

SOAP4R implements the SOAP 1.1 standard [\[SOA\]](#).

The wrapper package consists of

- a library script which implements a standalone SOAP server,
- the [Web Services Description Language \(WSDL\)](#) file which describes the interface to the server, and
- two control scripts which allow the server to be run in the foreground and as a [daemon](#).

The WSDL was not integrated with the SOAP server, but used as a standalone file. In a production environment, the web service should be integrated with existing web services infrastructure and the WSDL served the same way the WSDLs for the existing services are provided.

The structure of the SOAP server is illustrated in [figure 18](#).

6.1.1 LIBRARY SCRIPT

The library script uses the constraint problem solver [gem](#) to solve constraint satisfaction and constrained optimisation problems. The main class inherits the methods to set up a server and handle requests from the `StandaloneServer` class of the SOAP4R library. On initialisation of the server, a new constraint problem solver object is created and one method, `solve`, is added to the server. This method requires the following arguments,

- a list of domains,
- a list of variables,
- a list of constraints,
- a mapping from domain values to merit for a solution,

¹ <http://dev.ctor.org/soap4r>

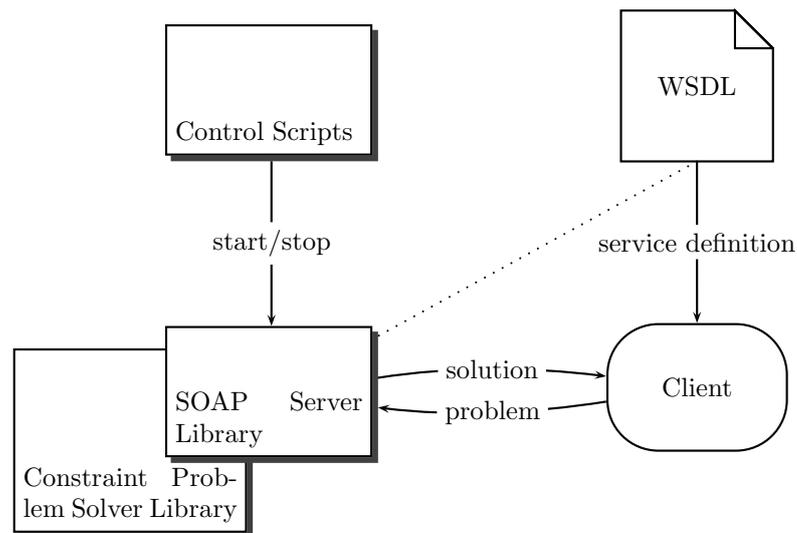


Figure 18. Architecture of SOAP Server

- a boolean indicating whether all solutions or only the first solution to the specified problem should be returned,
- a number specifying the maximum allowed cost of constraint violations,
- the maximum time allowed for finding a solution in seconds, and
- a boolean specifying if debug mode is on.

The implementation of the `solve` method converts the simple data structures received from the SOAP client into the classes which the constraint problem solver requires as input. After a new problem instance has been created, it is processed by the solver library. The result returned by the library is converted into simple data structures which can be passed back to the client via SOAP.

SOAP4R uses the `Log4r2` library to log the output of the solver. The log is set up automatically when the server is started. By default, the log messages are discarded. If debug mode is on, the messages are gathered into an array and included in the response. `Log4r` does not support logging messages to arrays; therefore the class `Array` has been extended with the `write` method which `Log4r` uses to log messages. The modified `Array` class is substituted for the default `Log4r` logger.

Only the lists of domains, variables, and constraints are explicitly checked for errors. If either of these lists is empty, a SOAP error will be sent back to the client. Errors in the problem definition will result in wrong solutions to the problem and should be detected by the client. Structural errors in the request will be detected by the SOAP library.

The SOAP layer is implemented entirely in SOAP4R. No specification of transport mechanism, parsing, or serialisation is required in the server. Therefore, no explicit dependencies on a particular SOAP library exist. Even the exchange of SOAP for a different protocol would be easily possible, as the main part of the server is concerned with the conversion of simple data structures which can be serialised by any web service protocol into more complex data structures.

² <http://log4r.sourceforge.net/>

6.1.2 WSDL

The WSDL which describes the SOAP interface to the server is an [XML Schema](#) document. It defines the data structures and types required in requests and given in responses. Additionally, the interface methods and the server parameters are specified.

For each of the lists passed to the service, a special type `ArrayOf...` is derived from the SOAP Array type. This ensures that only lists containing the right data type are passed to the service. The derived types do not allow to specify the minimum number of elements in the array or non-emptiness, therefore, explicit checking for the required number of elements is done in the service code.

The abstract constraint type with its four instantiations binary constraint, all different constraint, tuple constraint, and one-of-equals constraint is mirrored in the definition of the web service. A constraint must have exactly one of binary constraint, all different constraint, tuple constraint, or one-of-equals constraint as attribute. The cost of violation of the constraint is an attribute of the abstract constraint type and therefore common to all types of actual constraints. The encapsulation of actual constraints into elements of an abstract constraint type adds overhead to the description, but also the flexibility to handle the abstract constraint type at higher levels without knowing about the particulars of an actual constraint.

To map domain value to merit for a solution, a map data type is defined. It is an XML wrapper around the elements of a map with key and value attributes. A similar data structure is used to represent solutions as mappings from variable name to variable value.

Every element in the defined data structures is quantified, i.e. the minimum and maximum number of times it is expected to occur are specified. This rigid definition of the service interface enables both client and server to do extensive structural verification on request and response before processing it. The thorough error checking reduces the likelihood of wrong solutions to constraint problems because of syntactical errors in the problem definition.

The SOAP4R library does not enforce the quantifications though.

The SOAP server configuration parameters have to be duplicated in the WSDL and server scripts because SOAP4R provides no means of retrieving this information from a WSDL document. The interface of the service is specified in the WSDL only for both server and client.

The WSDL specification [[WSD](#)] does not allow messages to have optional parts. Therefore, the mapping from domain value to merit for a solution has to be specified. It may be empty though. Likewise, the booleans indicating the required number of solutions and if debug mode is on are required. Setting them to `false` produces the default behaviour, i.e. all solutions to the given problem are returned and no debug output is provided.

6.1.3 CONTROL SCRIPTS

The control scripts provide an interface to the library script which implements the server. They can be called directly from the command line when the server package has been installed and will start the SOAP server with the parameters specified in the WSDL.

The first script, `ConstraintSolverServer`, creates a new instance of the standalone SOAP server defined in the library script and starts it. The process runs in the foreground and can be controlled directly by the user who started it.

The second script, `ConstraintSolverServer-daemons`, uses the `daemons`³ library to run the first control script in the background. The `daemons` library offers convenient means to start and stop the server and takes care of logging. The system-wide temporary directory is used to store the log and accounting information for the daemonised process.

6.2 TESTS AND PACKAGE MANAGEMENT

The SOAP server comes with a set of functional tests which create a new server, set up a client to send a problem definition, and verify that the returned solution is correct. The functional tests are similar to those of the constraint solver library, but not as extensive. The main goal of the tests is to verify the interface to the solver, not that the solver itself is functioning correctly.

A Rakefile at the top level provides tasks to run the tests, create the distribution package and generate the documentation for the code.

6.3 LIMITATIONS

A few limitations of the SOAP4R library became apparent while implementing the server.

The errors caused by faulty WSDLs often give cryptic error messages and do not provide sufficient information to determine the cause of the problem. A lot of the current WSDL was verified by trial and error. Some parts of the WSDL, such as the quantifiers for the elements, are ignored by SOAP4R.

Nested sequences and choices of elements are not allowed, therefore the common element of constraints, the cost of violation, is included in the choice between binary, all different, tuple, and one-of-equals constraint. SOAP4R allows more than one of the elements specified in a choice to appear.

The WSDL specification does not support optional message parts, therefore all the parameters have to be given when calling the `solve` method.

6.4 USE OF THE INTERFACE

The SOAP service is intended to be used by software engineers who do not necessarily have a background in constraint programming or Artificial Intelligence. The interface deliberately only provides means of defining problems and solving them and not a lot of parameters to tweak, such as specific algorithms employed to solve the problem or similar in-depth configuration of the solver.

The user interface follows the “model and run” paradigm proposed in [Pugo4]. It elaborates on the approach by not only offering a standard format for in- and output, but by integrating it with technologies commonly not found in Artificial Intelligence programming. Web services are ubiquitous in software engineering however, so a software engineer will have little trouble using the SOAP interface and integrating the service with existing applications.

The development of contemporary user interfaces to constraint programming systems and simplifying their use for non-specialist users will enable the application of Artificial Intelligence techniques to new problems and facilitate research and development of new methods for solving hard problems.

³ <http://daemons.rubyforge.org/>

7

WEB USER INTERFACE

This chapter introduces the web user interface to the constraint problem solver. It will examine this part of the system in detail and discuss the implementation, design decision and features.

7.1 ARCHITECTURE

The user interface uses the Camping web framework¹. Camping is written in Ruby and was inspired by Ruby on Rails². Unlike Rails, there is no need to set up a specific directory structure and skeletons for the different parts of the application. The whole application can be implemented in a single file and enables developers to rapidly prototype web applications. This is why Camping was chosen over Ruby on Rails.

Camping is a very small and lightweight framework which can be integrated with a number of popular web servers. It is written entirely in Ruby and makes extensive use of the features of the language.

Camping uses the [Model-View-Controller \(MVC\)](#) design pattern [Bur92]. The application is separated into the data model, the controller and the view. There are specialised classes for each part – the model manages the domain specific data and the state of the application, the controller responds to user interactions by sending commands to model and view, and the view manages the [HTML](#) output and presentation.

The advantage of this approach is that the individual components are only loosely coupled and can easily be exchanged with components which provide a similar interface – e.g. a model which uses a different database or a view which renders its output with a window toolkit. There is a strict separation between the functional modules of a system, making them easier to understand and maintain.

7.1.1 DATA MODEL

The Camping application uses a [SQLite](#)³ database to store the data used to define constraint satisfaction problems to solve. SQLite is a [Structured Query Language \(SQL\)](#) database engine which uses in-memory databases and requires no database server. It was chosen because there are no external dependencies and requirements to be set up before the application can be run. The expected volume of the data for this application is very small, as only campaign and slot data for demonstration purposes is stored. Therefore, the resources the database needs and query performance will not be issues to consider.

¹ <http://code.whytheluckystiff.net/camping/>

² <http://www.rubyonrails.org/>

³ <http://www.sqlite.org/>

The data model is illustrated in figure 19.

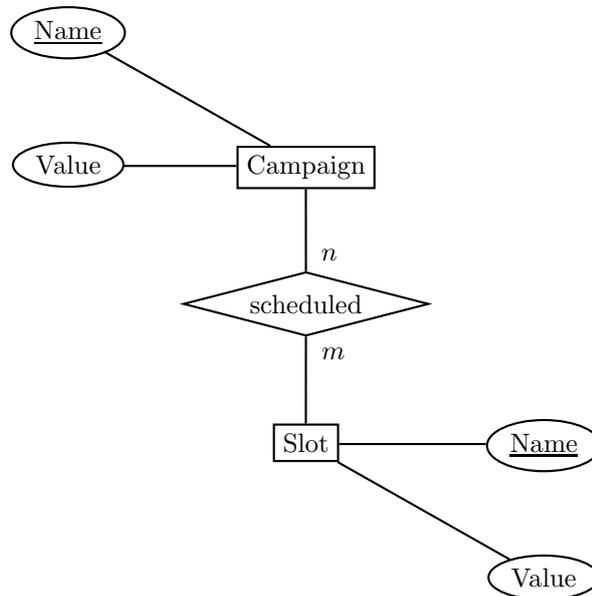


Figure 19. Entity-Relationship Diagram [Che76] of the Data Model of the Web User Interface

Camping provides facilities for setting up a database when the application is run for the first time, as well as migrating earlier versions of the data model to the current one. This feature is used to provide means of creating the database schema and populating the database with example values. When the application is run for the first time, this is executed. Most of the installation and setup is done automatically and very little user intervention is required.

7.1.2 CONTROLLER

The controller retrieves the campaigns and the slots they are scheduled in from the database and passes them on to the view component to generate a form if no problem definition has been posted to the application. Camping provides transparent access to the database via ActiveRecord⁴. For each entity in the database, a class which inherits from the ActiveRecord base class is defined in the model. Data can be accessed through static methods of these classes. The retrieved data is saved in variables such that the view component can access it.

After a problem description has been posted, the controller

- retrieves data from Amazon.com for the selected campaigns to determine which products are available to be promoted,
- collects the input data and assembles it into the data structures needed for the SOAP call,
- creates a connection to the SOAP server and sends the problem description, and
- receives and checks the solutions to the problem.

⁴ <http://ar.rubyonrails.com/>

The input data is separated into variables, the slots, and their domains, the campaigns scheduled in the slot. The domain for each slot variable consists of a name and an array of values. Each slot variable consists of a name, a reference to a domain, and a number designating the merit for a solution. The merit is stored in the database and can not be modified by the user. The slot variables are assembled into an all different constraint, i.e. no campaign must be shown twice. The cost of violating this constraint is specified by the user; the default value is 0.2.

If the user specified one or more campaigns to be shown at least once on the input page, one one-of-equals constraint for each of those campaigns is added. This constraint requires at least one of the slots to display the campaign. The cost of violating the constraint is set to be ten times the maximum cost of constraint violations allowed, i.e. the constraint is modelled as a hard constraint.

The uniqueness of products promoted is requested by introducing additional variables, domains, and constraints. Each slot has three places for products, therefore three new variables are added for each slot. The domain of the variables is the list of products which may be promoted by all campaigns. This list has been retrieved before by the call to Amazon.com. For each place variable, a binary tuple constraint with the slot variable the place is contained in is added. This constraint restricts the domain of the place variable to the products which may be promoted through the campaign assigned to the slot variable. All these constraints are modelled as hard constraints by setting the cost of violation to ten times the maximum cost of constraint violations specified by the user. Finally, a constraint which requires all place variables to be different is added. The cost of violating this constraint is specified by the user; the default value is 1.0.

After that a mapping from campaign to merit of the campaign for a solution is created. The merit is stored in the database and can not be modified by the user. The boolean specifying that only the first solution should be computed is always set to `true`. Requesting only the first solution reduces the time the solver runs significantly and only one solution is shown anyway.

The maximum allowed cost of constraint violations is retrieved from the input form; the default value is 0.5. The time limit for finding a solution to the problem is specified by the user; the default value is 0.5 seconds. This limit only applies for the run of the solver itself, the SOAP call adds overhead to the operation. The problem description and solution have to be marshalled into and demarshalled from XML, the objects converted from and to simple data structures, and the request has to be transmitted between the frontend and the SOAP server.

The debug argument to the call is set to `true` if the user selected the debug check box, `false` otherwise.

The connection to the SOAP server is established by calling the `solve` method which is created automatically by SOAP4R⁵ upon reading the WSDL of the web service with the parameters defined in the WSDL. No further specifications of service or supported methods are necessary. The interface to the SOAP service is updated automatically as the WSDL is updated. The result of the service call is checked whether it contains a solution and the time the call took is recorded. This is not only the time the solver needed to find a solution, but also the overhead introduced by the SOAP messaging, i.e. XML marshalling and demarshalling. If there is no solution, the class member variable holding the solution is set to a null value. If there is a solution to the problem, it is stored in the member variable. The first solution is guaranteed to be the solution with the highest merit (cf. section 5.3.4). As the goal is to maximise the value of the page and only one page is displayed, only the first

5 <http://dev.ctor.org/soap4r>

solution is required.

The controller assembles an additional data structure which holds the mappings from slot to list of products to promote. This mapping is assembled by collecting the assignments of the auxiliary place variables and grouping them by slot. The auxiliary variables are removed from the solution before it is forwarded to the view component.

If the service returned the log of the run of the solver, it is saved in a member variable for display.

The mechanisms and libraries used to interface with Amazon's web service will be explained in detail in section 7.2.

7.1.3 VIEW

The view constitutes the presentation layer of the application and is used to display the form which allows the user to specify the problem and render the page which demonstrates the solution to a problem.

The HTML of the web page is generated with Markaby⁶. It is a [domain specific language \(DSL\)](#) for writing HTML code in Ruby. The elements of the DSL are HTML tags. They are assembled into a page by an HTML builder. Markaby acts as an abstraction layer between the view and the rendered HTML output. It takes care of most of the details; especially the creation of tables and other nested structures is simplified. HTML headers and footers are inserted automatically.

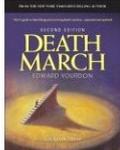
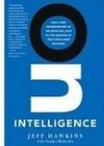
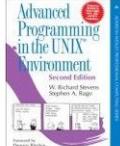
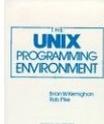
Please select the campaigns to be scheduled for display

designed by an engineer

center-1	<input type="checkbox"/> wishlist <input checked="" type="checkbox"/> recommendations <input type="checkbox"/> bestsellers
center-2	<input type="checkbox"/> wishlist <input checked="" type="checkbox"/> recommendations <input type="checkbox"/> bestsellers
center-3	<input type="checkbox"/> wishlist <input checked="" type="checkbox"/> recommendations <input type="checkbox"/> bestsellers
show at least once	<input type="checkbox"/> wishlist <input checked="" type="checkbox"/> recommendations <input type="checkbox"/> bestsellers
time limit	<input type="text" value="0.5"/> seconds
penalty for showing a campaign more than once	<input type="text" value="0.2"/>
penalty for showing a product more than once	<input type="text" value="1.0"/>
maximum penalties	<input type="text" value="0.5"/>
debug?	<input type="checkbox"/>
<input type="button" value="Render Website"/>	

Figure 20. Form to specify Problem to solve

⁶ <http://code.whytheluckystiff.net/markaby/>

center-1 wishlist	 Death March, Second Edition by Edward Yourdon	 Dance of the Gods by Tatsuya Ishida	 Joel on Software: And on Diverse and Occasionally Related Matters That Will Prove of Interest to Software Developers, Designers, and Managers, and to Those Who, Whether by Good Fortune or Ill Luck, Work with Them in Some Capacity by Joel Spolsky
center-2 wishlist	 On Intelligence by Jeff Hawkins, Sandra Blakeslee	 Sinfest by Tatsuya Ishida	 The Pragmatic Programmer: From Journeyman to Master by Andrew Hunt, David Thomas
center-3 recommendations	 Standard C Library, The by P. J. Plauger	 Advanced Programming in the UNIX(R) Environment (2nd Edition) (Addison-Wesley Professional Computing Series) by W. Richard Stevens, Stephen A. Rago	 The UNIX Programming Environment by Brian W. Kernighan, Rob Pike

Value of page: 16.8
Constraint Solver Service called in 4.829218s.

Figure 21. Result Page with rendered Solution for Input in Figure 20

7.2 INTERFACE TO AMAZON.COM

For each campaign in the database, the application calls Amazon E-Commerce Services (ECS)⁷ to retrieve item data. It uses the Ruby/Amazon⁸ library to make the service calls. The library provides a Request class which can be used to perform various searches for products – similarities, items from people’s wish lists, books with a specific title, or author. For the returned data, a Product class is implemented. This class provides access to the attributes of the item. Ruby/Amazon has many more features, only the ones used are described here.

Amazon E-Commerce Services limit the request rate to one per second. The library takes care of this and pauses automatically after each request. This limitation does not affect the development of the user interface, but the result page will be rendered more slowly.

The interface to Amazon.com has been added to the application to make the demonstration more realistic. Real products are advertised and the links go to real product pages on the Amazon.com website.

The following campaigns have been implemented,

WISHLIST A random selection of three products from a wish list is displayed.

RECOMMENDATIONS A random selection of three products which are similar to one in a set of seed items – which the user is interested in – is displayed.

BESTSELLERS A random selection of three best selling books is displayed.

⁷ <http://www.amazon.com/E-Commerce-Service-AWS-home-page/b?node=12738641>

⁸ <http://www.caliban.org/ruby/ruby-amazon.shtml>

The addition of further – especially personalised – campaigns would require access to more of Amazon’s data. The current implementation serves as a prototype to show what could be done and is not intended to be an accurate, exhaustive, or promotional successful representation of the real Amazon.com web site.

The activities performed by the application are summarised in figure 22. For simplicity reasons, it only considers the run of the application with debug mode off.

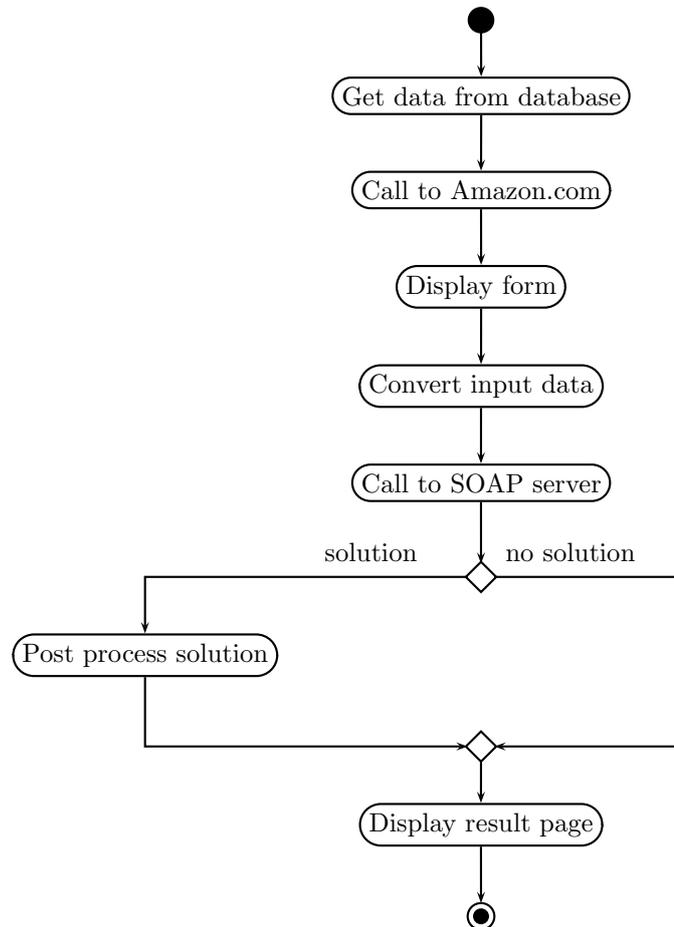


Figure 22. Activity Diagram for the Web User Interface

7.3 TESTS AND PACKAGE MANAGEMENT

The web interface has a test to make sure that the WSDL of the constraint solver service is what it expects. This test safeguards against unnoticed changes in the interface to the web service and signals the addition of new features.

There are no other tests for the web user interface, as it requires the constraint solver service for most of its functionality and can not be tested standalone.

A Rakefile at the top level provides tasks to run the tests, create the distribution package and generate the documentation for the code.

A [FastCGI \(FCGI\)](#) script, `dispatch.rb`, is included in the distribution for easy integration with existing web servers.

7.4 FULL EXAMPLE

This section gives a full example for the rendering of a page. The problem and solution are expressed in terms of the model described in chapter 3, the input data is illustrated in figure 20 and the result in figure 21. The products are designated by their Amazon identifier.

The page consists of three slots,

$$X_s = \langle \text{center} - 1, \text{center} - 2, \text{center} - 3 \rangle.$$

Each slot has three different places for product displays. Therefore, the variables are

$$X = \langle \text{center} - 1, \text{center} - 1_1, \text{center} - 1_2, \text{center} - 1_3, \\ \text{center} - 2, \text{center} - 2_1, \text{center} - 2_2, \text{center} - 2_3, \\ \text{center} - 3, \text{center} - 3_1, \text{center} - 3_2, \text{center} - 3_3 \rangle.$$

There are three campaigns, *wishlist*, *recommendations*, and *bestsellers*. Of the three campaigns, two have been selected for each slot,

$$d_c = \{\text{wishlist}, \text{recommendations}\}.$$

The sets of the products which each campaign may promote is

$$S_{\text{wishlist}} = \{0972466304, 1590593898, 020161622X, 0972466320, \\ 0805078533, 0972466312, 0735619670, 013143635X, \\ 0444875085, 1565927249, 1931836052, 0131103628\} \\ S_{\text{recommendations}} = \{0735605351, 0932633439, 1572316217, 0201485672, \\ 020161622X, 1556159005, 0201835959, 0201633612, \\ 0596007124, 0735621632, 0131315099, 0201433079, \\ 013937681X\}.$$

The set of all products which may be promoted is

$$d_s = S_{\text{wishlist}} \cup S_{\text{recommendations}}$$

The domains of the variables are

$$D = \langle d_c, d_c, d_c, d_s, d_s, d_s, d_s, d_s, d_s, d_s, d_s, d_s \rangle$$

The value of the slots is

$$\text{value}(\text{center} - 1) = 3$$

$$\text{value}(\text{center} - 2) = 2$$

$$\text{value}(\text{center} - 3) = 1.$$

and the value of the campaigns

$$\text{value}(\text{wishlist}) = 3$$

$$\text{value}(\text{recommendations}) = 2.$$

The constraints which link displayed products to assigned campaigns are specified by the allowed tuples;

$$C_{cp} = \{ \langle \text{center} - 1, \text{center} - 1_1, \{ \langle \text{wishlist}, 0972466304 \rangle, \langle \text{wishlist}, 1590593898 \rangle, \langle \text{wishlist}, 020161622X \rangle, \langle \text{wishlist}, 0972466320 \rangle, \langle \text{wishlist}, 0805078533 \rangle, \langle \text{wishlist}, 0972466312 \rangle, \langle \text{wishlist}, 0735619670 \rangle, \langle \text{wishlist}, 013143635X \rangle, \langle \text{wishlist}, 0444875085 \rangle, \langle \text{wishlist}, 1565927249 \rangle, \langle \text{wishlist}, 1931836052 \rangle, \langle \text{wishlist}, 0131103628 \rangle, \langle \text{recommendations}, 0735605351 \rangle, \langle \text{recommendations}, 0932633439 \rangle, \langle \text{recommendations}, 1572316217 \rangle, \langle \text{recommendations}, 0201485672 \rangle, \langle \text{recommendations}, 020161622X \rangle, \langle \text{recommendations}, 1556159005 \rangle, \langle \text{recommendations}, 0201835959 \rangle, \langle \text{recommendations}, 0201633612 \rangle, \langle \text{recommendations}, 0596007124 \rangle, \langle \text{recommendations}, 0735621632 \rangle, \langle \text{recommendations}, 0131315099 \rangle, \langle \text{recommendations}, 0201433079 \rangle, \langle \text{recommendations}, 013937681X \rangle \} \}, \dots \}.$$

The set of constraints is

$$C_{dcp} = \{ \text{AllDifferent}(\text{center} - 1, \text{center} - 2, \text{center} - 3), \text{AllDifferent}(\text{center} - 1_1, \text{center} - 1_2, \text{center} - 1_3, \text{center} - 2_1, \text{center} - 2_2, \text{center} - 2_3, \text{center} - 3_1, \text{center} - 3_2, \text{center} - 3_3) \cup C_{cp}.$$

The cost of violating the first constraint is $\varphi(\text{AllDifferent}(\text{center} - 1, \text{center} - 2, \text{center} - 3)) = 0.2$. The cost of violating the constraint which requires all products to be different is 1.0 and the maximum cost of constraint violations allowed is $v = 0.5$.

An additional constraint requires at least one slot to show the recommendations campaign,

$$C = C_{dcp} \cup \{ \text{OneOfEquals}(\{ \langle \text{center} - 1, \text{center} - 2, \text{center} - 2 \rangle, \text{recommendations} \} \}.$$

The time limit for finding a solution is 0.5 seconds.

The constraint which requires all campaigns to be different is dropped early in the solution process as there are only two campaigns available for three slots. All the other constraints can be satisfied. The one-of-equals constraint prevents the wishlist campaign from being shown in all slots; there are enough distinct products available for wishlist to make showing it everywhere a feasible solution.

A solution to the problem which involves constraint violations is

```

assignment(center - 1) = wishlist
assignment(center - 2) = wishlist
assignment(center - 3) = recommendations
assignment(center - 1_1) = 013143635X
assignment(center - 1_2) = 0972466320
assignment(center - 1_3) = 1590593898
assignment(center - 2_1) = 0805078533
assignment(center - 2_2) = 0972466304
assignment(center - 2_3) = 020161622X
assignment(center - 3_1) = 0131315099
assignment(center - 3_2) = 0201433079
assignment(center - 3_3) = 013937681X.

```

The value of the page is

$$v(p) = (3 \cdot 3) + (3 \cdot 2) + (2 \cdot 1) - 0.2 = 16.8.$$

The pages in real applications are much larger than the example presented here. Usually there will be more than 50 variables and dozens of constraints in the proposed framework. This shows that rendering a website is indeed a problem of considerable size and constraint programming techniques are adequate.

SUMMARY

8.1 FUTURE WORK

This section gives an overview of directions for future research and further exploration of the methods and concepts introduced in this work.

8.1.1 CONSTRAINT MODEL

The model which uses constraint programming to generate the web site of an online retailer as detailed in chapter 3 provides most of the features required in the description of the problem in chapter 1. Some of the more sophisticated aspects have not been modelled though.

The model does currently not take external events and availability of data about a customer into account. It is assumed that all data is known when the solving starts, i.e. the domains are static sets of values and not dynamically influenced by completed service calls or similar. If the solution to the constraint problem and the available data could be determined in parallel, less time would be required to find a solution.

Limits on available data could be modelled by starting with the largest possible domains and introducing more constraints into the problem as the available data becomes known. Problems with this approach are that the initial domain size would potentially be very large, requiring lots of space and computing time, and that in some cases no data can be provided before the completion of some service calls, making this approach infeasible.

Another extension would be to model the values of slots and campaigns more accurately. Every customer experiences promotions differently and the intrinsic value of every campaign is different for different customers. There are lots of trade-offs involved, for example a campaign might sell only few high-value items, e.g. plasma TVs, but still be more valuable than a campaign which sells something almost every time it is shown but only low-value items, e.g. discounted DVDs. It might be better to show the campaign which sells things more often even if it is less valuable.

There are different ways to incorporate issues like this into the model. The values of slots and campaigns could be modelled as dynamic values which are different for every customer according to the data available. The framework of soft constraints could be extended to dynamically adjust the costs of constraint violations or use fuzzy values instead of fixed ones.

The tuple constraint allows to model many more things specific to the retailer policy, e.g. not to promote DVDs on pages of the book store, not to promote more than one article of a certain product category, or to promote certain products together with matching accessories. The campaigns themselves could be modelled as constraints.

8.1.2 CONSTRAINT SOLVER

The constraint solver library described in chapter 5 introduced a number of new concepts which provide directions for future research.

The library is able to handle valued soft constraint problems, but does not employ algorithms specific to soft constraints. Instead, a new, relaxed problem is solved for each soft constraint violated. Specialised algorithms for enforcing and maintaining arc consistency would eliminate the need to solve several problems and significantly increase the performance. Current research has already developed a number of filtering algorithms for soft constraints [PRBo1] [vHo4].

The integration of soft constraints with real-time constraint satisfaction is an entirely new concept. There are many things left to research; the fundamental concept has only been explained and not proofed, the model only integrates with valued soft constraints, and the performance of the implementation leaves room for improvement (cf. appendix B).

An interesting challenge is to integrate the model of real-time constraint satisfaction with distributed constraint satisfaction. Parallelising the search for a solution should provide better estimates for how long it takes to find a solution, as several branches of the search tree can be explored at the same time. It should also enable the solver to determine a solution more accurately within the time limit.

Distributed constraint satisfaction does not only improve the performance of the solver, but also allows separated agents to collaborate. No aggregation of all the data required to model the problem is necessary, saving time and work [YDIK98]. Distributed constraint satisfaction is an active area of research itself [CDK91] [YHoo] [HY97] [BSo6] [Ham06].

The constraint solver library offers no facility to process textual descriptions of problems or a user interface. The implementation of these features would significantly improve the usability of the solver.

Another issue is the performance of the solution process. Some other solvers (cf. table 4) provide a significantly better performance. Only few algorithms to enforce and maintain arc consistency have been implemented. A more comprehensive library of algorithms would enable the user to choose the algorithm which is best for the specific problem.

8.2 CONCLUSION

The aim of this work was to model the problem of generating a web site as a constraint problem and provide a prototypical implementation to solve such problems.

The first chapters gave an in-depth description of the problem, the constraints involved, and the theoretical background. Constraint programming was introduced and definitions for the most important concepts, including constraint satisfaction problems, constrained optimisation problems, and soft constraint problems were given. The definition of constraint satisfaction problems given is different from the ones usually found and corrects an inaccuracy in other definitions. In addition to constrained optimisation problems, extended constrained optimisation problems were introduced and defined. They are a new concept to provide better means of modelling the problem which was subject to this thesis.

A comprehensive review of literature available on constraint programming facilitated a detailed summary of known algorithms, frameworks, and concepts. References to literature which explains techniques only mentioned more in-depth are found throughout this work.

For the first time, the concept of real-time constraint satisfaction with time limit was introduced. A general framework was provided and integrated with the existing framework of valued soft constraint satisfaction problems. This work provided definitions, suggested an instantiation of the framework, and evaluated its performance. The described algorithms and concepts are unique to this thesis and constitute a major contribution to the constraint programming research community.

The problem of rendering a web site using constraints was concisely presented, formal definitions of the involved entities and concepts were given and illustrated by several examples. An initial simple model was developed into a full-featured one by successively applying constraint programming techniques to a real-life problem.

The implementation of the constraint solver library uses no existing libraries but was done from scratch using an own approach to architecture and design. It is the first implementation of a constraint programming system in Ruby and provides a feature set which is not even found in most commercial solvers after considerably more development. The library has been documented comprehensively and the performance of the implemented algorithms evaluated extensively. It uses a specialised algorithm of low complexity for constraints which are extensionally specified.

The constraint problem solver library did not incorporate restrictions usually found in other solvers, such as that the domains of variables must contain intervals of enumerable values. The algorithms used do not rely on specific properties of domains, variables, or constraints.

The SOAP web service interface presents a new and unique approach to integrate constraint problem solving with contemporary communication technologies and software engineering techniques. The implementation of the interface as a service makes distribution and maintenance of the software easier.

The web frontend to the service illustrated the problem of rendering web sites with constraint programming techniques descriptively and demonstratively. The interface to a real online retailer shows the feasibility of the approach in industry and adds to the value of the frontend as a means of demonstrating the problem and its solution.

The implementation used standard software engineering techniques, there is detailed documentation both of high-level concepts and implementation details. There are extensive tests to ensure that the software is working as desired. The code is concise, special attention has been paid to a clean separation of the interfaces, functional separation, encapsulation and abstraction, and other principles of object oriented software design.

The aims of this work as outlined in the first chapter have been realised. Additionally, several new and unique concepts have been identified and described, flaws in existing definitions and techniques corrected, and comprehensive reviews and analyses of existing technologies presented.

Part III

APPENDIX



PERFORMANCE EVALUATION OF CONSISTENCY ALGORITHMS

A.1 BINARY CONSTRAINTS

The consistency of binary constraints is one of the best studied topics in constraint programming. A wide variety of algorithms to enforce and maintain consistency have been developed and investigated (cf. section 2.3.1). In addition to AC 3 (cf. section 5.4.1), a modified version of the AC 3.3 algorithm [LBH03] has been implemented to assess the performance.

For every value in the domain of a variable, a set of values from the domain of the other variable which are supported by this particular assignment and another set of values which are not supported by it are maintained. These data structures are high-level caches of constraint checks and do not need to be revised when the solver backtracks. There is no need to maintain elaborate data structures along with arc revision [BC94] – once the sets of supported and unsupported values have been initialised, they never need to be revised. The information gathered during one run of the arc consistency algorithm is reused during subsequent runs [LZBF04].

The maintained sets of supported and unsupported values enable caching of both positive and negative checks. Instead of caching the result of a constraint check as proposed in [LZBF04], the result is cached a layer above in the algorithm that enforces arc consistency. This eliminates the need for explicit caching of constraint checks. It furthermore enables the revision algorithm to process a constraint in constant time after the auxiliary data structures have been filled. The list of values to prune can be taken directly from the set of unsupported values for the assignment, no computation is necessary.

The original AC 3.3 algorithm had to be modified because the implementation of domains uses sets, i.e. there is no order imposed on the elements. The AC 3.x family of algorithms [BR01] [YY01] [BRYZ05] [LBH03] relies on an auxiliary structure which saves the last support for a certain value. This requires the elements of a domain to be ordered. The implementation could be changed to use ordered sets for domains, but that would restrict the domain elements to things which can be compared.

The modified algorithm maintains the time complexity of AC 3.3, $O(ed^2)$, where e is the number of edges in the constraint network – i.e. the number of constraints – and d the maximum domain size of all variables. The space complexity increases from $O(ed)$ to $O(ed^2)$, as for every value in every domain sets of supported and unsupported values are stored.

The number of constraint checks is greatly reduced compared to all other AC algorithms. Because of the auxiliary data structures, every constraint is at most checked exactly once for every possible assignment to the variables involved in it.

The upper bound of constraint checks is

$$\sum_{i=1}^n x_i, y_i \text{ involved in } c_i : \|\text{dom}(x_i)\| \cdot \|\text{dom}(y_i)\|,$$

i.e. the sum of the products of the cardinality of the domains of the two variables which are involved in the binary constraint for all constraints.

The following sections present the methodology and the results of the performance evaluation of the two algorithms.

A.1.1 METHODOLOGY

To compare the performance of the implemented algorithms, three types of problems were examined,

- the decomposition of the all different “pathological problem” [Pug98] into binary constraints with n variables, $n \cdot (n - 1)$ binary \neq constraints, and domains ranging from $\{1\}$ for the first variable to $\{1, \dots, n\}$ for the n th variable,
- a problem class with n variables with identical domains of size n and $n \cdot (n - 1)$ binary $=$ constraints requiring the value of each variable to be identical to the value of every other variable, and
- a problem class which requires the values of n variables with identical domains of size n to be ordered ascending, i.e. to fulfil a $<$ constraint between each pair of consecutive variables.

The time taken to find the first solution to a problem was measured. Additionally, the number of constraint checks was recorded.

The time required was averaged over 100 runs of the solver for a particular problem instance. Additionally, the first ten runs were discarded to allow for caching effects to settle. The maximum and minimum values measured were considered to create confidence intervals. To minimise disturbance by exterior influences, the CPU time used was measured instead of the wall clock time.

The experiments were conducted on a 1.4 GHz Intel Pentium M Processor machine with 1024 MB RAM. The installation of Ruby was custom-compiled with optimised compiler settings.

A.1.2 RESULTS

The measured times and constraint checks taken to solve the problems are depicted in diagrams 23 to 25.

Figure 23 shows that although the modified AC 3.3 algorithm reduces the number of constraint checks, the run time performance is approximately equal to that of the original AC 3 algorithm. The constraint check cache needs to be maintained and requires its part of the performance.

Surprisingly, figures 24 and 25 show a similar picture although the number of constraint checks performed by the modified AC 3.3 algorithm is significantly less than of the other algorithm. The run time performance is actually slightly worse.

In this implementation, the cost of a constraint check is very low, about the same as a lookup in the cache. Therefore using a cache does not yield any performance benefits, but increases the run time due to the overhead of maintaining it. Similar results were observed in [Dono4].

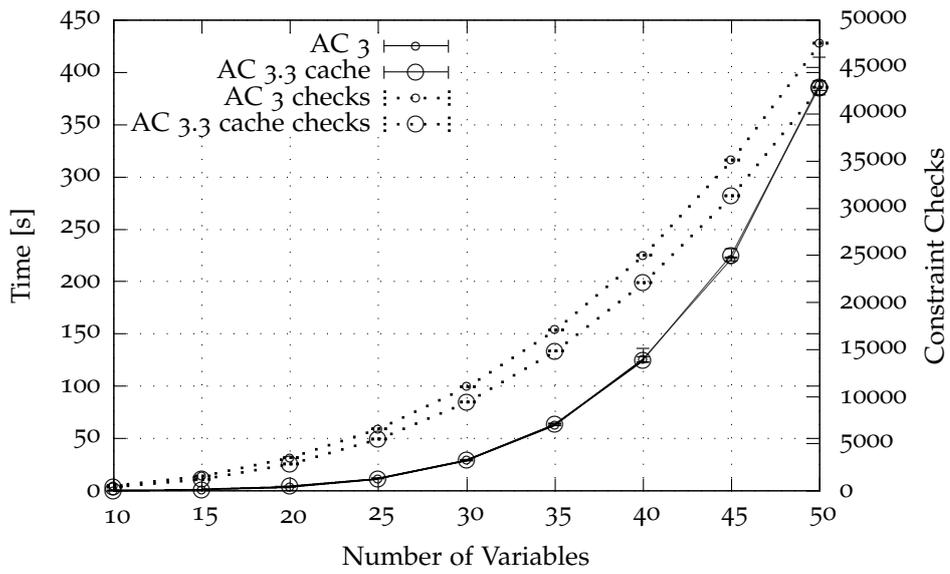


Figure 23. Binary Constraint Performance for Solution of “pathological” Problems

The AC 3 algorithm was chosen for the implementation of the solver because the modified AC 3.3 algorithm is more complex but does not improve the performance for this particular implementation.

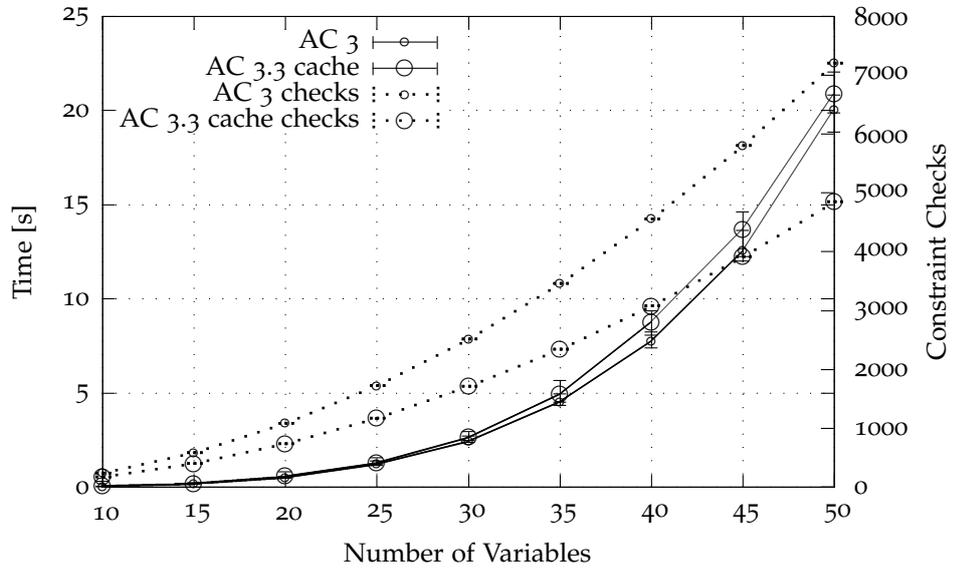


Figure 24. Binary Constraint Performance for Solution of Identity Problems

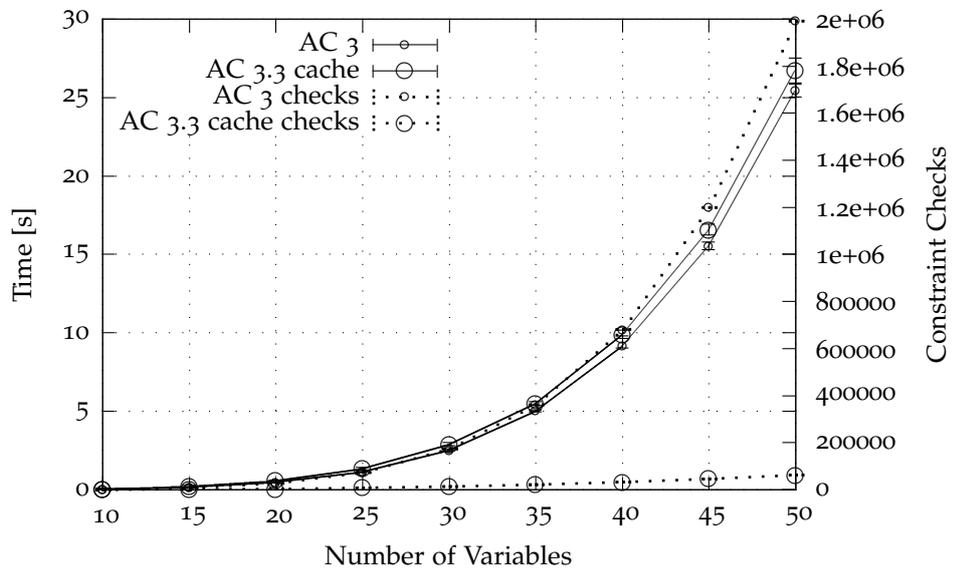


Figure 25. Binary Constraint Performance for Solution of Ordering Problems

A.2 ALL DIFFERENT CONSTRAINTS

There are several algorithms to enforce different levels of consistency of the all different constraint. Overviews can be found in [vHo1] and [Ana04].

Recently developed algorithms achieve range or bounds consistency and assume that the domains of the variables involved consist of an interval of values [Lec96] [Pug98] [MToo] [LOQTVBo3]. The problem class which is subject to this thesis does not have intervals as domains, but discrete values. A mapping of these discrete values to intervals to be able to use algorithms for range or bounds consistency would be highly artificial and not always possible. Therefore, none of these algorithms has been implemented.

To assess the performance of the employed algorithm to enforce consistency on the binary decomposition of the all different constraint (cf. section 5.4.2), the algorithm developed by Régin to enforce hyper arc consistency has been implemented [Ré94].

More sophisticated approaches to enforcing consistency, such as lazy or randomised filtering [KH06], are not considered, as they would be beyond the scope of this thesis.

The methodology and the results of the comparison are presented below.

A.2.1 METHODOLOGY

To compare the performance of the implemented algorithms, three types of problems were examined,

- dense problems with n variables with identical domains of size n ,
- random problems with n variables, each with a domain of up to n random values from the interval $1..n$, and
- a variation of the “pathological problem” [Pug98] with n variables and domains ranging from $\{1\}$ for the first variable to $\{1, \dots, n\}$ for the n th variable.

All the variables were assembled into a single all different constraint and the time taken to find the first solution to the problem measured. For the random problems, no distinction between solvable and unsolvable problems was made. A study [GMP⁺98] suggests that although many random problems become insolvable as the problem size increases, the effect is reduced by reasonably large domains, as chosen here.

The methodology for conducting the measurements and the machine used were the same as described before in section A.1.1.

A.2.2 RESULTS

The measured times taken to solve the problems are depicted in diagrams 26 to 28.

For dense and random problems (figures 26 and 27 respectively) the simple approach of enforcing local consistency on the binary decomposition of the all different constraint clearly outperforms the more complex approach of enforcing hyper arc consistency. Not only is the constant overhead smaller, but the time taken to solve a problem grows slower with increasing number of variables. The hyper arc consistency algorithm does significantly more work during each individual revision, but is unable to prune significantly more values than the local consistency algorithm for these classes of problems. In fact, for dense problems, enforcing local and hyper arc consistency results in the same values being pruned.

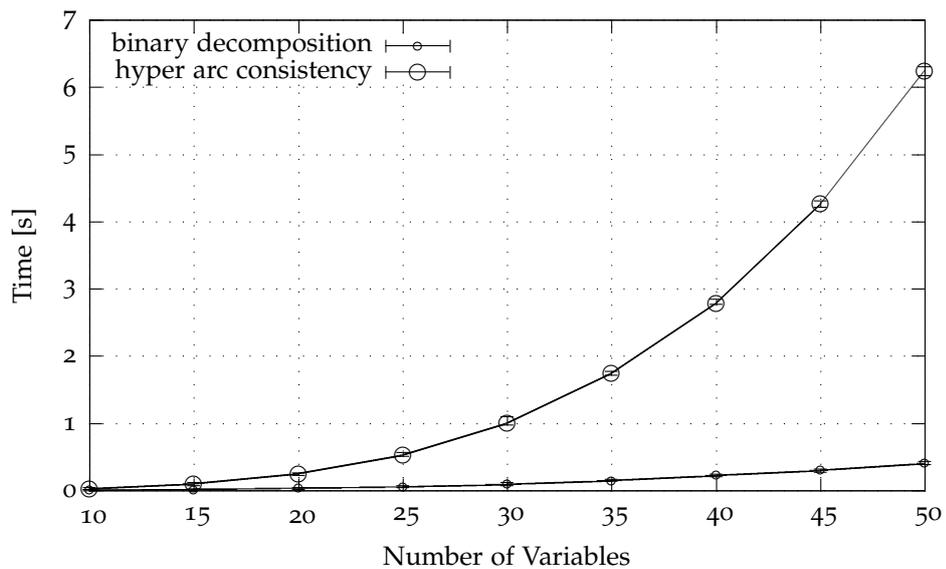


Figure 26. All Different Performance for Solution of dense Problems

The class of “pathological” problems (figure 28) clearly favours hyper arc consistency. This particular class of problems allows only one solution. All the values which are not part of the solution can be pruned by enforcing hyper arc consistency during the first revision of the constraint, i.e. the hyper arc consistency algorithm does not perform any work after the first revision. For a small number of variables the overhead of enforcing hyper arc consistency is too large to achieve better performance than local consistency, but for larger problem instances local consistency of the binary decomposition is clearly outperformed.

The nature of the problems which are subject of this thesis is closest to random problems, therefore the algorithm to enforce local consistency on the binary decomposition of the all different constraint was chosen for the solver.

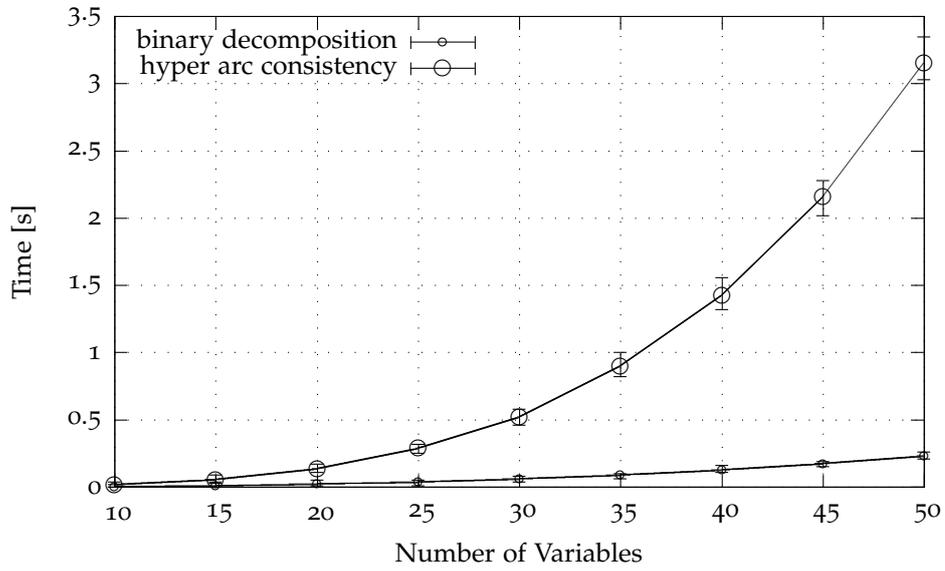


Figure 27. All Different Performance for Solution of random Problems

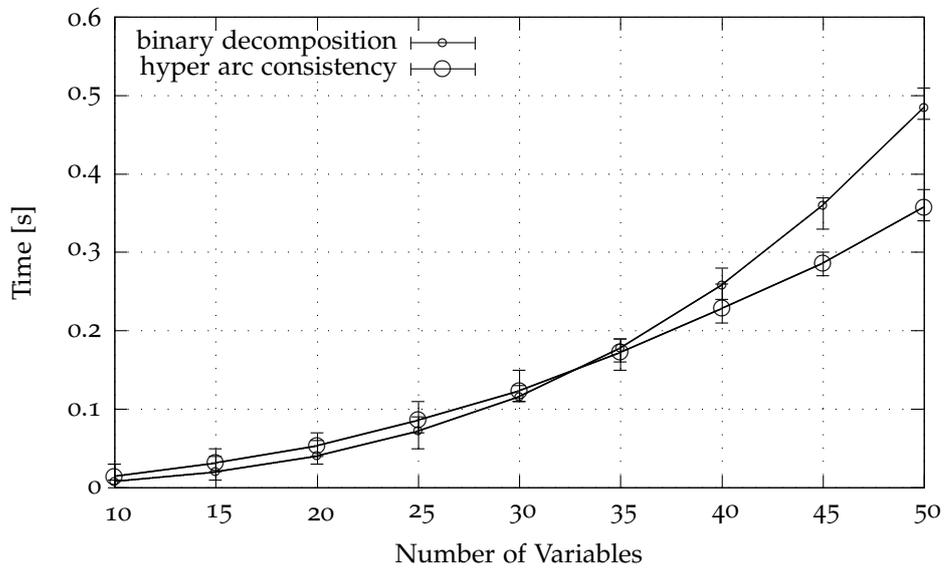


Figure 28. All Different Performance for Solution of "pathological" Problems

A.3 THE DIFFERENCE ALL DIFFERENT MAKES

The performance differences between revising an all different constraint as one constraint and the binary constraints resulting from the decomposition individually has been studied in [SW99]. The findings of this performance evaluation are consistent with [SW99] and presented in diagram 29. It is clearly visible that enforcing and maintaining consistency on the all different constraint outperforms the algorithms for enforcing and maintaining consistency on the binary decomposition by several orders of magnitude.

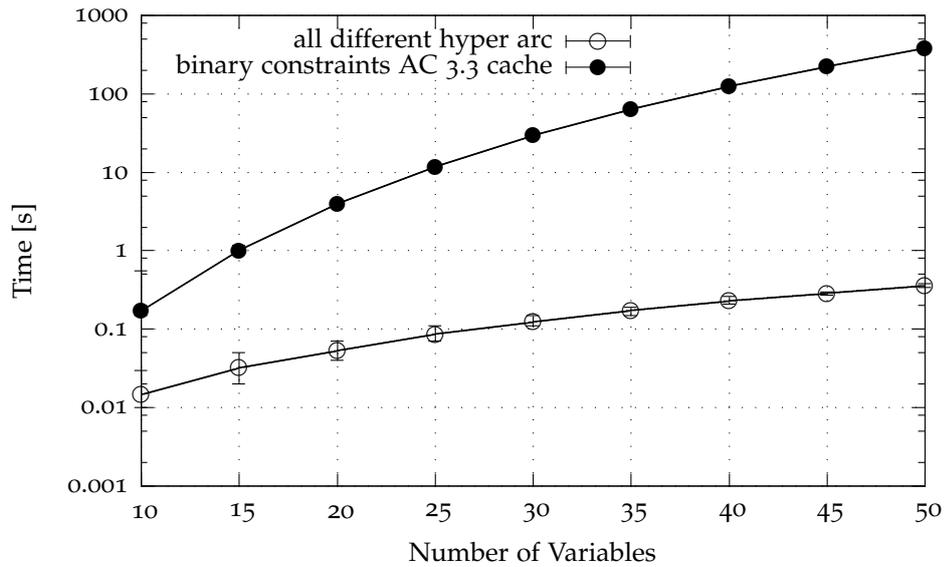


Figure 29. All Different and Binary Constraint Consistency Performance for Solution of “pathological” Problems

B

EFFECTIVENESS OF REAL-TIME CONSTRAINT SATISFACTION

For constraint problems with real-time requirements, it is crucial that the solver returns a solution within the specified time. The effectiveness of the implementation of real-time constraints with regards to this quality is evaluated here.

Two classes of time limits have to be distinguished; before a solution has been found and after a solution has been found. Both cases will be analysed, with special emphasis on the first case. Furthermore, a distinction between soft constraint problems and hard constraint problems will be made because of the different times constraints are dropped.

B.1 METHODOLOGY

The performance was evaluated with two different problem classes,

- a class with n variables with identical domains of size n and $n \cdot (n - 1)$ binary \neq constraints requesting that the value of every variable is different from the values of every other variable and
- a class with n variables with identical domains of size n and $n \cdot (n - 1)$ binary $=$ constraints requesting that the value of every variable is the same as the values of every other variable.

For soft constraint problems, every constraint was assigned a random cost of violation from the interval $(1..n)$.

The two problem classes represent hard and easy problems. The problem class which requires all variables to have equal values is very easy to solve, as after an initial assignment all other values can be pruned from all domains and the solver reaches the solution without any further constraint propagation. When all variables must have different values, only one value at a time can be pruned. The resulting search tree is very large and after each assignment step all domains have to be pruned of the new value.

The time limits were determined by averaging the time required to find the first solution to a particular problem instance over 100 runs of the solver and multiplying it by a factor. The factor 1.5 was chosen for the time limit after a solution has been found, the factors 0.3, 0.4, 0.5, 0.6, 0.7, and 0.8 were chosen to represent time limits before a solution has been found at different stages of the solution process.

The time the solver returned a solution was averaged over 100 runs for a particular problem instance. The deviation from the time limit was recorded. Additionally, the deviation from the time limit was recorded as a percentage of the time limit.

The experiments were conducted on a 1.4 GHz Intel Pentium M Processor machine with 1024 MB RAM. The installation of Ruby was custom-compiled with optimised compiler settings.

B.2 RESULTS

B.2.1 TIME LIMIT AFTER FIRST SOLUTION

The deviations from the time limit for a limit after the first solution has been found are depicted in diagrams 30 to 33.

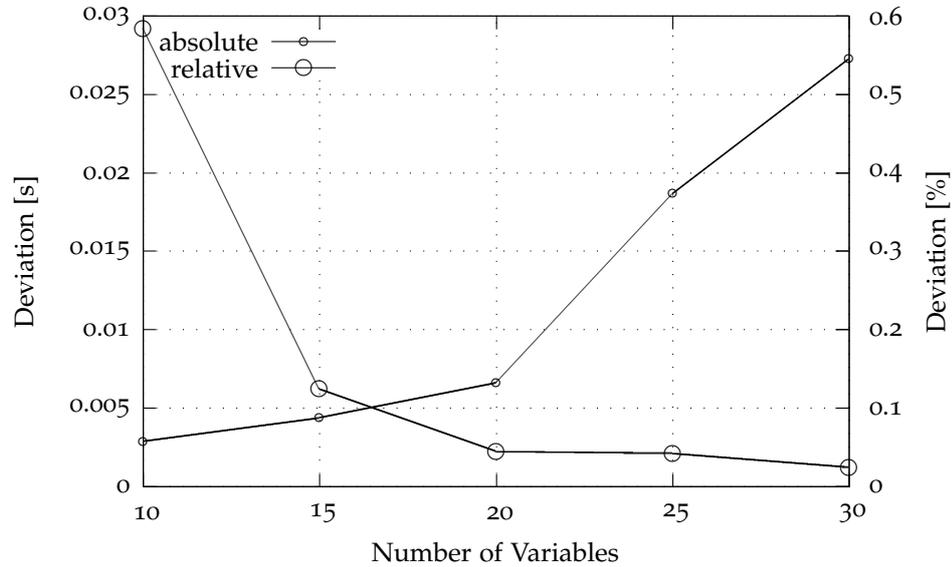


Figure 30. Deviation from the Time Limit after a Solution has been found for all different Problems with hard Constraints

As can be seen in figures 30 and 31, the performance for all different problems is very good. For hard constraints, the accuracy is within less than one percent deviation from the specified time limit and improves with increasing problem size. For soft constraints, the deviation is bigger than for hard constraints for small problems, but still within acceptable limits. As the problem size increases, the deviation quickly decreases to less than one percent of the specified time limit.

The performance for the identity problem depicted in figures 32 and 33 is not as good as for the all different problem. Hard and soft constraints show an equally bad deviation of up to almost 30%. The deviation even increases with increasing problem size.

The accuracy for time limits after the first solution has been found depends on the type of the particular problem. The harder the problem and the bigger the search tree, the better the performance. The identity problem prunes most of the initial search tree in early stages of the solution process, which appears to have a negative effect on real-time constraint satisfaction performance.

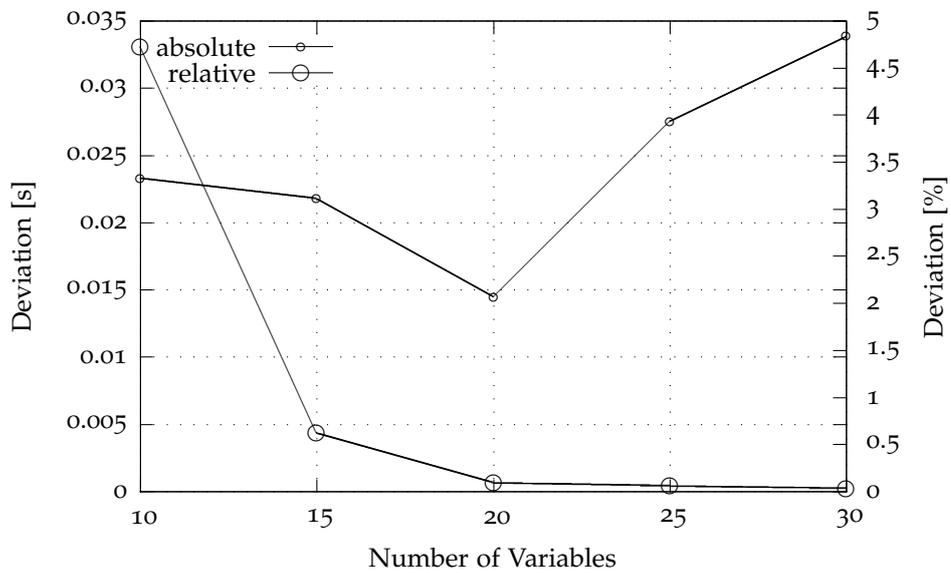


Figure 31. Deviation from the Time Limit after a Solution has been found for all different Problems with soft Constraints

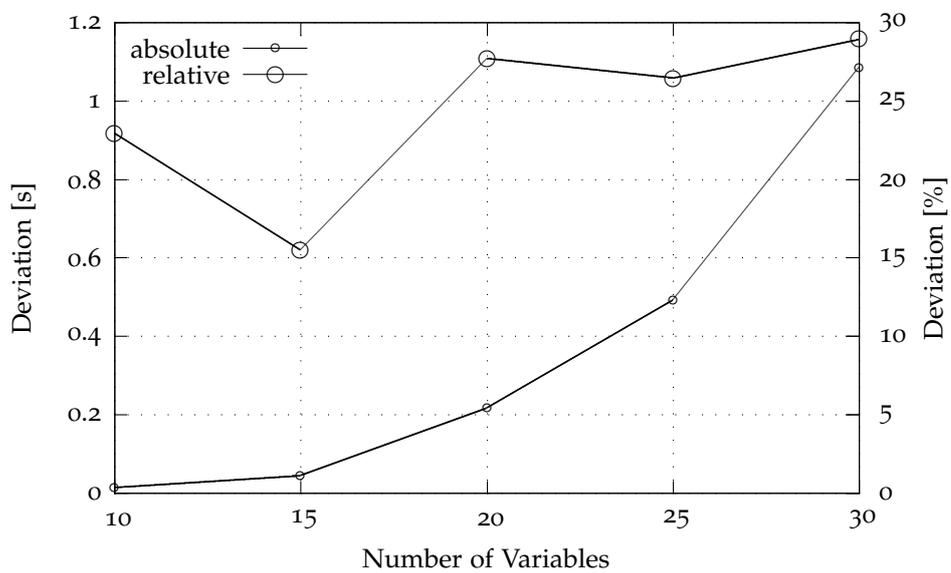


Figure 32. Deviation from the Time Limit after a Solution has been found for Identity Problems with hard Constraints

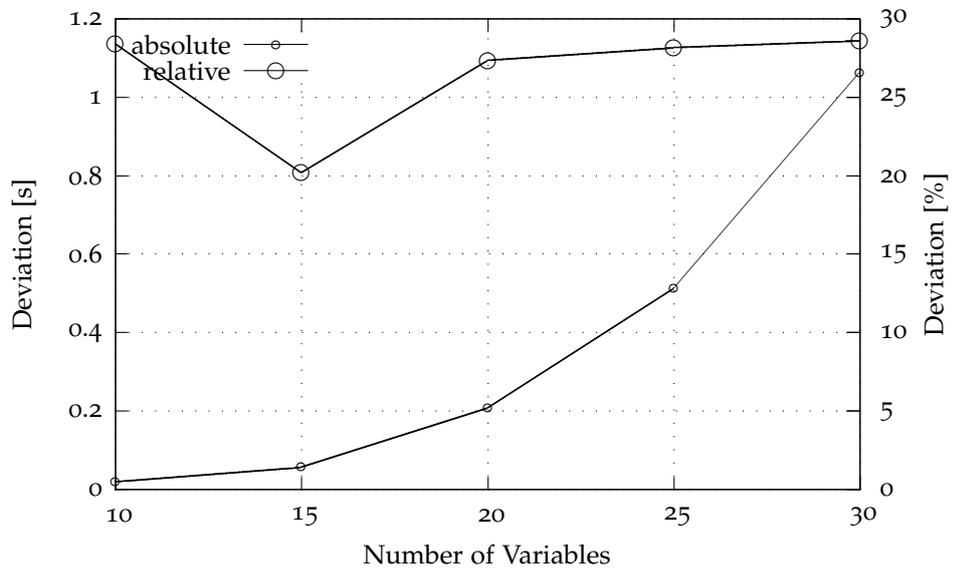


Figure 33. Deviation from the Time Limit after a Solution has been found for Identity Problems with soft Constraints

B.2.2 TIME LIMIT BEFORE FIRST SOLUTION

The deviations from the time limits for limits before the first solution has been found are depicted in diagrams 34 to 37.

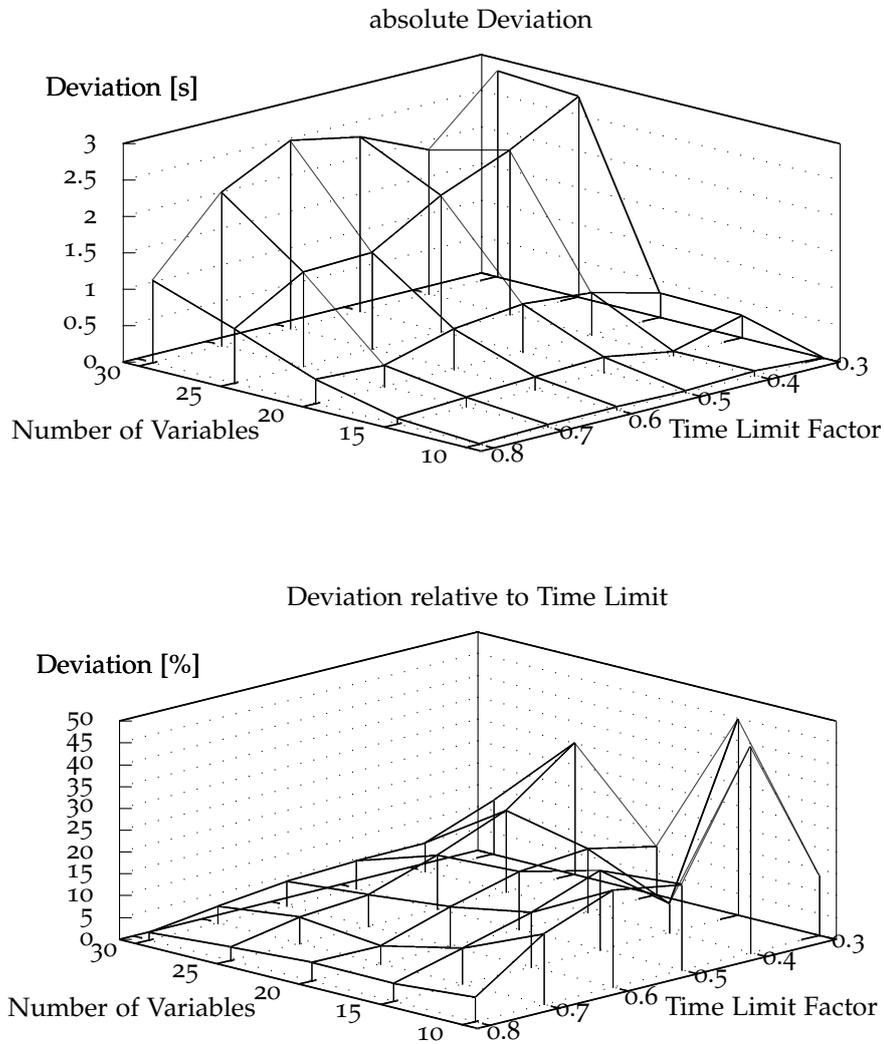


Figure 34. Deviation from the Time Limits before a Solution has been found for all different Problems with hard Constraints

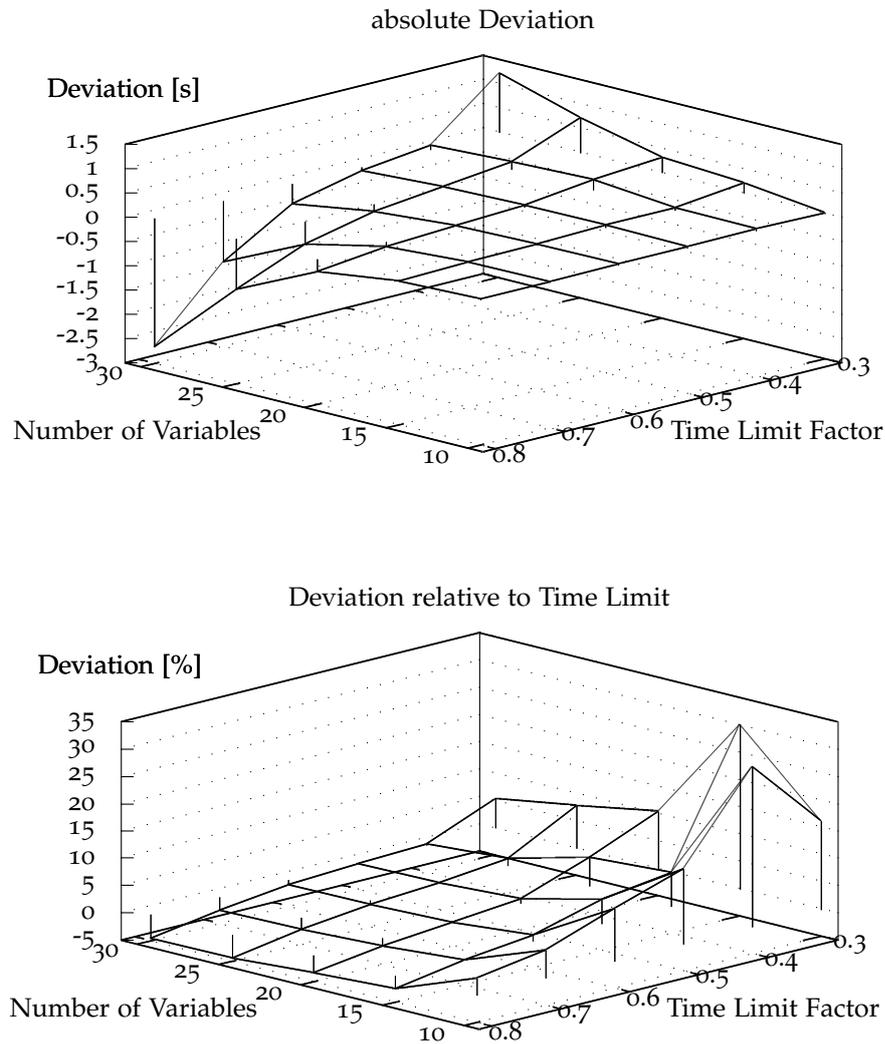


Figure 35. Deviation from the Time Limits before a Solution has been found for all different Problems with soft Constraints

Figures 34 and 35 show that the performance for time limits before the first solution has been found is comparable to the performance after the first solution has been found for all different problems. The deviation is high for small problems and very early termination, but for large problems and termination after half the time required to solve the problem has passed the solver returns even earlier than requested with soft constraint problems, denoted by the negative value. It does not only fulfil the real-time requirements, but adds a safety margin for the calling application. For hard constraints, the accuracy is not as good but the deviation is still less than 10% for large problems and late termination.

The results for the identity problem for time limits before the first solution has been found are even worse than the results for time limits after the first solution

has been found. When the solver is terminated early, it takes more than twice as long as requested, rendering it unusable to fulfil real-time requirements under these conditions. Even for late termination the deviation from the specified time limit is significantly more than 10%.

The implementation language of the solver, Ruby, has a significant impact on the performance. It does not focus on run time efficiency, so the overhead for dispatching method calls and similar is large. For small and easy problems, it constitutes a substantial part of the time required to find a solution. The intervals between the time checks are too small compared to the constant overhead to fulfil the requirements.

The main application of the real-time constraint satisfaction mechanism is the solving of large and hard problems. Easy and small problems are often better solved using application-specific algorithms. The accuracy of the solver for hard problems is very good, especially because the performance increases with problem size.

Further research is required to improve the performance for small and easy problems. The current implementation provides a solid base for further investigation and improvements.

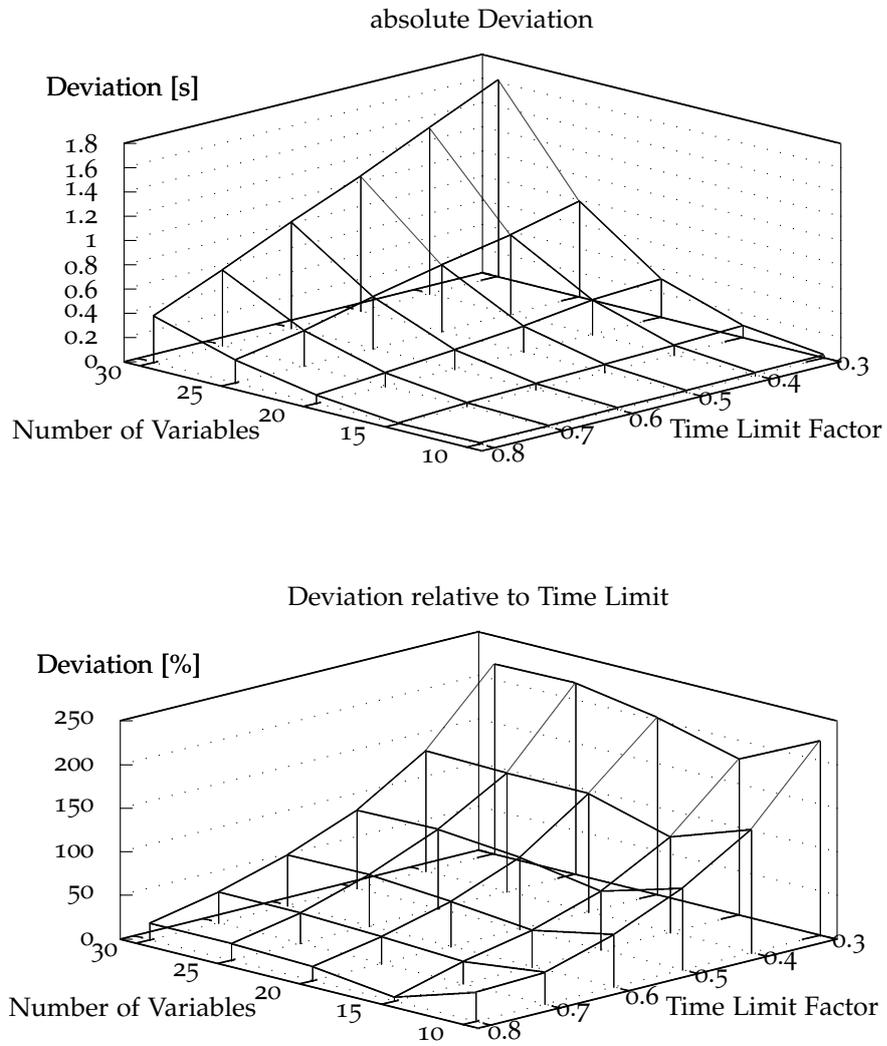


Figure 36. Deviation from the Time Limits before a Solution has been found for Identity Problems with hard Constraints

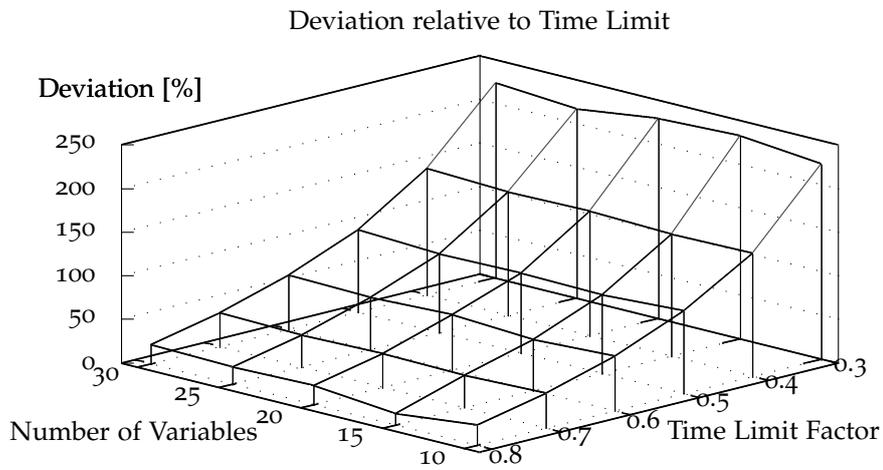
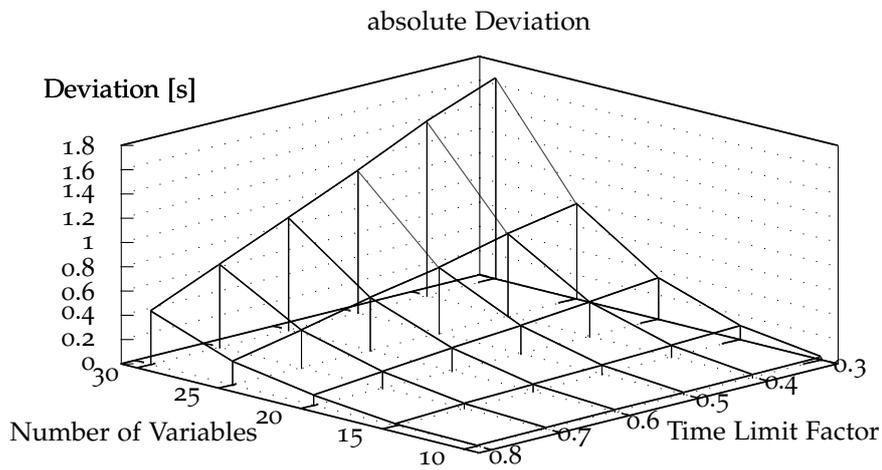


Figure 37. Deviation from the Time Limits before a Solution has been found for Identity Problems with soft Constraints



INSTALLATION INSTRUCTIONS AND SOFTWARE VERSIONS

This appendix describes how to install the parts of the implementation and lists the versions of the software and libraries used during development.

C.1 GENERAL

All the parts of the prototypical implementation were implemented in Ruby. The version of Ruby used was 1.8.5. There are Ruby gems for all parts which can be generated using the provided Rakefiles. The version of Rake used was 0.7.3 and the Gems version was 0.8.11.

The documentation for the source code is generated with diagrams illustrating the dependencies. The GraphVizR package is required to make this work, the version used was 0.4.0.

The distribution packages of the implementation parts can be installed through the Ruby Gems system.

C.2 CONSTRAINT PROBLEM SOLVER LIBRARY

The constraint problem solver library uses the log4r library. The version used during development was 1.0.5.

C.3 CONSTRAINT PROBLEM SOLVER SOAP WRAPPER

The SOAP wrapper uses the SOAP4R library included with the standard Ruby distribution. It furthermore uses the log4r library. The standalone script uses the daemons library, version 1.0.5.

The SOAP server must be run standalone, either in the foreground with the script `ConstraintSolverServer`, or through the daemons library with the script `ConstraintSolverServer-daemons`. The server is configured in the first script and listens on port 3333 for connections from the local host only by default. The port must also be specified in the WSDL of the service for the client to work.

C.4 WEB USER INTERFACE

The web user interface was built using the Camping web application framework. The version of Camping used was 1.5. The HTML is generated with Markaby, version 0.5. The database backend is a SQLite database, the `sqlite3-ruby` library, version 1.1.0.1, was used. The interface to Amazon.com is realised by the `ruby-amazon` library, version 0.9.0 was used during development.

The web frontend can either be run standalone by calling the `camping` binary on `ConstraintSolver.rb` or be integrated with a webserver which is able to handle

FastCGI scripts. The file `dispatch.rb` provides the interface to the FastCGI server.

The SOAP server must be running for the frontend to work. The host and port to connect to are configured in the service WSDL.

GLOSSARY

C

commit Submission of a set of changed files to a revision control system.

Common Gateway Interface (CGI) Standard protocol for interfacing external application software with an information server, commonly a web server.

D

daemon Computer program which runs in the background, rather than under the direct control of a user.

Design Pattern Template solution for a common problem in software design.

domain specific language (DSL) Programming language for a specific set of tasks within a limited problem domain.

F

FastCGI (FCGI) CGI protocol which instead of spawning a new process for every request uses persistent processes to handle multiple requests.

G

gem Package of the RubyGems package management system which can be used to install, update and remove software written in the Ruby programming language.

H

Hypertext Markup Language (HTML) Markup language most widely used for the creation of web sites.

Hypertext Transfer Protocol (HTTP) Method used to transfer or convey information on the World Wide Web.

M

Model-View-Controller (MVC) Design pattern which separates data access, business logic, and data presentation.

R

reflection Process by which a computer program of the appropriate type can be modified in the process of being executed, in a manner which depends on abstract features of its code and its runtime behaviour.

Remote Procedure Call (RPC) Technology which allows a computer program to cause a procedure to execute on another computer on a shared network.

Representational State Transfer (REST) Simple communication interface which transmits domain-specific data over HTTP without an additional messaging layer such as SOAP.

revision control system Management of multiple revisions of the same unit of information.

s

Service-oriented Architecture (SOA) Architecture which uses services to support the requirements of business processes and users.

SOAP Communication protocol for exchanging XML-based messages over computer networks, normally using HTTP.

Structured Query Language (SQL) Computer language used to create, retrieve, update and delete data from relational database management systems.

u

Unified Modelling Language (UML) Modelling language used to specify and describe software systems, user interactions, and business processes.

w

Web Service Software system designed to support interoperable machine to machine interaction over a network.

Web Services Description Language (WSDL) XML-based language used for the description of web services.

x

eXtensible Markup Language (XML) General-purpose markup language.

XML-RPC RPC protocol which uses XML and HTTP as a transport mechanism.

XML Schema Set of rules to which an XML document must conform in order to be considered 'valid' according to that schema.

BIBLIOGRAPHY

- [AF03] Slim Abdennadher and Thom Frühwirth. *Essentials of Constraint Programming*. Springer, 2003.
- [Ana04] Basileios Anastasatos. Propagation algorithms for the alldifferent constraint. 2004. (Cited on page 89.)
- [Apt03] Krzysztof Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.
- [Bar99] Roman Barták. Constraint programming: In pursuit of the holy grail. In *Proceedings of Workshop of Doctoral Students '99*, 1999. <http://kti.ms.mff.cuni.cz/~bartak/html/publications.html>. (Cited on page 43.)
- [Bar03] Roman Barták. Modelling soft constraints: A survey. *Neural Network World*, 12(5):421–431, 2003. (Cited on page 18.)
- [BB01] Nicolas Barnier and Pascal Brisset. FaCiLe: A functional constraint library. *ALP Newsletter*, 14(2), May 2001. (Cited on page 44.)
- [BB04] Nicolas Barnier and Pascal Brisset. *FaCiLe: A Functional Constraint Library*, 1.1 edition, September 2004. <http://www.recherche.enac.fr/opti/facile/doc/>. (Cited on page 44.)
- [BC94] Christian Bessière and Marie-Odile Cordier. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65(1):179–190, 1994. (Cited on pages 16 and 85.)
- [BCFR04] J. Christopher Beck, Tom Carchrae, Eugene C. Freuder, and Georg Ringwelski. Backtrack-free search for real-time constraint satisfaction. In *Principles and Practice of Constraint Programming - CP 2004*, pages 92–106, 2004. (Cited on page 20.)
- [BDFB⁺87] Alan Borning, Robert Duisberg, Bjørn N. Freeman-Benson, Axel Kramer, and Michael Woolf. Constraint hierarchies. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 48–60, 1987. (Cited on page 18.)
- [BFM⁺96] Stefano Bistarelli, Hélène Fargier, Ugo Montanari, Francesca Rossi, Thomas Schiex, and Gérard Verfaillie. Semiring-based CSPs and valued CSPs: Basic properties and comparison. In Michael Jampel, Eugene Freuder, and Michael Maher, editors, *Over-Constrained Systems (Selected papers from the Workshop on Over-Constrained Systems at CP'95, reprints and background papers)*, volume 1106, pages 111–150. 1996. (Cited on page 18.)
- [BFR99] Christian Bessière, Eugene C. Freuder, and Jean-Charles Régin. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107(1):125–148, 1999. (Cited on page 16.)

- [BMMW89] Alan Borning, Michael Maher, Amy Martindale, and Molly Wilson. Constraint hierarchies and logic programming. In Giorgio Levi and Maurizio Martelli, editors, *Proceedings 6th International Conference on Logic Programming, Lisbon, Portugal, 19–23 June 1989*, pages 149–164. The MIT Press, Cambridge, MA, 1989. (Cited on page 18.)
- [BMR95] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Constraint solving over semirings. In Chris Mellish, editor, *IJCAI'95: Proceedings International Joint Conference on Artificial Intelligence, Montreal, 1995*. (Cited on page 20.)
- [BMR97] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Semiring-based constraint satisfaction and optimization. *Journal of the ACM*, 44(2):201–236, 1997. (Cited on page 18.)
- [BMR⁺99] Stefano Bistarelli, Ugo Montanari, Francesca Rossi, Thomas Schiex, Gerard Verfaillie, and Hne Fargier. Semiring-based CSPs and valued CSPs: Frameworks, properties, and comparison. *Constraints*, 4(3):199–240, 1999. (Cited on page 18.)
- [BO03] Hachemi Bannaceur and Aomar Osmani. Computing lower bound for max-csp problems. In *IEA/AIE: Proceedings of the 16th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, pages 614–624, 2003. (Cited on page 17.)
- [Boo93] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 1993. (Cited on page 48.)
- [BR97] Christian Bessière and Jean-Charles Régin. Arc consistency for general constraint networks: preliminary results. In *Proceedings of IJCAI-97*, pages 398–404, 1997. (Cited on page 61.)
- [BR01] Christian Bessière and Jean-Charles Régin. Refining the basic constraint propagation algorithm. In *Proceedings of the 14th IJCAI*, pages 309–315, 2001. (Cited on pages 16, 62, and 85.)
- [BRYZ05] Christian Bessière, Jean-Charles Régin, Roland H. C. Yap, and Yuanlin Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2):165–185, 2005. (Cited on page 85.)
- [BS06] Michael Benisch and Norman Sadeh. Examining dcsp coordination tradeoffs. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1405–1412, New York, NY, USA, 2006. ACM Press. (Cited on page 80.)
- [BSKC97] Nicolas Beldiceanu, Helmut Simonis, Philip Kay, and Peter Chan. White paper: The CHIP system. http://www.cosytec.com/production_scheduling/chip/pdf/the_chip_system.pdf, April 1997. (Cited on page 44.)
- [Bur92] Steve Burbeck. Applications programming in Smalltalk-80(TM): How to use model-view-controller (MVC). 1992. <http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html>. (Cited on page 69.)
- [CdGS07] Martin C. Cooper, Simon de Givry, and Thomas Schiex. Optimal soft arc consistency. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*, 2007. (Cited on page 20.)

- [CDK91] Zeev Collin, Rina Dechter, and Shmuel Katz. On the feasibility of distributed constraint satisfaction. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence, IJCAI-91, Sidney, Australia*, pages 318–324, 1991. (Cited on page 80.)
- [Che76] Peter Pin-Shan Chen. The entity-relationship model – toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976. (Cited on pages viii and 70.)
- [Cho] *Choco User Guide*. http://choco-solver.net/index.php?title=User_guide. (Cited on page 44.)
- [CJ98] Assef Chmeiss and Philippe Jégou. Efficient path-consistency propagation. *International Journal on Artificial Intelligence Tools*, 7(2):121–142, 1998. (Cited on page 16.)
- [CN88] Wesley W. Chu and Patrick Ngai. A dynamic constraint-directed ordered search algorithm for solving constraint satisfaction problems. In *IEA/AIE '88: Proceedings of the 1st international conference on Industrial and engineering applications of artificial intelligence and expert systems*, pages 116–125, New York, NY, USA, 1988. ACM Press. (Cited on page 58.)
- [Com] *Comet Reference*. <http://www.cs.brown.edu/people/pvh/comet/reference.html>. (Cited on page 44.)
- [CS04] Martin Cooper and Thomas Schiex. Arc consistency for soft constraints. *Artificial Intelligence*, 154(1–2):199–227, 2004. (Cited on page 20.)
- [DCC06] Douglas Downing, Michael Covington, and Melody Mauldin Covington. *Dictionary of Computer and Internet Terms*. Barron's Educational Series Inc., Woodbury, NY, USA, 2006.
- [Deco3] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003. (Cited on page 9.)
- [DF97] Rina Dechter and Daniel Frost. Backtracking algorithms for constraint satisfaction problems: a survey. Technical report, UCI, 1997. (Cited on page 12.)
- [DFP94] Didier Dubois, Hélène Fargier, and Henri Prade. Propagation and satisfaction of flexible constraints. *Fuzzy Sets, Neural Networks and Soft Computing*, pages 166–187, 1994. (Cited on page 18.)
- [DKL01] Rina Dechter, Kalev Kask, and Javier Larrosa. A general scheme for multiple lower bound computation in constraint optimization. *Lecture Notes in Computer Science*, 2239:346–361, 2001. (Cited on page 17.)
- [Dono4] Marc R.C. Van Dongen. Saving support-checks does not always save time. *Artificial Intelligence Review*, 21(3–4):317–334, 2004. (Cited on page 86.)
- [Dun93] Tim Duncan. A review of commercially available constraint programming tools. Technical Report AIAI-TR-149, University of Edinburgh, 1993. (Cited on page 43.)

- [FL93] H el ene Fargier and J er ome Lang. Uncertainty in constraint satisfaction problems: A probabilistic approach. In *Proceedings of the European Conference on Symbolic and Qualitative Approaches to Reasoning and Uncertainty (ECSQARU)*, pages 97–104. Springer-Verlag, 1993. (Cited on page 18.)
- [FM94] Eugene C. Freuder and Alan K. Mackworth, editors. *Constraint-based reasoning*. MIT Press, Cambridge, MA, USA, 1994.
- [FMW01] Alan M. Frisch, Ian Miguel, and Toby Walsh. Modelling a steel mill slab design problem. In *Proceedings of the IJCAI-01 Workshop on Modelling and Solving Problems with Constraints*, pages 39–45, 2001. (Cited on page 15.)
- [Fre78] Eugene C. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21(11):958–966, 1978. (Cited on page 53.)
- [Fre89] Eugene C. Freuder. Partial constraint satisfaction. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, IJCAI-89*, pages 278–283, 1989. (Cited on pages 18 and 19.)
- [Fre97] Eugene C. Freuder. In pursuit of the holy grail. *Constraints*, 2(1):57–61, 1997. (Cited on page 2.)
- [Gec] *Gecode Reference Manual*. <http://www.gecode.org/gecode-doc-latest/index.html>. (Cited on page 44.)
- [GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and Jon Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. (Cited on pages 45 and 48.)
- [GJM06] Ian P. Gent, Christopher A. Jefferson, and Ian Miguel. MINION: A fast scalable constraint solver. In *Proceedings of the Seventeenth European Conference on Artificial Intelligence*, pages 98–102, 2006. (Cited on pages 20 and 44.)
- [GJM⁺07] Ian P. Gent, Christopher A. Jefferson, Ian Miguel, Karen E. Petrie, and Andrea M. Reindl. *Getting started with Minion*, 0.4 edition, March 2007. (Cited on page 44.)
- [GLSS79] Jr. Guy Lewis Steele and Gerald Jay Sussman. Constraints. In *APL '79: Proceedings of the international conference on APL: part 1*, pages 208–225, New York, NY, USA, 1979. ACM Press. (Cited on page 9.)
- [GMP⁺98] Ian P. Gent, Ewan MacIntyre, Patrick Prosser, Barbara M. Smith, and Toby Walsh. Random constraint satisfaction: Flaws and structure. Technical report, 1998. (Cited on page 89.)
- [GNT04] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann, 2004. (Cited on page 9.)
- [GSW99] Ian P. Gent, Kostas Stergiou, and Toby Walsh. Decomposable constraints. In *New Trends in Constraints*, pages 134–149, 1999. (Cited on page 59.)
- [Hal35] Philip Hall. On representatives of subsets. *Journal of the London Mathematical Society*, s1-10(37):26–30, 1935. (Cited on page 59.)

- [Hamo06] Youssef Hamadi. *Disolver: the Distributed Constraint Solver*. Microsoft Research, 2.44 edition, 2006. <http://research.microsoft.com/~youssefh/DisolverWeb/disolver.pdf>. (Cited on pages 44 and 80.)
- [HDT92] Pascal Van Hentenryck, Yves Deville, and Choh-Man Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57(2–3):291–321, 1992. (Cited on page 16.)
- [HE80] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980. (Cited on pages 13 and 53.)
- [Hen89] Pascal Van Hentenryck. *Constraint satisfaction in logic programming*. MIT Press, Cambridge, MA, USA, 1989. (Cited on page 16.)
- [HM05] Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search*. The MIT Press, 2005. (Cited on page 44.)
- [Hua06] Dong Huang. Semantic descriptions of web services security constraints. In *SOSE '06: Proceedings of the Second IEEE International Symposium on Service-Oriented System Engineering (SOSE'06)*, pages 81–84, Washington, DC, USA, 2006. IEEE Computer Society. (Cited on page 2.)
- [HY97] Katsutoshi Hirayama and Makoto Yokoo. Distributed partial constraint satisfaction problem. In *Principles and Practice of Constraint Programming*, pages 222–236, 1997. (Cited on page 80.)
- [Ilo] Ilog constraint solver. <http://www.ilog.com/products/cp/>. (Cited on page 44.)
- [IMMH83] Toshihide Ibaraki, Shojiro Muro, Takeshi Murakami, and Toshiharu Hasegawa. Using branch-and-bound algorithms to obtain suboptimal solutions. *Mathematical Methods of Operations Research*, 27(1):177–202, 1983. (Cited on page 16.)
- [Jea98] Peter Jeavons. Constructing constraints. In *CP '98: Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming*, pages 2–16, London, UK, 1998. Springer-Verlag.
- [JLCo6] Rong-Hong Jan, Ching-Peng Lin, and Maw-Sheng Chern. An optimization model for web content adaptation. *Computer Networks*, 50(7):953–965, 2006. (Cited on page 2.)
- [KH06] Irit Katriel and Pascal Van Hentenryck. Randomized filtering algorithms. Technical report, Brown University, 2006. (Cited on pages 14 and 89.)
- [Koa] Koalog. *An overview of Koalog Constraint Solver™*. <http://www.koalog.com/resources/doc/jcs-overview.pdf>. (Cited on page 44.)
- [Koto7] Lars Kotthoff. *Ruby Constraint Solver Manual*, 2007. <http://conssolv.rubyforge.org/>.
- [KT03] Irit Katriel and Sven Thiel. Fast bound consistency for the global cardinality constraint. In *CP-03: 9th International Conference on Principles and Practice of Constraint Programming*, pages 437–451, 2003. (Cited on page 14.)

- [KT05] Irit Katriel and Sven Thiel. Complete bound consistency for the global cardinality constraint. *Constraints*, 10(3):191–217, 2005. (Cited on page 14.)
- [LBH03] Christophe Lecoutre, Frederic Boussemart, and Fred Hemery. Exploiting multidirectionality in coarse-grained arc consistency algorithms. In *CP-03: 9th International Conference on Principles and Practice of Constraint Programming*, pages 480–494, 2003. (Cited on pages 16, 62, and 85.)
- [Lec96] Michel Leconte. A bounds-based reduction scheme for constraints of difference. In *Constraint-96: Second International Workshop on Constraint-based Reasoning*, 1996. (Cited on pages 16, 62, and 89.)
- [Lem] Michel lemaitre’s constraint solver library. <ftp://ftp.cert.fr/pub/lemaitre/LVCSP/>. (Cited on page 44.)
- [LOQTVBo3] Alejandro López-Ortiz, Claude-Guy Quimper, John Tromp, and Peter van Beek. A fast and simple algorithm for bounds consistency of the all different constraint. In *Proceedings of IJCAI-03*, pages 245–250, 2003. (Cited on pages 16, 62, and 89.)
- [LSPGo6] Ruopeng Lu, Shazia Sadiq, Vineet Padmanabhan, and Guido Governatori. Using a temporal constraint network for business process execution. In *ADC ’06: Proceedings of the 17th Australasian Database Conference*, pages 157–166, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc. (Cited on page 2.)
- [LW66] Eugene L. Lawler and David E. Wood. Branch-and-bound methods: a survey. *Operations Research*, 14(4):699–719, 1966. (Cited on page 11.)
- [LZBFo4] Chavalit Likitvivanavong, Yuanlin Zhang, James Bowen, and Eugene C. Freuder. Arc consistency in MAC: A new perspective. In *Proceedings of CPAI’04 workshop held with CP’04*, pages 93–107, 2004. (Cited on page 85.)
- [Mac75] Alan K. Mackworth. Consistency in networks of relations. Technical report, University of British Columbia, Vancouver, BC, Canada, 1975. (Cited on pages 13, 16, 58, and 59.)
- [MF85] Alan K. Mackworth and Eugene C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25(1):65–74, 1985.
- [MH86] Roger Mohr and Thomas C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986. (Cited on page 16.)
- [Migo6] Ian Miguel. CS4402 Constraint Programming lecture notes. University of St Andrews, 2006. (Cited on page 9.)
- [MS98] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction*. MIT Press, Cambridge, MA, USA, 1998.
- [MToo] Kurt Mehlhorn and Sven Thiel. Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint. In *Principles and Practice of Constraint Programming*, pages 306–319, 2000. (Cited on pages 16, 62, and 89.)

- [PCM⁺06a] Alun Preece, Stuart Chalmers, Craig McKenzie, Jeff Pan, and Peter Gray. Handling soft constraints in the semantic web architecture. In *Proceedings of WWW 2006 Workshop Reasoning on the Web (Row 2006)*, Edinburgh, UK, May 2006. (Cited on page 2.)
- [PCM⁺06b] Alun Preece, Stuart Chalmers, Craig McKenzie, Jeff Z. Pan, and Peter Gray. A semantic web approach to handling soft constraints in virtual organisations. In *ICEC '06: Proceedings of the 8th international conference on Electronic commerce*, pages 151–161, New York, NY, USA, 2006. ACM Press. (Cited on page 2.)
- [Pfa03] Bryan Pfaffenberger. *Webster's New World Dictionary of Computer Terms*. Webster's New World, 10 edition, 2003.
- [PRB00] Thierry Petit, Jean-Charles Régin, and Christian Bessière. Meta-constraints on violations for over constrained problems. In *Proceedings of The Twelfth IEEE International Conference on Tools with Artificial Intelligence (ICTAI'00)*, pages 358–365, Vancouver, Canada, November 2000. (Cited on page 18.)
- [PRB01] Thierry Petit, Jean-Charles Régin, and Christian Bessière. Specific filtering algorithms for over-constrained problems. In *CP '01: Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, pages 451–463, London, UK, 2001. Springer-Verlag. (Cited on pages 20 and 80.)
- [Pug95] Jean-François Puget. Applications of constraint programming. In *CP '95: Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, pages 647–650, London, UK, 1995. Springer-Verlag. (Cited on page 9.)
- [Pug98] Jean-François Puget. A fast algorithm for the bound consistency of alldiff constraints. In *AAAI '98/IAAI '98: Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, pages 359–366, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence. (Cited on pages 16, 62, 86, and 89.)
- [Pug04] Jean-François Puget. Constraint programming next challenge: Simplicity of use. In *Principles and Practice of Constraint Programming*, pages 5–8, 2004. (Cited on page 68.)
- [Pyc] Python-constraint constraint solver. <http://labix.org/python-constraint>. (Cited on page 44.)
- [QGLOB05] Claude-Guy Quimper, Alexander Golynski, Alejandro López-Ortiz, and Peter Beek. An efficient bounds consistency algorithm for the global cardinality constraint. *Constraints*, 10(2):115–135, 2005. (Cited on page 14.)
- [QW05] Claude-Guy Quimper and Toby Walsh. Beyond finite domains: the all different and global cardinality constraints. In *CP-05: 11th International Conference on Principles and Practice of Constraint Programming*, pages 812–816, 2005. (Cited on page 14.)

- [Rég94] Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *AAAI '94: Proceedings of the twelfth national conference on Artificial intelligence (Volume 1)*, pages 362–367, Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence. (Cited on pages 16 and 89.)
- [Rég96] Jean-Charles Régin. Generalized arc consistency for global cardinality constraint. In *AAAI '96: Proceedings of the thirteenth national conference on Artificial intelligence*, pages 209–215, Menlo Park, CA, USA, 1996. American Association for Artificial Intelligence. (Cited on page 14.)
- [Rég02] Jean-Charles Régin. Cost-based arc consistency for global cardinality constraints. *Constraints*, 7(3–4):387–405, 2002. (Cited on page 14.)
- [RN02] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2002. (Cited on page 9.)
- [RPBP01] Jean-Charles Régin, Thierry Petit, Christian Bessière, and Jean-François Puget. New lower bounds of constraint violations for over-constrained problems. In *CP '01: Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, pages 332–345, London, UK, 2001. Springer-Verlag. (Cited on page 20.)
- [RPP02] Jean-Charles Régin, Jean-François Puget, and Thierry Petit. Representation of soft constraints by hard constraints. In *Proceedings of Onzièmes Journées Francophones de Programmation Logique et Programmation par Contraintes*, pages 191–198, 2002. (Cited on pages 18, 20, and 55.)
- [Rum] Wheeler Ruml. Real-time heuristic search for combinatorial optimization and constraint satisfaction.
- [Rut94] Zsofi Ruttkay. Fuzzy constraint satisfaction. In *Proceedings 1st IEEE Conference on Evolutionary Computing*, pages 542–547, 1994. (Cited on page 18.)
- [RvBW06] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming*. Elsevier Science, 2006.
- [Sch92] Thomas Schiex. Possibilistic constraint satisfaction problems or “how to handle soft constraints?”. In *Proceedings of the Eight International Conference on Uncertainty in Artificial Intelligence*, pages 268–275, Stanford, CA, 1992. (Cited on page 18.)
- [Scho0] Thomas Schiex. Arc consistency for soft constraints. In *Principles and Practice of Constraint Programming*, pages 411–424, 2000. (Cited on page 20.)
- [Scho5] Thomas Schiex. Soft constraints and over-constrained problems. <http://www.math.unipd.it/~frossi/cp-school/Ecole.pdf>, September 2005. First International Summer School on Constraint Programming. (Cited on page 18.)
- [SF94] Daniel Sabin and Eugene C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *PPCP '94: Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming*, pages 10–20, London, UK, 1994. Springer-Verlag. (Cited on pages 14 and 53.)

- [SFV95] Thomas Schiex, Hélène Fargier, and Gérard Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. In Chris Mellish, editor, *IJCAI'95: Proceedings International Joint Conference on Artificial Intelligence*, 1995. (Cited on pages 17, 19, and 20.)
- [Sha92] Stuart C. Shapiro. *Encyclopedia of Artificial Intelligence*. John Wiley & Sons, Inc., New York, NY, USA, 1992.
- [SOA] SOAP/1.1 note. <http://www.w3.org/TR/2000/NOTE-SOAP-20000508/>. (Cited on page 65.)
- [SW98] Klaus Schild and Jörg Würtz. Off-line scheduling of a real-time system. In *SAC '98: Proceedings of the 1998 ACM symposium on Applied Computing*, pages 29–38, New York, NY, USA, 1998. ACM Press. (Cited on page 20.)
- [SW99] Kostas Stergiou and Toby Walsh. The difference all-difference makes. In *IJCAI '99: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 414–419, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. (Cited on pages 48 and 92.)
- [SW05] Martin Sachenbacher and Brian C. Williams. Solving soft constraints by separating optimization and satisfiability. In *Proceedings of the International Workshop on Preferences and Soft Constraints (SOFT-05)*, pages 119–132, 2005. (Cited on page 20.)
- [Tsao2] Edward P.K. Tsang. Constraint satisfaction in business process modelling. Technical Report CSM-359, University of Essex, Colchester, UK, January 2002. (Cited on page 2.)
- [UML] Unified modeling language specification. <http://www.omg.org/technology/documents/formal/uml.htm>. (Cited on page 45.)
- [vHo01] Willem-Jan van Hoeve. The alldifferent constraint: A survey. 2001. Extended version from <http://www.cs.cornell.edu/~vanhoeve/papers/alldiff.pdf>. (Cited on pages 14, 59, and 89.)
- [vHo04] Willem-Jan van Hoeve. A hyper-arc consistency algorithm for the soft alldifferent constraint. In *CP-04: 10th International Conference on Principles and Practice of Constraint Programming*, pages 679–689, 2004. (Cited on pages 20 and 80.)
- [vHo05] Willem-Jan van Hoeve. Operations research techniques in constraint programming. 2005. PhD Thesis.
- [VLS96] Gerard Verfaillie, Michel Lemaitre, and Thomas Schiex. Russian doll search for solving constraint optimization problems. In *AAAI/IAAI, Volume 1*, pages 181–187, 1996. (Cited on page 17.)
- [Wal96] Mark Wallace. Practical applications of constraint programming. *Constraints Journal*, 1(1):139–168, September 1996. (Cited on page 9.)
- [WF99] Rainer Weigel and Boi Faltings. Compiling constraint satisfaction problems. *Artificial Intelligence*, 115(2):257–287, 1999. (Cited on page 20.)
- [WSD] WSDL/1.1 note. <http://www.w3.org/TR/2001/NOTE-wsdl-20010315>. (Cited on page 67.)

- [YDIK98] Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *Knowledge and Data Engineering*, 10(5):673–685, 1998. (Cited on page 80.)
- [YH00] Makoto Yokoo and Katsutoshi Hirayama. Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems*, 3(2):185–207, 2000. (Cited on page 80.)
- [YHW07] Benjamin Yen, Paul Jen-Hwa Hu, and May Wang. Towards analytical approach to effective website designs: A framework for modeling, evaluation and enhancement. *Electronic Commerce Research and Applications*, 6:159–170, 2007. (Cited on page 2.)
- [YY01] Zhang Yuanlin and Roland H. C. Yap. Making AC-3 an optimal algorithm. In *Proceedings of IJCAI-01*, pages 316–321, 2001. (Cited on pages 16, 62, and 85.)
- [Zhu06] Haibin Zhu. Separating design from implementations: Role-based software development. In *Proceedings of the 5th IEEE International Conference on Cognitive Informatics*, pages 141–148, July 2006. (Cited on page 2.)

COLOPHON

This thesis was typeset with $\text{\LaTeX} 2_{\epsilon}$ using a custom style based on `classthesis` by André Miede.

The illustrations and diagrams were typeset with `PSTricks` and `PDFTricks`, `METAPOST`, and `gnuplottex`. The `hyperref` package was used to create links and cross-references, `bibtex` for the bibliography, the `tocloft` package for the list of figures, tables, and definitions, and `glosstex` for the glossary.

Final Version as of 15th August 2007, 14:07.

DECLARATION

Ich versichere, daß ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Leipzig, 15. August 2007

Lars Kotthoff